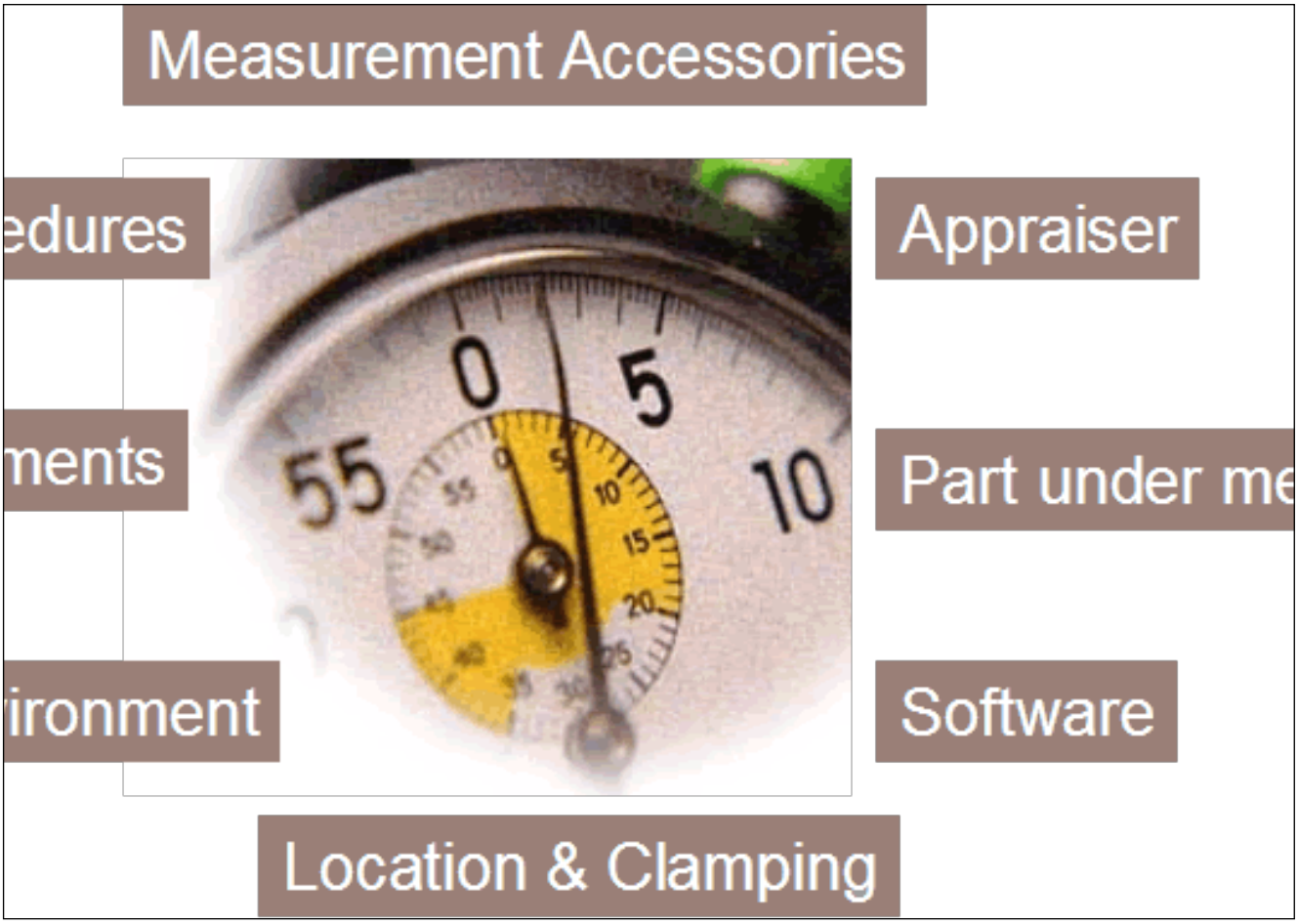
Systems Measurement

CSE 221, Fall '14

Kaushik Kalyanaraman
kkalyana@ucsd.edu
December 13, 2014



Introduction

Operating Systems are an essential part of almost every modern computing hardware. They serve primarily as a resource manager, an interface between the abstract underlying hardware and other higher level programs / user. Individual management of hardware resources requires extensive knowledge of system-level components and would have been a herculean task for all but operators intimately familiar with the architecture. The abstraction of these resources by the OS into a familiar, machine independent, usable environment has led to the proliferation of applications across different platforms and architectures.

In this paper, we seek to profile, report and analyze the various underlying characteristics of the hardware which in turn influence and constrain the services of the operating system. The hardware is comprised of various underlying components such as the CPU, RAM, cache, Disk, Network. Many of these components are tightly integrated into one another (cache and CPU) and affect the overall performance of the system in myriad ways. These resources dire"ctly impact the performance of user level applications too, most importantly, the “responsiveness” of the application. Benchmarking a system may also lead to crucial insights into potential performance bottlenecks, and going by Amdahl's Law, improving critical bottlenecks may lead to high gains in performance.

Most of these benchmarks have been implemented in **C++**, on **Mac OS X 10.10**, using **Apple LLVM version 6.0 (clang-600.0.51)**. Compiler optimizations in LLVM have been turned off using the flag -O0 and the system has been restricted to one active core, with HT disabled using the XCode Instruments.app. A group of two (Kaushik Kalyanaraman and Priyanka Ganapathy) was involved with this project and Kaushik Kalyanaraman has spent about 30 hours in the CPU, Scheduling and OS Services, Memory and the Network Part. Priyanka Ganapathy has spent another 25 hours in the File system and report writing part.

Table 1: Hardware Information

Hardware Type	Description
Processor	Intel(R) Core(TM) i5-4258U@ 2.40GHz
Cores	2 Physical, 4 Logical
L1 I-Cache	32K 8-way set associative, 64-byte line size
L1 D-Cache	32K 8-way set associative, 64-byte line size
L2 Cache	256K 8-way set associative, 64-byte line size
Unified L3 Cache	3M 12-way set associative, 64-byte line size
Memory Bus Speed	1600 MHz
RAM	4096M Dual Channel DDR3 SDRAM 11-11-11-28
Disk Name	Apple SSD SD0128F (PCIe)
Disk Capacity	120.47GB
Disk Info	256MB DDR DRAM buffer (Marvel-88SS9174 disk controller)
Network Card Speed	AirPort Extreme (0x14E4, 0x112) - Up to 450 Mbit/s
Operating System	Mac OS X 10.10 (Yosemite)

Machine Description

I. CPU, Scheduling and OS Services

1. Measurement Overhead

1.1 Estimation

For measuring the overhead of reading time, we use the mach_absolute_time() method mach/mach_time.h which provides the finest granularity of measuring time by providing the number of clock ticks since machine start. This method, defined in [1] performs an inlined call to the method "fast_get_nano_from_abs()" method which in turn executes inlined assembly and reads the time from the rdtsc() register. We estimate the overhead of retrieving time to be about 13 cycles because retrieving time from RDTSC translates into reading the actual register into 32-bit integers, right shifting the LSB and then performing a logical OR of

both and returning that value. Each of these operations take 1 cycle on an average, and two additional if statements are part of this function call which can be expected to take another two cycles. Finally, function call and return may take at least 6 cycles (assuming that this code is not always inlined) and hence, an estimate of about 13 cycles seems reasonable.

1.2 Methodology

The program is bound to one core, running with the highest priority and HT is disabled. Two consecutive calls to mach_absolute_time() are made one after another and we assume that these underlying instructions are not reordered in the CPU and executed serially, one after another. Subtracting the value returned by the first call from the value returned by the second call gives us one value of the time overhead. This is placed into a loop which runs 250,000 times and the averaged out overhead is reported as the actual overhead of reading time on this machine. We chose this method because this is the finest granularity time measurement available in OS X and any other time measurement (such as clock()) is susceptible to coarse granularity and hence, not enough accuracy. mach_absolute_time() provides nanosecond level granularity and there is no function finer than this measure. It also requires the least overhead to be calculated (reads directly from hardware) and hence seems most reliable among the available methods.

1.3 Results

Measurement	Time (in ns)
Mean Overhead	6.076ns
Std. Dev Overhead	3.449ns

1.4 Analysis and Discussion

Two calls to mach_absolute_time() are initiated simultaneously, taking care to ensure that all other unnecessary applications have been shutdown, so that these two instructions are not reordered and are executed consecutively on the processor.

We call these two functions about **250,000** times in a loop and average the observations over all iterations in order to report a better statistical average overhead of reading time. Going by the implementation of mach_absolute_time() here [1], it is extremely fast and provides nanosecond level granularity. Our results (with a mean overhead) of 31 nanoseconds can be translated into clock cycles (on a 2.4GHz processor, 1 clock cycle takes 0.4166 nano seconds => 6 nanoseconds translates into about 14 clock cycles which is reasonably close to our

estimate. Something to note here is the std. dev which equates to about 8 clock cycles and this tells us that there seems to be quite some variance in the data collected. Also, the actual value is more than the predicted by a few clock cycles because we did not consider the loop overhead in the overhead calculation and so this could possibly be the contributing factor to those additional cycles.

2. Loop Overhead

2.1 Estimation

Running a loop involves three basic steps, initialization of a loop variable, computing the result of a condition on the loop variable and then performing an arithmetic operation on the loop variable. Initializing the loop variable is a single time cost done only once during the loop initialization. It does not impact the average running time of a loop over several iterations. The second step translates into a conditional jump in assembly (cmpl and jae) and so can be executed in 2 cycle. Incrementing the loop variable is just an ADD operation and takes another cycle to execute, so we estimate the overhead of a loop to be 3 cycles.

2.2 Methodology

The loop overhead involves a slightly complicated operation: even after turning off compiler optimizations, loops are being optimized either by being unrolled, by being eliminated as dead code by the hardware and/or by a two-level predictor with a 32-bit GHB on the Intel core i5. Running the same loop 250,000 times provides an average runtime for each loop.

2.3 Results

Measurement	Time (in ns)
Mean Overhead	3.413

2.4 Analysis and Discussion

A mean overhead of 3.413 nanoseconds translates into approximately 8 cycles which is more than what we anticipated and estimated. We think this is because there seem to be some additional instructions involved while setting up a loop and not just the ones we estimated. Looking at the assembly version of the loop, it involves the following instructions,

```
movl   $0, -68(%rbp)
LBB2_1:                                 ## =>This Inner Loop Header: Depth=1
        movl   -68(%rbp), %eax
```

```
        cmpl   -4(%rbp), %eax
        jae    LBB2_4
## BB#2:                        ##   in Loop: Header=BB2_1 Depth=1
        jmp    LBB2_3
LBB2_3:                        ##   in Loop: Header=BB2_1 Depth=1
        movl   -68(%rbp), %eax
        addl   $1, %eax
        movl   %eax, -68(%rbp)
        jmp    LBB2_1
```

Thus, according to this version of the loop, about 8-9 cycles seem to be a good indication of the actual runtime of a loop and hence, that is what we observe.

3. Procedure Call Overhead

3.1 Estimation

A procedure call comprises of quite a few steps. The procedure arguments are saved onto the stack, a new stack frame is allocated for this function call, the stack pointer is incremented to the next highest unused address, the return address is also pushed onto the stack and any local variables are allocated space in this stack. On returning from the stack, the return address is popped from the stack into EIP, the procedure's activation record is also destroyed from the stack and the stack pointer is decremented by the size of this procedure's activation record. For all these operations, we estimate about 4 - 5 cycles per operation and in all, about 20 - 30 cycles per zero argument procedure call. Also, for a procedure call with one argument, an additional assembly instruction performed is movl -1636(%rbp), %edi whereas for a procedure call with two arguments, two additional assembly instructions are performed,

```
movl   -1636(%rbp), %edi
movl   -1640(%rbp), %edx
```

For each additional argument, there is one additional instruction and thus, we expect just about a cycle increase in its overhead.

3.2 Methodology

Eight procedures have been created, each with an increasing number of 32-bit int arguments, starting with zero all the way up to seven with only a return statement in the body. We call each procedure 250,000 times and average the result of those measurements to report one measurement for each corresponding procedure call overhead. The loop overhead is also subtracted from each of these measurements in order to obtain more accurate measurements.

3.3 Results

Number of Arguments	Overhead (in cycles)
0	25.2627
1	26.1713
2	26.6398
3	27.5422
4	27.7148
5	28.5264
6	31.5264
7	32.1614

3.4 Analysis and Discussion

The cost of adding an additional argument would be the order of a cycle, since most of the above mentioned procedures are the same, with only the inclusion of one or two additional commands to handle the extra argument. We notice that the results are in sync with what we predicted (about 20-30 cycles) for each procedure call and an average of a 1 cycle increase per increase in argument.

4. System Call Overhead

4.1 Estimation

A system call is a very expensive operation which includes all of the following operations [2]:

1. A privileged context switch - in which the kernel has to save the entire state of the current process including open file descriptors, any in-memory data structures, call stacks, activation records, virtual memory etc. and then switch to a kernel context
2. A trap into a fixed address OS interrupt vector which contains a list of system calls to interrupt values
3. A kernel supervisor routine which takes control, decides what system call has been invoked, checks for correctness of arguments etc.

Mac OS X has ultra fast system calls as well as non-ultra fast system calls [3]. A non-ultra fast system call basically does very little work, has no context switch overhead and returns quickly. Taking all of this into account, we estimate about 300-400 cycles to perform a context switch into the kernel, and about 200-300 cycles for a trap into the OS interrupt vector, perform argument checking, actually execute the system call, compute the results, return the results and another 300-400 cycles to context switch back into the parent process. On average, we estimate about 1000 cycles for a non glibc cached system call.

4.2 Methodology

We use the stat() system call to report the overhead of running a system call. Since idempotent system calls such as getpid() are cached by glibc, micro-benchmarking the system call overhead by running the same system call multiple times gives us incorrect results because only the first call is actually a trap into the OS, subsequent calls are just fetched from cache. We run a loop 250,000 times and call the system call stat() in this loop. The result of this operation is then averaged over all the iterations and the loop overhead is subtracted to provide the final result.

4.3 Results

We notice that the average cost of the system call stat() is about 1062.25 cycles.

4.4 Analysis and Discussion

A system call is indeed an expensive operation and involves several steps. Because of this, it is expensive and glibc tries to cache it for faster access. The observation is pretty consistent with what we estimated once again and the reasons are explained in the estimation section. We specifically used stat() here because we initially tried using getpid() and getppid() and both ended up being cached by glibc and we ended up getting incorrect results for those system calls. Also stat() is a lightweight system call because it just reads a file's inode information and returns it, does not perform any computations.

5. Task Creation Time

5.1 Estimation

We estimate the creation time of a thread to be at least 20x faster than that of a process because threads are lightweight process that share the same address space as that of the parent process but the creation of a new process requires setting up of its own address space, virtual memory etc. We are not sure as to what operations exactly go on under a thread

creation in POSIX, the assembly just calls callq _pthread_create. Looking at the source of _pthread_create in [15], we were able to see a huge number of operations involved in creating a thread i.e., operations such as setting a stacksize, a guardsize, a stackaddr, scheduling policy, scheduling parameter, allocate stack, TLS, TCB, copying the stack guard canary etc. A very rough estimate of the thread creation time, based on all of this information would be about 100000 cycles. Since a process is much, much slower than that and it involves creation of its address space, descriptor tables, virtual memory etc., we estimate its speed to be about 20-30x slower than a thread creation.

5.2 Methodology

We create a posix thread using pthread_create() and then call pthread_join() on that thread so that the main thread waits for that dummy thread to finish execution. The time for creating 100 threads is measured and averaged over all these values by using mach_absolute_time() before creation and after joining. A process is created using the fork() system call and in this child process we call exit() to immediately exit while we wait on this child process, in the parent process, to exit, using the call waitpid() and measure the time before calling fork() and after calling waitpid(). This is measured about 1000 times and the results averaged over all these runs.

5.3 Results

Operation	Time (in cycles)
Thread Creation	95560
Process Creation	1742520

5.4 Analysis and Discussion

The estimates are in line with observed values. The process creation time is about 25x slower than thread creation time. In this experiment, we ensure that everything is running on one core and HT is turned off, and as many applications have been terminated as possible. Only the underlying benchmarking program is being run. This gives us fair confidence in our values.

6. Context Switch Time

6.1 Estimation

A context switch is an expensive operation, involving several steps such as saving and restoring processor registers, reloading TLB entries, flushing the processor pipeline etc [4]. Cache pollution or cache interference is another factor while considering the context switch time. We do estimate the time for a thread context switch to be less than a process context switch because of the inherent light - weight nature of a thread combined with the fact that it usually shares resources with other threads within the same process. A thread context switch would therefore not involve some of the heavyweight operations such as TLB flushes, processor pipeline flushes etc. About a 30% difference in the context switch times is a reasonable estimate as mentioned in [4] but we observe almost a 10x difference between the two. A thread context switch, based on the above reasoning seems reasonable to have about a few hundred cycles in context switch time (about 300) and a processor context switch to be about 400 cycles.

6.2 Methodology

A process context switch is measured using two processes communicating using pipes (a parent and its forked child process). The direct cost associated by blocking on pipes is then measured by calling mach_absolute_time() before and after reading from the pipe and this context switch is forced about 500 times. Both processes are pinned to the same processor and all other active physical cores have been disabled (so that context switches occur on the same core). Data is written into one end of the pipe in the child process and the parent process blocks on read until the data is written into the pipe and the CPU is yielded to the parent, during which the child blocks on read from the parent. Once the CPU is yielded to the parent, the current clock ticks are measured and a value is written back into the pipe which unblocks the child. This continues for about 500 iterations, resulting in 1000 context switches.
A thread context switch is measured by creating a thread using the pthread library's pthread_create() and then forced to switch to this created thread using the scheduler yield function sched_yield(). Sched_yield() yields the CPU and the current thread is put at the back of the thread scheduling pool. Calling sched_yield() on the other thread once again brings back control to the original thread. Both threads were run on a single core, with multithreading disabled. These observations were also averaged over 500 iterations i.e., 1000 thread context switches.

6.3 Results

Operation	Time (in cycles)
Process Context Switch	3930.78
Thread Context Switch	322.482

6.4 Analysis and Discussion

A context switch is an extensive and expensive operation and there doesn't seem to be any deterministic way of quantifying these numbers other than hypothesizing about the various operations going on underneath when it occurs. Something to note though is that we estimated the process context switch to be about 30% more than a thread context switch time but it looks like more than 10 times the thread context switch time. Probably, TLB flushes and cache interferences are very expensive operations and so we didn't account enough for these context switch penalties. We did not use sched_yield() for measuring the process context switch time because it didn't force process context switches for some reason and we were unable to measure anything significant in that experiment. The blocking pipes forcing process context switches worked perfectly every time.

Memory Operations

7. RAM Access Time

7.1 Estimation

Main memory access latency is defined by the the DRAM timing (RAS-CAS latency, t_{RAS} etc.) and our DRAM has timings of 11-11-11-28 and is a 1600MHz DDR3 SDRAM chip. This DRAM chip is rated to have a 60 ns average latency and so our estimate would be around 70-80 ns due to additional OS overhead of access.

7.2 Methodology

We used the same back-to-back load latency measure used in the lmbench paper[5] (pointer chasing). Hardware and software prefetching of cache lines may interfere with our operations, so we filled a uint64_t array with fixed stride lengths which possibly might exceed the prefetched blocks (a cache line is 64-bytes, a stride of 1K or more potentially skips prefetched cache lines) but short of total randomization of strides, we could not defeat (if

any) additional access pattern algorithms might might optimize the prefetched lines to further reduce access time.

The pseudo-code used for our measurement was:

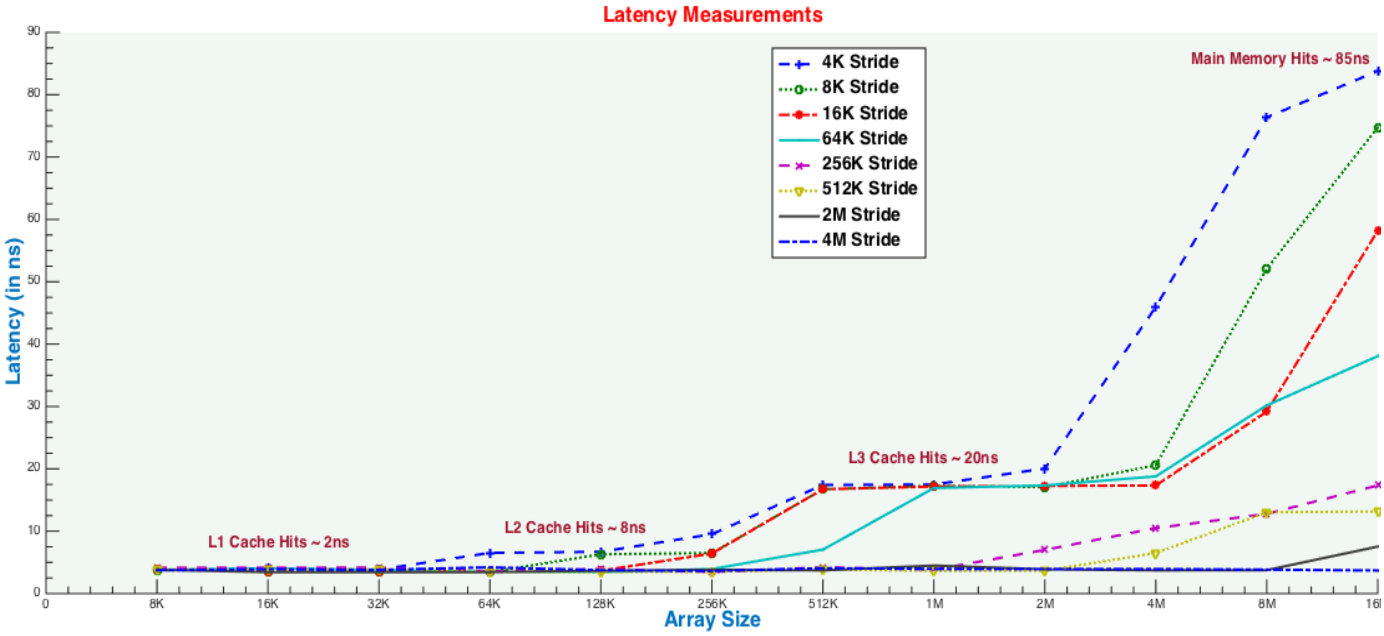
```
start = mach_absolute_time();
for(uint i = 0; i < NUM_SAMPLES; i++)
        p = reinterpret_cast<uint64_t*>(*p);
end = mach_absolute_time();
```

and then averaged all readings over NUM_SAMPLES to obtain the final result. reinterpret_case is just the C++ style version of C style casting and hence has no overhead. In fact, the operation int *p = reinterpret_cast<int*>('a') translates into the two assembly instructions,

movabsq **$97, %rcx**
movq **%rcx, -16(%rbp)**

which uses just 2 cycles to perform the operation. We estimate the L1 cache hit time to be about 2-3 cycles, the L2 cache hit time to be about 8-10 cycles, the L3 cache hit time to be about 15-20 cycles and main memory hit time to be about 50-100 cycles (each higher level being order of magnitude slower than the previous). The L1 D-cache is exclusive to each core and is closest to the processor while main memory is the farthest from the processor in the memory hierarchy.

7.3 Results



Stride / Arr Size	4K	8K	16K	64K	256K	512K	2M	4M
8K	3.96274	3.7561	3.81917	3.826	4.13745	3.85462	3.82508	3.73209
16K	4.02081	3.524	3.44721	4.04499	4.15607	3.85799	3.43943	3.91531
32K	3.84303	3.44447	3.44358	3.81457	4.16223	3.88246	3.44363	3.80024
64K	6.52711	3.44692	3.44814	3.48672	3.55289	3.44777	3.44605	4.21047
128K	6.70607	6.30301	3.67807	3.44828	3.96221	3.44717	3.66737	3.75244
256K	9.52284	6.50269	6.42394	3.94119	3.45035	3.50712	3.8172	3.6164
512K	17.4223	16.7264	16.7295	7.04998	4.22234	3.85269	3.72856	4.07119
1M	17.4939	17.2873	17.1692	16.9257	3.82977	3.64741	4.49462	3.9491
2M	20.0531	17.0793	17.2576	17.3464	6.99774	3.67312	3.89769	3.92317
4M	45.9738	20.5752	17.3439	18.7981	10.5045	6.50443	3.73109	3.93338
8M	76.3649	52.0846	29.1885	30.1836	12.7795	13.0446	3.75738	3.83827
16M	83.8091	74.7569	58.1748	38.0846	17.3528	13.1674	7.54531	3.70682

7.4 Analysis and Discussion

In the graph, we observe that the L1 cache hits (of around 2-3 ns) occur initially for array sizes of 8K-32K and then we observe a spike in the latency at the 32K mark. We deduce that the size of our L1 cache is thus 32K. From 32K to 256K, we see a steady averaged latency of around 10-15 ns and beyond 256K a sharp spike again. We deduce the size of the L2 cache to be 256K. Further, the L3 cache latency is around 25-30 ns and a very sharp spike in latency somewhere between the 2M and the 4M mark and thus taking the average, we infer the size of the L3 cache to be 3M.

An anomaly we notice in the graph is that for larger strides, the latency penalties are not as high as the smaller strides. We are wrapping around the array while striding and this could be a possible reason for lesser and lesser latency. For instance, the 4M stride almost always is an L1 cache hit because a 4M stride wraps around the size of the array to access elements recently accessed, once again.

Intel's spec sheet for the Haswell micro architecture [8], reports that the L1 cache latency is about 4 cycles, the L2 cache latency is about 11 cycles which is very close to the observed

values. The software visible cache latency is obviously dependent on access patterns and a lot of other factors and so we cannot micro benchmark precisely the latencies of different memory levels.

We did not run into any extreme cache contention issues since the L1 cache is not shared (core local) and having disabled the other active core and multi threading, we ensure that this is one of few processes running in the system and hopefully has all the cache to itself.

8. RAM Bandwidth

8.1 Estimation

RAM bandwidth goes up to a maximum write bandwidth of 25.6 GB/s for a dual channel, 1600MHz DDR3 SDRAM [7] and a maximum read bandwidth of 51.2 GB/s for the same setup [8]. Memory bandwidth is computed by the product of the base DRAM clock, the number of data transfers per clock, the bus width and the number of interfaces [9]. We estimate the maximum write bandwidth to be about 20 GB/s taking into account the OS overhead of sending a request to RAM, the RAM pre-charge time etc. and then sending that data back to the operating system. Since we are utilizing a dual channel RAM but running only single threaded benchmarks, we expect our read bandwidth to be of the same amount as the write bandwidth.

8.2 Methodology

We initially went with the simplest approach i.e., repeatedly assigning a value to an array element and averaging its value. The results obtained were less than half the theoretical limit (around 9.34 GB/s). The memory transactions are done in multiples of the cache line width (64-byte) and if we wanted to write data smaller than the cache line size, the cache actually has to read the 64-byte line from memory and then write into it, which results in much more traffic than anticipated.

To avoid this, we tried with non-temporal instructions [10] which directly write into memory and avoid the additional cache read. This gives us a write bandwidth of about 13.65 GB/s. These instructions seem to occur an additional overhead in terms of microcode translation into multiple opcodes (they are not single opcode instructions but multiple opcodes making up a single assembly instruction). Finally, we went with a simple implementation of writing into memory using stosq using asm in c++ [11] and were able to achieve a theoretical **write** bandwidth of about **19.975 GB/s**. The instruction used for this benchmarking was,

asm volatile("cld\n rep stosq" : : "a" (0), "c" (ARR_SIZE / 8), "D" (myArray));

with an 8-byte aligned character array. This is still not close to the maximum theoretical bandwidth and we theorize that additional processes which use the same memory channel might reduce available bandwidth.

The speed of the memory controller and DRAM page conflicts also play an important role in limiting the memory bandwidth. Since our bandwidth experiment was conducted on one thread, in a single process, it wasn't as efficient as it would be if multiple threads accessed memory at the same time (which would be able to obtain the maximum possible throughput from the memory.)

For reads, we tweaked the same instruction a little, to use lodsq instead of stosq,

asm volatile("cld\n rep lodsq" : : "S" (myArray), "c" (ARR_SIZE / 8) : "%eax");

and we observe a **read** bandwidth of about **22.213 GB/s.**

8.3 Results

Experiment	Bandwidth (GB/s)
Write Bandwidth	19.975
Read Bandwidth	22.213

8.4 Analysis and Discussion

Surprisingly, this is less than half the theoretical maximum read bandwidth of 51.2 GB/s and we surmise that the same problems affecting write bandwidth affects read bandwidths also i.e., absence of multithreaded sequential and random access reads, no guarantee of full channel bandwidth for our benchmark etc.

Hardware prefetchers try to predict memory accesses and issue speculative prefetch requests to the predicted addresses before the instructions are actually run [12] so we read an entire 1 GB chunk of memory into a globally allocated char* by using the lodsq / stosq instructions. Additionally, the Intel Haswell micro-architecture has 3 prefetchers [13],
a) A hardware prefetcher which detects access at a uniform stride (within 64B)
b) An adjacent cache line prefetch which pairs cache lines for read access
c) A DCU prefetcher for L1 D-cache prefetching

Our assembly instructions perform a uniform stride > 64B and so does not come under the hardware prefetcher. We disabled the adjacent cache line prefetch by setting **bit 19 of register 1A0h** on our machine. After this, we have taken reasonable assumptions as to the absence of

the influence of the cache prefetcher on our readings. Another point to note is that having a small number of assembly instructions (1/2) ensures that only the first OPCODE fetch is a miss in the L1 instruction cache, subsequent fetches have a high probability of being a cache hit in the L1 I - cache or at some other higher level cache and so the overhead of the processor having to fetch the instruction OPCODE from memory is reduced too.

9. Page Fault Service Time

9.1 Estimation

Initially expected page fault service times were on the order of several milliseconds but then we reviewed the machine specs again and came to the conclusion that our disk is a high end PCIe SSD with a disk latency of around 0.01ms [14]. It does not have any of the overheads of a spinning disk, such as rotational latency, seek latency, command processing time and settle time. The FTL (flash translation layer) does impose a small overhead when writing / reading from disk but does not impact the performance as much as when compared to a spinning disk. Hence, based on these revised configurations, we estimated a page fault service time of about 0.03 ms and the values are much lower than that, about 0.016 ms.

9.2 Methodology

We measure the page fault service time using mmap to map a 1GB file into virtual memory using the command

memMapFile = mmap(0, chunkSize, PROT_READ, MAP_PRIVATE, fd, offset);

where the file is divided into chunks of size chunkSize and and we load each chunk into memory, from the specified offset. By dereferencing the memory object returned by mmap and striding across this object by a size > pagesize (4K), we ensure that subsequent accesses are all page faults and the VM manager brings the pages from disk onto the main memory as well as the TLB.

9.3 Results

On the initial runs through the program, we calculate the average page fault service time to be about **16618.9 ns**. But on going through frequent, multiple runs of the program, the service time reduces significantly, to about **1445.86 ns**

9.4 Analysis and Discussion

We postulate that we observe these results for two reasons:

1) The presence of a TLB in the VM manager. Subsequent page faults are accessed quickly from the TLB and addresses are quickly translated from the TLB instead of a page table lookup
2) A soft page fault - Many pages may not be paged out to disk (remaining in main memory or secondary page caching) but whose address mapping is no longer in the TLB, in which case there is no need to load the page from disk but merely update the TLB with the address mapping and re access the in-memory page.

Comparison with main memory access:
Page Fault: 16618.9 * (10^-9) s = 4 * (2^10) b
=> **1B** requires **4.057 ns** to load from disk

Memory: 1 s = 19.975 GB
=> **1B** requires **46.62 ps** to load from memory
which is orders of magnitude faster than disk load

Network Operations

10. Round Trip Time

10.1 Estimation

10.1.1 Loopback - We estimate the loopback round trip time to be of the order of microseconds because packets do not go out of the eth0 interface but instead are copied from one buffer to another on the same machine. This few microseconds overhead is due to the overhead of copying data packets to the buffer, and then the receiving process dequeueing packets from the buffer. An additional overhead is of the socket system calls going through an additional layer of indirection by calling into the kernel but ICMP requests are processed directly at the kernel level. ICMP loopback is expected to be slightly less than measuring RTT through sending TCP packets but by not much (since its only one additional level of indirection). Considering the operations involved in a loopback RTT measurement i.e., the TCP server sending one byte into its buffer, a system call into the kernel to dequeue that buffer into the receiver's buffer, and then the receiver dequeueing its own buffer to receive the packet we estimate the loopback RTT to be about 90 microseconds and the ICMP RTT to be about 60-70 microseconds.

10.1.2 Remote - Remote RTTs are influenced by several factors, such as the physical distance between the two machines, the number of hops to the destination, the congestion present in the routers present in the link, overhead of routing decisions etc. Our packets might also be dropped / have lower priority, there might be additional overhead of NAT, firewall etc and so its very difficult to have a reasonable RTT estimate over the network. For this reason, we give a very rough estimate of remote RTT over one hop, on an ad-hoc network to be about 3-4ms for measuring it using a TCP flow and about 2-3ms for an ICMP measurement of RTT.

10.2 Methodology

For calculating RTT, we use the algorithm for calculating RTT from the TCP flow between two hosts using the TCP timestamp which is mentioned in RFC 1323 [16]. We setup an ad-hoc network between two Macbook Pro Retina's of the same type and then send run run a TCP client on one machine and a TCP server on the other. We send packets of the maximum MTU size (1472 bytes) between the client and server (ping-pong strategy of sending packets back and forth between the two) and then compute the SRTT (smoothed RTT which is an exponential moving average of the RTT) by storing the delays between the sent and received packets on both machines.

10.3 Results

Methodology	Loopback Latency (in ms)	Remote Latency (in ms)
ICMP	0.077	2.13
TCP	0.094	3.412

10.4 Analysis and Discussion

We notice a significant difference between the ICMP remote host RTT and TCP remote host RTT because

1. ICMP goes directly through the kernel while TCP is an additional level of indirection into the kernel
2. TCP has additional connection setup and teardown overheads (SYN, ACK, SYN-ACK) while ICMP does not have any of these overheads.
3. Adding and removing the TCP header (transport layer) are additional processing steps done at the kernel level on the sender and the receiver ends while the relatively lightweight ICMP header is much faster to add and remove than the TCP header.

Loopback is the baseline hardware performance because it involves only the OS overheads of reading from disk, storing it into a processor buffer, dequeueing it from a process buffer and

then computing the RTT. Thus, the OS overhead here is just the creation of the socket, the sys call overhead and reading from disc. Other additional operations may include packet checksumming etc.

11. Peak Bandwidth

11.1 Estimation

11.1.1 Loopback - The loopback estimate should be somewhere around the maximum disk read bandwidth offered by the operating system. In the disk bandwidth section, we estimated the average disk bandwidth (after including OS overheads) is about 10 GB/s and this is what we will estimate as the peak-bandwidth for the loopback. Loopback packets are merely copied into a temporary network buffer and dequeued from this same buffer and never goes out the WLAN interface and hence is primarily limited by the bandwidth of DRAM as well as any additional overheads incurred due to the operating system.
11.1.2 Remote - Since we are on a 1-hop ad-hoc network, remote bandwidth should be close to the max theoretical bandwidth of the 802.11ac wifi card i.e., taking into account the os overhead of adding and removing headers, enqueueing and dequeueing into packet buffers, TCP setup and teardown costs etc we estimate the remote bandwidth to be about 50 MB/s.

11.2 Methodology

An ad-hoc network is hosted to which two Macbook Pro Retina with similar configurations connect to, ensuring that only one hop is present between the two devices. The TCPClient running on one of the machines sends 10packets of 1472bytes to the remote machine, where the TCPServer receives them, records the time and resends (reflects) the same packets back to the TCPClient. The TCPClient records the sending and receiving time and computes the bandwidth using the formula mentioned in RFC 1323 [16]. This value is reported in the following section.

11.3 Results

Interface Type	Bandwidth
Loopback	10.1147 GB / s
Remote	46.84 MB / s

11.4 Analysis and Discussion

The loopback estimated bandwidth, the actual bandwidth and the memory read bandwidth are agreeing with each other. The loopback bandwidth is primarily limited by the DRAM read bandwidth and 0.5 * the DRAM read bandwidth which is ~10 GB/s (because we are not accessing the dual channel RAM via parallel threads / read requests). Thus, there is very few overhead in this measurement other than that of the OS being able to write into the process' buffer and reading back again from the same buffer.
The remote estimated bandwidth is limited by the protocol overhead of adding and removing TCP headers, setting up and tearing down a TCP connection, going out of the WiFi interface and being received and placed into a particular ingress buffer and then being dequeued. All of this overhead contributes quite a bit to the overall bandwidth and so we are able to obtain only about 46 MB/s instead of the advertised 50 MB/s.

12. Connection Overhead

12.1 Estimation

We estimate the connection overhead (TCP setup teardown time, adding and removing TCP headers) to be a few microseconds (because they are fast operations probably implemented in hardware) and a part of the network stack of every operating system in existence today. The TCP setup cost involves sending a SYN, an ACK and a SYN-ACK (3-way handshake) wheres a TCP teardown cost involves sending a closing now packet.

12.2 Methodology

We compare the overhead of sending a packet using a raw socket (SOCK_RAW) vs a TCP socket (SOCK_STREAM) and subtract the time taken for the raw socket from the TCP sock which is reported as the connection overhead. For the raw_socket, we add our own IPv4 header manually and send it over the network. This should provide a reasonable estimate of the TCP overhead of the handshake and header addition / removal process. This is measured about 2000 times and the results averaged to provide an average TCP connection overhead information. This is not measured in loopback because a TCP connection / flow is never setup in the loopback interface case.

12.3 Results

The TCP connection overhead is found to be about **2.134 us**.

12.4 Analysis and Discussion

Considering the actual overheads involved in the TCP 3-way handshaking protocol, the result is pretty consistent with our estimate of a few microseconds for this overhead. It is very small and usually is never a matter of concern for applications (because latency is usually dominated by the packet exchange latency) and users don't pay much attention to it. It becomes a bottleneck in very exceptional cases, such as where the application requires absolutely as close to the ideal hardware performance as possible and so decides to implement its own protocol and stack which is much closer to the hardware and with much less overhead than TCP.

File System Operations

13. File Cache Size

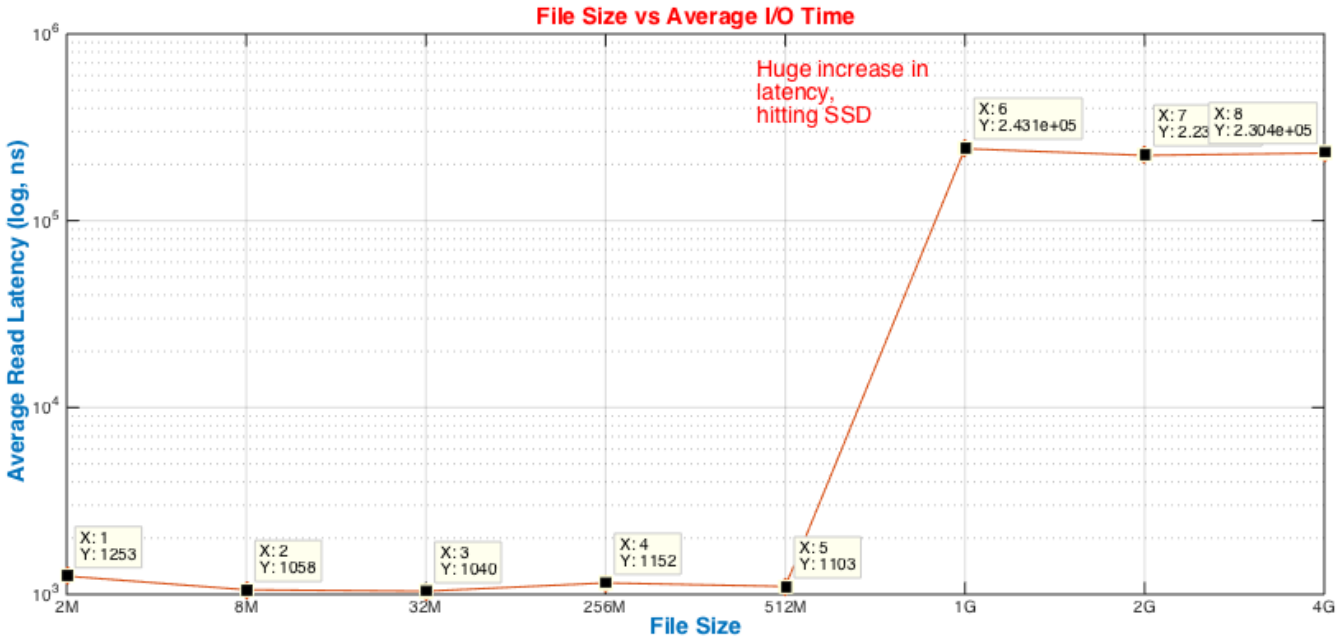
13.1 Estimation

We estimate the average block read for a non-cached file to be somewhere close to the SSD access time of about (150 microseconds) because this is the SSD latency as specified by the manufacturer along with some additional OS file system overhead. The File cache size is estimated to be about 1G because this is a Macbook Pro with only 4G of DRAM and 3G of the memory is wired and compressed memory used by the operating system. We test this hypothesis in the following section and see some surprising results.

13.2 Methodology

We create random .txt files of specific file sizes (from 2M all the way up to 1G) of random content using the command “dd if=/dev/urandom of=512MFile.txt count=1048576” where count specifies a multiple of the device block size. We find out the device block size and OS allocation size using “diskutil info / | grep "Block Size” and we notice that the device block size is 512 bytes and the Allocation Block size is 4096 bytes. We disable file caching using fcntl(fid, F_NOCACHE, 1); and read file blocks of different file sizes (from 2M to 2G) and report the average block read time for each file both cached and non-cached. The file system cache is purged before very run using the command “sudo purge”. Everywhere, we are reading the file blocks using the raw device interface and the read() system call. We read the file backwards by seeking by 2 blocks from SEEK_CUR each time and then incrementing the file pointer by 1 block.

13.3 Results



File Size	I/O Read Time (per block, in ns)
2M	1252.8
8M	1058.29
32M	1040.04
256M	1151.52
512M	1102.56
1G	243124
2G	203731
4G	210358

13.4 Analysis and Discussion

From the graph and results, we see that files up to a size of 512M are entirely in the file cache and so are always cache hits wheres files > 1G are not cached and hence are fetched from disk. As expected, the DRAM latency is around 1100 ns per 4K block read (this is adding to the overhead of the operating system, the file system and other DRAM limitations such as

t_{RAS} etc). An interesting thing to note is that the file cache is probably somewhere between 512M and 1G and is of a variable size (never a constant size) because it is a dynamically adjusted parameter which is controlled by the host operating system and not by any other application or user. We used a reverse file read along with F_NOCACHE option specifically so that we will be able to measure cached file read as well as non-cached file read explicitly. Furthermore, we note that as per our estimation of the SSD access time of about 150 us, the average access time for a block coming from disk is about 200 us with an added overhead of having to add this block to the file cache as well as the OS overhead of having to fetch a 4K block from disk.

14. File Read Time

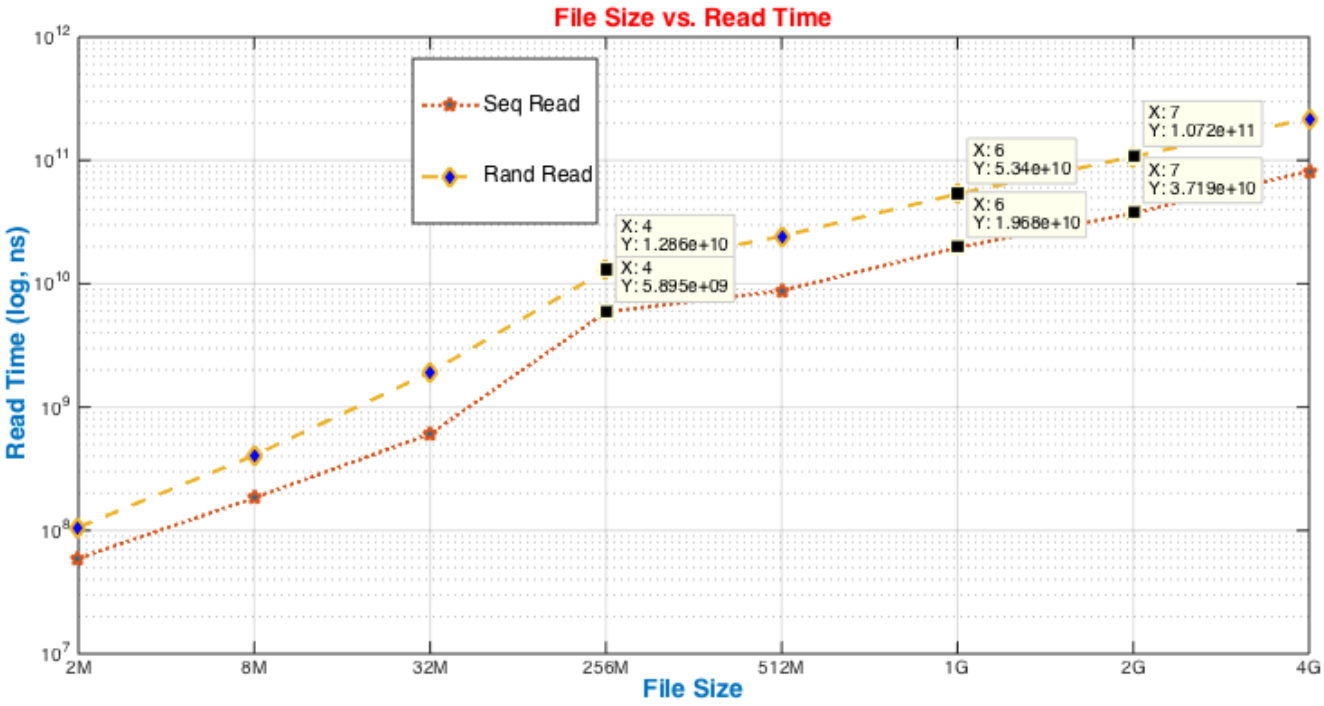
14.1 Estimation

14.1.1 Sequential - We use a high end PCIe SSD and hence sequential or random file read shouldn't really impact our figures. Sequential file reads will be faster due to the SSD controller being able to batch together sequential block reads into a single read from the underlying disk but not by much. Thus, our estimate of the sequential file read time is just based off the (average SSD access time + the OS file system overhead) * number of blocks in the file.
14.1.2 Random - SSDs don't have as much a performance loss when comparing sequential and random access reads as an SSD but its still significant. SSDs can perform very well with a high queue depth. Hence our estimate for random file I/O is about 2x slower than sequential file I/O.

14.2 Methodology

We first off ensure that we read the same number of blocks in sequential and random access so as to have a uniform result. For **sequential file** reads, we run a loop until the maximum number of blocks possible and read an allocation block worth of data each time from the file using the underlying raw read() interface. Time taken to perform the read is measured and totaled in a variable. This variable gives us the total sequential read time for a specific file. For **random file reads**, we perform almost the same operation as the above except two additional operations just before the read system call. We seek to the beginning of the file using lseek() each time before reading, generate a random number between 0 and the total number of blocks on the file and then seek to this block on the file using lseek() once again, before reading from the file. Throughout this operation, the F_NOCACHE option is set on the descriptor to prevent file caching. Thus, all reads are guaranteed to be from the disk and not from in-memory. We total all of these values and then return this as the total read time for random access for each of the files.

14.3 Results



File Size	I/O Seq Read Time (in ns)	I/O Rand Read Time (in ns)
2M	5.94642E+07	1.06246E+08
8M	1.83524E+08	4.0641E+08
32M	6.06282E+08	1.90806E+09
256M	5.89461E+09	1.28649E+10
512M	8.79969E+09	2.42765E+10
1G	1.96784E+10	5.34049E+10
2G	3.71897E+10	1.07222E+11
4G	8.18774E+10	2.18774E+11

14.4 Analysis and Discussion

Our estimation that the random file read time is about 2x that of the sequential read time is about right. As explained before, the sequential read saturates an SSD's queue depth as a result of which it performs better than for random reads but the relative performance gain of

an SSD when comparing sequential and random file reads is not as much as when compared to the performance of a HDD.

15. Remote file read time

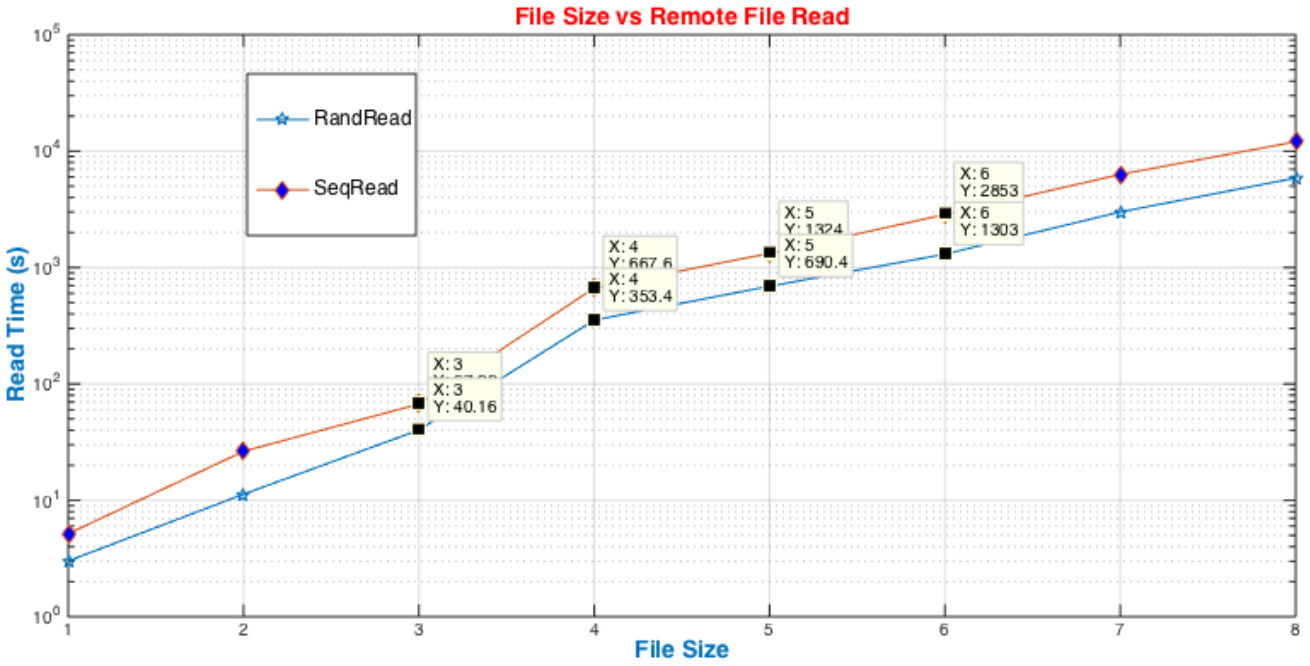
15.1 Estimation

Since we are running a FUSE fs over a 1-hop ad-hoc network but have a severe bandwidth limitation, the estimate of remote file read time is at least 50x that of the local file read. Thus for every particular reading in the previous section, a 50x penalty is applied and estimated as the new remote file read time. We have the basic network over head of both WLAN cards, the TCP overhead in setting up and tearing down connections, the network bandwidth limitation. File system calls are translated into marshaled calls over the network, where they are un-marshaled, and file blocks are retrieved, sent back to the client which adds a lot of overhead to the entire process. Hence, a 50x penalty seems like a reasonable estimate for files being transferred over the network.

15.2 Methodology

SSHFS using FUSE is used as the remote file system, and it is mounted as a local directory. The rest of the methodology remains the same as the local file read section and we see that the results are somewhat according to the estimates.

15.3 Results



File Size	I/O Seq Read Time (in s)	I/O Rand Read Time (in s)
2M	3.012	5.162
8M	11.283	26.364
32M	40.162	67.218
256M	353.432	667.649
512M	690.371	1324.265
1G	1303.113	2853.142
2G	3001.972	6304.773
4G	5876.374	12034.117

15.4 Analysis and Discussion

We see that our estimate was a bit off, in that there is a linear increase in the read time as the file size increases and not much difference between seq and random reads (the remote disc is also an SSD). This is probably because there is a lot of additional overhead we probably did not account for, such as block checksumming, block fetch retry (for corrupted or lost blocks) all of which add to additional costs. We used SSHFS because it seemed like a simple, lightweight network file system implementation and it works for Mac OS X. We did not want a very heavy weight NFS which would add a lot of additional overhead and skew our results. Also as measured earlier, the maximum network bandwidth we were able to achieve was 46.84 MB / s which provides an absolute upper bound of about 90 seconds for a 4G file operation. Thus we are able to see that there is a very significant overhead in transferring files over the network.

16. Contention

16.1 Estimation

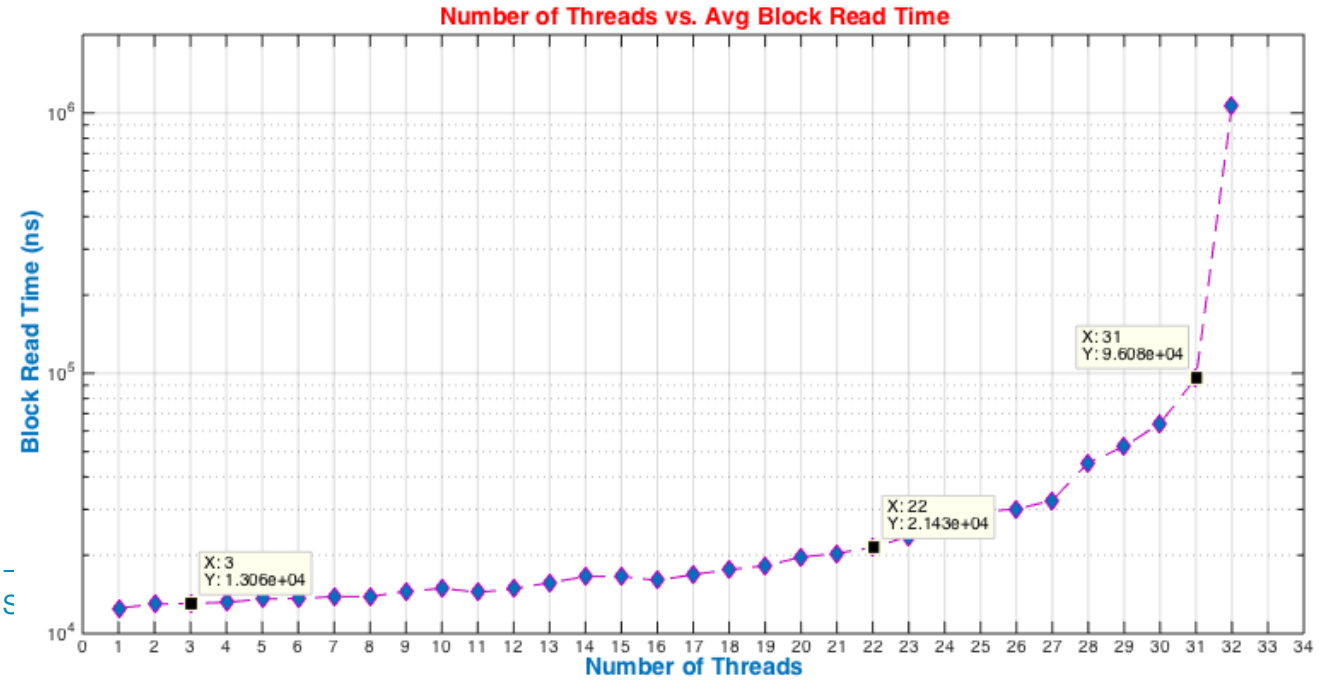
We already estimated the average block read time to be about 100-200 us in the PCIe SSD of our Macbook Pro Retina. We postulate that the average block read time will increase as the number of threads contending to read from the controller increases but this will not be a linear increase. This is because the SSD can handle a good queue depth reasonably well but once the number of outstanding requests exceed the queue depth, the latency will up by a large amount.

16.2 Methodology

We disable file caching using F_NOCACHE on fctl. Up to a maximum of 32 threads read 32 different .dat files of size (32M each) and they read 7000 blocks (of size 4K each) during each iteration. The time required to run is measured using mach_absolute_time() before the thread initialization and just after joining all threads. All file descriptors are reset to zero by using lseek() before the next iteration so that the thread count can be increased for the next iteration. The average block read time is computed using totalTime / (7000 * num_threads).

16.3 Results

TNum	1	2	3	4	5	6	7	8	9
Read Time	12529.8	12962.2	13059.9	13177.9	13567.5	13623.6	13866.3	13884.6	14505
TNum	10	11	12	13	14	15	16	17	18
Read Time	14917.5	14413.5	14824.3	15701.3	16547.4	16597	16002.7	16856.4	17597.3
TNum	19	20	21	22	23	24	25	26	27
Read Time	18234.6	19638.7	20342	21425.5	23656.6	28161.1	29387.2	30076.6	32498.5
TNum	28	29	30	31	32				
Read Time	45418.7	52301.3	64225	96080.8	1066907.2				



16.4 Analysis and Discussion

Multi threading has not been disabled for this experiment and all active cores are running (understandably we want as much parallelism as possible to have contention among threads). From the graph, we can see the SSD controller is able to handle up to 24 threads pretty reasonable well without much increase in the average block read time but beyond 24 the latency just keeps increasing at higher and higher rates. Finally, from 31-32 threads there is a tremendous spike in latency because of queue overflows of the disk controller (because of which the requests are queued in the OS queue instead of the disk queue) which is extremely slow. Also, latency will only increase as more and more threads contend for a share of the disk controller's time. Thus we see expected behavior for this particular hardware and it wasn't very far from the estimate. We can also notice that the average disk block read time of about 100-200 us is as per our previous observations that the average disk block read time is around that amount (disk access time + OS overhead).

References

[1] http://www.opensource.apple.com/source/Libc/Libc-320.1.3/i386/mach/mach_absolute_time.c [Accessed: October 2014]

[2] http://en.wikipedia.org/wiki/System_call

[3] Mac OS X Internals: A Systems Approach. by Amit Singh. Publisher: Addison- Wesley Professional. Release Date: June 19, 2006. ISBN: 9780321278548.

[4] Li, Chuanpeng, Chen Ding, and Kai Shen. "Quantifying the cost of context switch." *Proceedings of the 2007 workshop on Experimental computer science*. ACM, 2007.

[5] McVoy, L. (1996). "lmbench: Portable Tools for Performance Analysis." Usenix ATC.

[6] Cepeda, S. (2009). "What you Need to Know about Prefetching." Intel.

[7] http://www.intel.com/content/dam/www/public/us/en/documents/datasheets/4th-gen-core-family-desktop-vol-1-datasheet.pdf

[8] http://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-optimization-manual.html

[9] http://en.wikipedia.org/wiki/Memory_bandwidth

[10] Drepper, U. (2007). "What Every Programmer Should Know About Memory."

[11] http://www.ibiblio.org/gferg/ldp/GCC-Inline-Assembly-HOWTO.html#ss6.2

[12] http://hps.ece.utexas.edu/pub/srinath_hpca07.pdf

[13] Hegde, R. (2008). "Optimizing Application Performance on Intel® Core™ Microarchitecture Using Hardware-Implemented Prefetchers." Intel.

[14] Norman, L. (2010). "Latency: The heartbeat of a Solid State Disk." SNIA Education Committee.

[15] https://sourceware.org/git/?p=glibc.git;a=blob;f=nptl/pthread_create.c;h=34d83f94ade0cef92f6c9a769b7d65c96c8cfe44;hb=HEAD

[16] https://www.ietf.org/rfc/rfc1323.txt