CSE 221: System Measurement Project

Winter 2014 – Voelker

Andreas Prodromou and Samuel Wasmundt
(A53049230) , (A53053728)

INTRODUCTION

Performance measurements are a critical analysis tool used to determine the effectiveness of the state-of-the-art technologies. This analysis can be done in many different ways, however one very important thing to take into consideration when designing these measuring tools is the overhead of the measuring code itself. To that effect we looked at experiments to better understand how the overhead associated with basic pieces of code can effect the overall measurements taken. The initial experiments conducted target the CPU, Scheduling, and OS Services. Specifically we looked at the following 5 overheads: time measurement, loop, procedure call, system call, task creation, and context switch.

After determining the overheads associated with basic operations, we looked into measurements analyzing the memory and network components of our machine. For memory we looked into measurements to get the RAM access time, RAM bandwidth, and page fault service time. Additionally, for the network we looked at the round trip time, peak bandwidth, and connection overhead. Specifics about these tests, our estimations, what we compared them against, and actual results will be in the following sections. Finally, we will look at the file system and various operations such as: file read time, remote file read time, contention, and size of file cache.

A quick note on the distribution of the work. We share an office, and as a result have worked in tandem on the entire project. If one person had a primary lead on one of the experiments, then the other was the primary code reviewer. As a result, we were both heavily involved in all experiments. The initial tests were jointly completed. The RAM experiments were headed by Samuel and the network experiments by Andreas. We will jointly tackle the file system experiments.

The system we ran our tests on was a small server box with the following system specifications. Most information was obtained straight from the console using /proc/cpu.info and sudo lshdw, other information was also taken straight from the CPU manufacturer's page at intel.com.

- **Processor**
  - Intel(R) Core(TM) i7-3770K CPU @ 3.50GHz.
    - L1 cache: 64 KB (32 KB Instruction cache && 32 KB Data cache).
      - Write Through, 16-way Set-associative
    - L2 cache: 256 KB (per core).
      - Write Through, 16-way Set-associative
    - L3 cache: 8 MB.
      - Write Back
- **Memory Bus**
  - DDR3-1333/1600 (machines RAM – 667 MHz)
  - Channels: 2
  - Width: 64 bits
  - Max Memory Bandwidth: 25.6 GB/s
- **RAM Size**
  - Max Memory Size: 32 Gbs
  - Actual Size: 32 GB (4x8 GB)
- **I/O Bus**
  - Socket: LGA 1155
  - 20 Gb/s DMI 2.0 with x4 link

  - PCI Express 3.0
  - Xeon E3-1200 v2/3rd Gen Core processor DRAM Controller
- **Disk**
  - Western Digital WD10EZEX (x2 in RAID 0)
    - Capacity: 1TB
    - RPM: 7200 RPM
    - 64 MB Cache
    - SATA 6.0 Gb/s
  - SATA controller : 6 Series/C200 Series Chipset Family SATA AHCI Controller
    - vendor: Intel Corporation
    - width: 32 bits
    - clock: 66MHz
    - capabilities: storage ahci_1.0 bus_master cap_list
    - configuration: driver=ahci latency=0
- **Network**
  - Controller: RTL8111/8168/8411 PCI Express Gigabit Ethernet Controller
    - Vendor: Realtek Semiconductor Co., Ltd.
    - Size/Capacity: 1Gbit/s
    - Width: 64 bits
    - Clock: 33MHz
    - Capabilities: bus_master cap_list ethernet physical tp mii 10bt 10bt-fd 100bt 100bt-fd 1000bt 1000bt-fd autonegotiation
    - Configuration: autonegotiation=on broadcast=yes driver=r8169 driverversion=2.3LK-NAPI duplex=full firmware=rtl8168e-3_0.0.4 03/27/12 ip=132.239.95.65 latency=0 link=yes multicast=yes port=MII speed=1Gbit/s
- **Operating System**
  - Platform: Linux
  - Distro: Ubuntu
  - Release: 12.04.4 LTS
  - Codename: precise

CPU, Scheduling, and OS Service

The methodology we used for these tests was a bit of a hierarchical one. The operations done in this section gradually increase in complexity, and the initial measurements are used to measure the later ones. The first task was to get accurate information on the measurement overhead. In order to guarantee the code we wrote was the code that was actually run, and thus measured, we disabled all optimizations. This was necessary because we will run loops which have no useful instructions, but are required to determine the overhead associated with a loop. We do NOT want these optimized away.

As an attempt discover any outliers within the data, we ran each experiment 100 times. We then averaged these results. Further, we calculated the standard deviation to further illustrate the reproducibility of our metrics. Continuing along those lines, we executed with Dynamic Voltage Scaling (DVS) turned off, specifically we set the frequency to be locked at 3.5 GHz using the CPU frequency scaling indicator app provided by the Ubuntu Software Center. Additionally, we ran each experiment on a single core, selecting thread 7 by using “taskset -c 7 <executable> <arguments>”. We chose thread 7 since thread 0 had a lot of load variation due to system processes being run on it. In the rare case that we did see outliers, 2/100 test runs in the process creation test which had a 200% increase in creation time, were simply rerun for accuracy. Clear outliers were then discarded.

MEASUREMENT OVERHEAD

To isolate the overhead associated with reading time, the simplest of benchmarks was created. When measuring the time of the code block of interest we use assembly inline instructions to read the rdtsc register. This function, getticks(), was coded after referencing a forum posting on StackOverflow.com. By having two function calls to getticks() in a row we are able to get the time reported by the first call, however this does not tell us any information because we have no reference point. By placing the second getticks() call immediately after the first, we are able to determine the overhead associated with the second call to getticks() relative to the first call. Thus, by subtracting the second call from the first we are able to directly calculate the overhead associated with calling a single call to getticks(). To produce a more accurate result we instantiate the timers before calling the function calls to avoid any cache interactions that may skew the results.

We predicted the inline assembly instructions to only take a couple of cycles to read the rdtsc register, note there is no overhead associated with calling the “function” because it is an inline'd call! This is then followed by a 32 bit shift to make sense of the assembly call, which should take 50-100 cycles if done sequentially in a vonNeuman style. This overhead is then coupled with the delay to actually schedule and commit the assembly code, which could range from 25-50 cycles, so we estimate an overall overhead in the range of 100-200 cycles.

The measured overhead is 312 cycles. This is fairly close to our estimation of the overhead, we suspect the rdtsc register may require a privileged mode and as a result additional cycles may be lost to kernel code needing to be run. The following table represents our measurements.

Task	Time (microseconds)	Time (cycles)	Standard Deviation
Reading Time	.089	312.33	.00426
Loop Overhead	.001857	6.5	6.05 E-6

To measure the overhead with loops we build upon the knowledge of the timer from above. By wrapping a 'for' loop which does nothing (inner portion of loop is simply an empty instruction) we are able to determine the overhead we incur for running a loop! We predict the loop overhead to be fairly minimal; after all, if it wasn't loops would not be so heavily used because they would be prohibitively expensive to do so.

When analyzing how a loop is broken into assembly, we note that there is a jump instruction incurred at the end of every loop iteration, minus the last one. Additionally, there is an increment instruction followed with a comparison at the beginning of every loop iteration. So we estimate that the loop overhead will be 3-5 cycles. The preceding table highlights the overheads we examined. The 6.5 cycles is indeed close to what we expected.

PROCEDURE CALL

Another very common section of code within a program is a function, or a procedure, call. These will vary greatly both in the number of arguments passed to the call as well as the type of arguments that are being passed in. To capture this relationship we measured 16 different procedure calls with varying parameter types and total parameters. We performed 8 procedure calls with varying number of parameters passed but all of floating point types. Similarly, we again varied the number of parameters passed in to 8 procedure calls, but this time with double precision floating point parameters being passed. To isolate the overhead from the procedure call itself the block of code executed is nothing. The procedure call simply returns doing no work.

When a procedure call is translated into assembly by the compiler it will consist of a jump instruction that will push its return address onto the stack. Additionally, any parameters that are being passed into the call will also get pushed onto the stack accordingly. Thus, we expect that as the number of parameters passed into the procedure call increased the total overhead will also increase.

We predict that for a procedure call with 0 parameters the overhead will simply consist of the jump to the procedure call, the store of the return address, the load back of that address, and finally the jump return to get back to the calling code. In our scenario, the return address should stay within the cash because it is a write through policy and because we are not doing anything useful within the procedure call, it should still be in the cache upon the jump return.

We thus predict the 0 parameter call to take about 5 cycles. Further, as you increase the number of parameters that are passed in this should result in an additional small overhead being added for each parameter. We estimate approximately 1-2 cycle for each additional parameter to be able to place the additional parameter in the calling stack. Finally, by using a double precision floating point type we are increasing the size/complexity of each parameter that we are passing in. This should require additional bits to be copied onto the calling stack, and as a result we would anticipate a slight increase in the overall overhead. We would also anticipate that even more complex data types, like structs, would incur an even steeper overhead cost. As shown below, our predictions line up very well with the overall overheads observed.

Task	Time (microseconds)	Time (cycles)	Standard Deviation
Procedure Call: 0 Parameters	.00169 / .0018	5.939 / 6.389	3.6 E-6 / 1.316 E-6
Procedure Call: 1 Parameters	.00186 / .0021	6.52 / 6.558	2.13 E-6 / 1.322 E-5
Procedure Call: 2 Parameters	.002 / .0022	7.324 / 7.337	2.68 E-6 / 1.422 E-5
Procedure Call: 3 Parameters	.0022 / .0024	7.619 / 7.615	4.14 E-5 / 1.642 E-5
Procedure Call: 4 Parameters	.0025 / .0025	8.797 / 8.502	4.38 E-6 / 1.793 E-5
Procedure Call: 5 Parameters	.00257 / .0028	9.006 / 8.764	2.96 E-6 / 1.728 E-5
Procedure Call: 6 Parameters	.0028 / .003	9.859 / 9.723	2.82 E-5 / 1.897 E-5
Procedure Call: 7 Parameters	.003 / .0032	10.57 / 10.402	3.15 E-5 / 1.419 E-5

* NOTE: The table is represented as <statistic 1> / <statistic 2>. This represents the overheads observed when using a Float type for statistic 1, and a Double for statistic 2.

We speculate this is due to the detailed breakdown we did in our prediction and observation of the underlying hardware/software implementations. One interesting trend that we see is the Double parameter type starts off being a larger overhead, but as the number of parameters increases this is actually reversed and the Double procedures end up incurring less of an overhead than their float counterparts. We speculate that this is due to our machine being a 64-bit architecture and consequently the 64 bit parameters are scheduled more efficiently than the 32 bit parameters.

We are confident in our analysis, but one additional thing that could be tested would varying the parameter types even further to understand how the complexity of the data type is effecting the overhead of the overall procedure call.

SYSTEM CALLS

Vital components of many programs, functions, and code in general will either directly or indirectly make use of a system call. To measure the overhead associated with this we will analyze 3 minimal system calls of time, getPid, and open. The first two are about as basic of system calls as you can get, and the last – open, is a bit more complicated as it must first resolve the file descriptor and could cause slow down due to buffers. We predict the system calls to themselves to resolve in short order, taking a couple of cycles to complete. Even though the open system call could have some possible dependencies that would need to be resolved, we expect these all to be able to complete under 20 cycles, with the time and getPid system calls resolving much faster, probably closer to 5-10 cycles due to the fact that they will get cached. Note that with system calls some operating systems will cache the results and as a direct result skew our overhead measurements. Thus, we show one such instruction, namely getPid, to illustrate the effects this caching has.

To measure the overhead associated with these system calls we wrap the system call with a timer to measure the amount of time to complete the system call. Within the actual test we only call one function call at a time, and then run each test 1,000,000 times to determine the average and standard deviations. If there are any obvious outliers to the data we discard them.

The predictions for time and getPid were spot on, however the prediciton of 20 cycles for the open was not even close the our observed average of 1,389 cycles. This must be because the open call is page faulting, causing the system call to invoke the fault handler which is then in turn going out to main memory to service the fault. The 1,389 agrees with this analysis. The following table shows our observed data.

Task	Time (microseconds)	Time (cycles)	Standard Deviation
System Call: time	.00318	11.146	1.42 E-5
System Call: getPid	.00344	12.055	6.19 E-6
System Call: open	.39713	1389.99	4.65 E-3

TASK CREATION TIME

For this experiment, we were asked to determine the overhead of creating (a) a process and (b) a kernel thread. To measure the overhead of creating a new process, we call the fork() function and measure the time consumed from the time it was called until the time it returned. The process creation was verified within our code, and the process does indeed get created. In order to get accurate measurements, we implemented a loop that forks 1000 processes, measured the overall time and then divided by 1000 to get the overhead of a single fork() call. The newly-spawned processes simply returned without doing anything.

In the case of thread creation overhead, we followed a similar approach, where in a loop we spawn 100K threads. Each thread is asked to execute a function that simply returns and kills the thread. Following the spawn-and-kill approach in both the process and thread creation overhead measurements, we ensure that we isolate the required measurement without introducing more overhead.

When creating a new process, the processor creates a copy of the parent's address space while in the case of spawning a new thread, the parent process's address space is shared. Assuming that the new thread will run on the same core as its parent, there is no need for memory transfer. Consequently, we estimate that a thread's creation will have significantly less overhead that creating a new process. And since memory operations take place in this process, we estimate the overhead difference to be in the order of hundreds of clock cycles.

As mentioned earlier in this section, the two benchmarks were ran 100 times each, with disabled frequency scaling (set to 3.5 GHz) and we also ensured that the benchmarks are only using one cpu. In this section, we report the average time in microseconds, the average time in clock cycles and the standard deviation of our measurements.

Task	Time (microseconds)	Time (cycles)	Standard Deviation
Process Creation	20.33	71146	3.171
Kernel Thread Creation	7.37	25805	0.164

Based on our measurements, it is obvious that spawning a new thread incurs significantly less overhead than creating a new process. However, we estimated the difference to be in the order of hundreds of cycles, while in fact it's an order of magnitude higher. Process creation incurs roughly three times the overhead of spawning a thread. We estimate that the reason for this overhead is the memory transfers and the required book-keeping by the operating system.

From our results, we can also observe very high standard deviation in our measurements regarding processes. We estimate that the memory hierarchy probably has an impact on this. Since memory transfers are introduced, whether or not the data is in the processor's cache can have significant impact on the overhead.

Context Switch Time

This experiment measures the overhead of a context switch between two processes and between two threads. This experiment proved to be more challenging than the rest, since we had to figure out some methodology in order to enforce a context switch in the processor.

To overcome this challenge, we used blocking pipes. To measure the overhead of a process's context switch, first we fork a new process and create two pipes for printer-process communication. Afterward, the parent process sends one byte of data through a pipe and then waits to read from the second pipe. Similarly, the child process first reads the data in the first pipe and then transmits one byte through the second pipe. In our benchmark, this process was repeated 100 times in a loop, in order to get more accurate results.

The instruction that reads from a pipe is blocking, meaning that a process will stall until data is available for reading. By blocking one process, the processor immediately switches context to the other process. As mentioned, we forced the program to only use one of the eight threads our cpu supports, essentially ensuring that the second process will not start executing in parallel in some distant thread/core.

To measure the context switch overhead of two kernel threads, we followed the same strategy with blocking pipes. Two child threads were spawned. Each thread was given its own function to execute. The two functions essentially implemented the same pipe-communication as before. Again, the benchmark was executed on a single thread with the use of the taskset instruction.

To provide an estimation on the context switching overhead, we need to consider what happens during a context switch. In the case of a process context switch, the processor needs to copy all the necessary registers (for example the program counter) in memory, in order to be able to resume operation later. However, when the context switch is between two threads of the same process, the processor does not need to “backup” anything, since the threads are sharing the process's address space. Based on the hardware's different approach to each “version” of context switching, we estimate that the memory transactions required will once more introduce higher overhead in the order of thousands of clock cycles for the process context switch compared to the thread's context switch. The following table presents the obtained measurements:

Context Switch	Time (microseconds)	Time (cycles)	Standard Deviation
Process	4.57	15985	0.714
Kernel Thread	2.4	8422	0.27

MEMORY

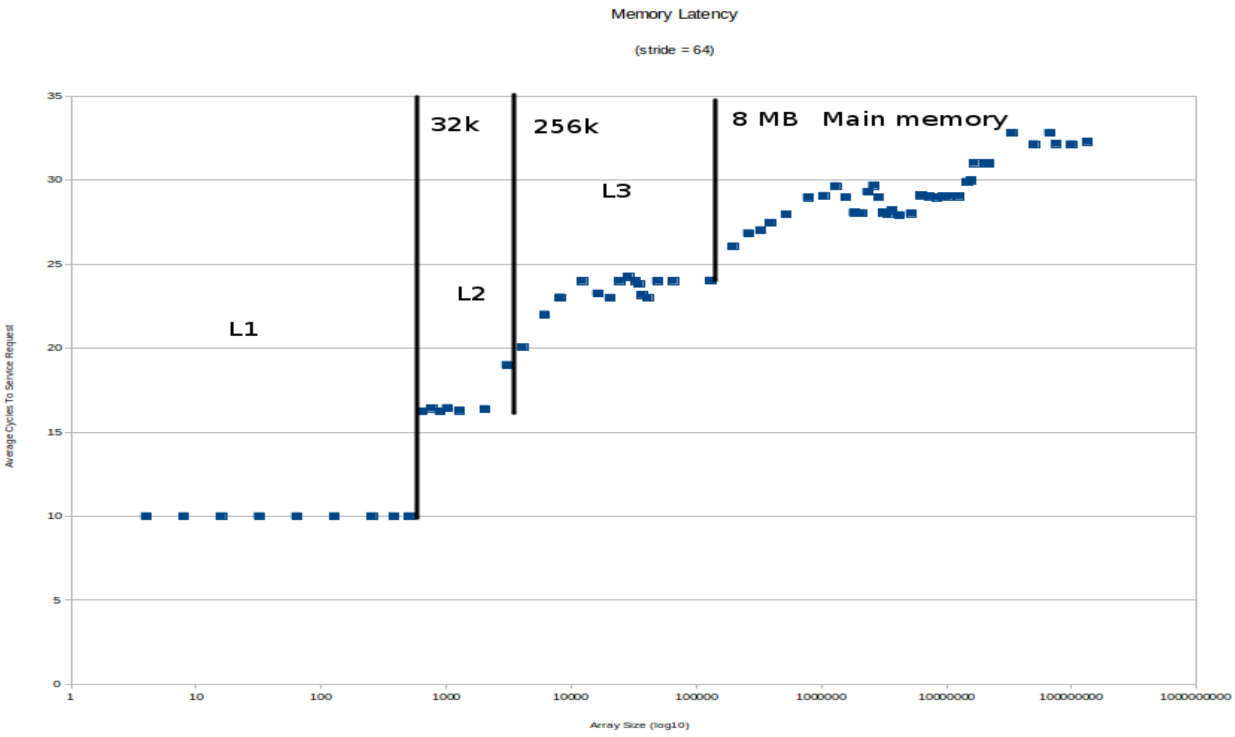
The memory hierarchy in a modern PC is broken into multiple levels of caches. There is the first level of cache, called the L1 cache, which is dedicated to a single core within the processor. This level is very small, but also very fast to access. There's a second level of L2 cache which is again private to a given core. The L2 cache is generally larger than the L1 cache, sacrificing speed for size. Finally, many multi-core chips have a third level of cache called the L3 which is much larger than either the L1 or L2 caches. The L3 cache, unlike the L1 or L2 caches, is shared across cores.

MEMORY LATENCY (L1, L2, L3, AND MAIN MEMORY)

Our goal is to measure the individual integer accesses to each of these levels of cache as well as to main memory. We want to do it in a way that is implementation independent. That is to say, we want to be able to determine these speeds without needing to know the specific size of the caches, nor how many levels of caching are present. To accomplish this goal we implemented a strategy discussed in the lmbench paper. When running the experiments targeted at measuring cache metrics we ran them within a loop of increasing power of 2 sizes. The performance results were then plotted, and the cache level performance metrics, along with main memory, are directly extracted from the graph. The benchmark has two parameters, the array's size and the array's stride. We will use an array stride equal to that of the cache lines within our processor, that is we set the stride equal to 64. This will later be tweaked and explained below.

In an attempt to prevent hardware prefetching we divided our pointer list into 4 sections, we then point the first fourth to the last fourth, the last fourth to the second fourth, the second fourth to the third fourth, and the third fourth back again to the first fourth (but incremented by one index position). Thus, we visit every position within the array, but do so in a pattern that should be confusing to the prefetcher.

The pointer list is then traversed multiple times with the average time per access reported. The basic concept here is that as the array sizes increase they can no longer fit into lower levels of cache. Ie: with an array size of 4, by following our pointer list of 0->3->2->1->END and repeatedly applying these steps, after the first iteration each of the 4 pointers will have already been brought into the first level cache. However, as the array size grows, 0->3->2->1->0+1->3+1->...->END, the entire pointer list will no longer fit in the first level of cache! However, it may still fit in the second level. As you can see in the graph below, there are indeed these inflection points shown from L1, L2, L3, and main memory where the array size has grown past the point of being able to fit within a level of memory hierarchy.

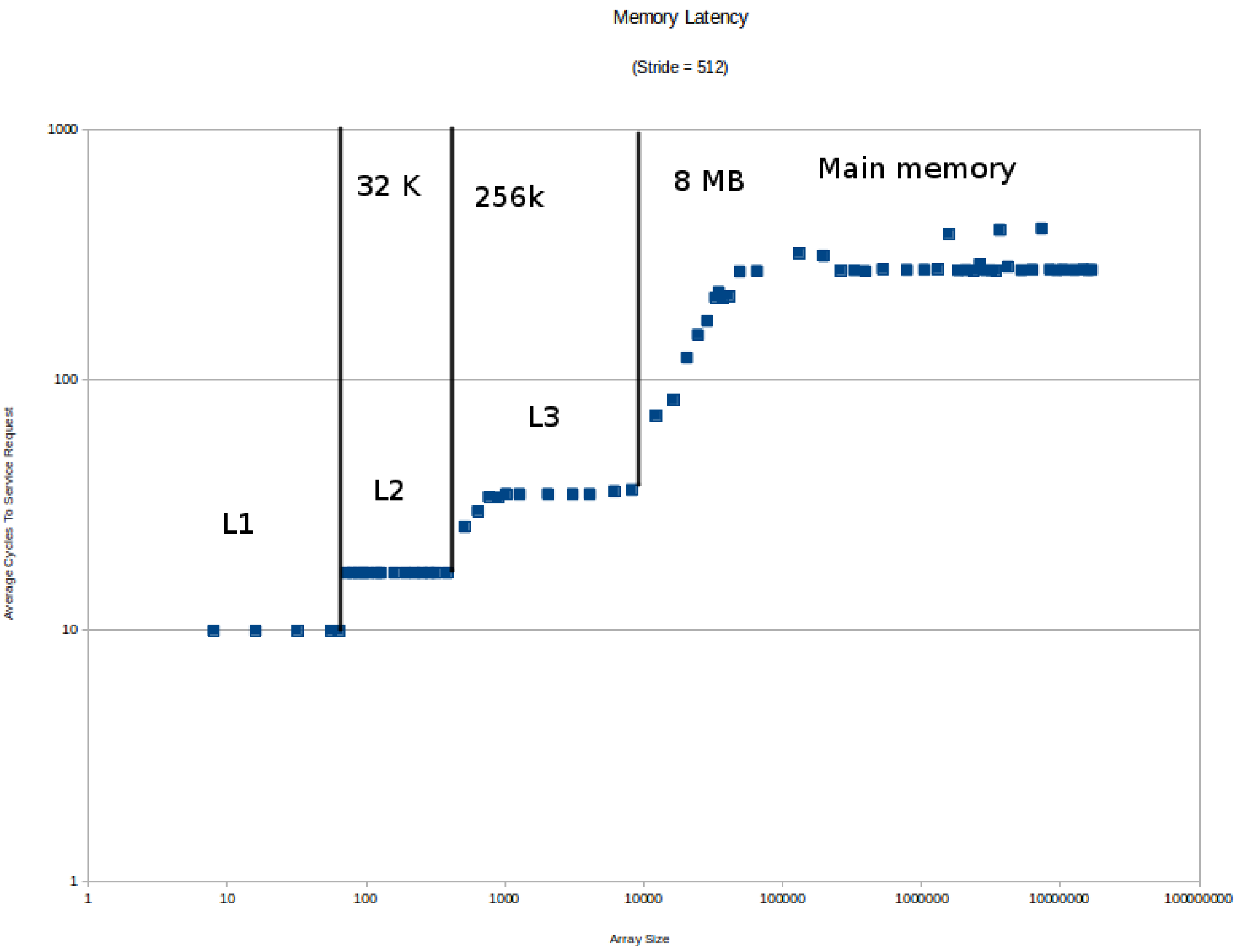


Consequently, the average access for these elements spikes. In an attempt to avoid skewed metrics, we first ran through the entire array one time to warm up the caches. We predict there to be spikes at the memory boundaries, or specifically 32KB, 256KB, and 8MB for the L1, L2, L3 caches respectively.

We predict that the L1 will hit with a low latency of only several cycles, so in the range of 3-5. Based on experience and a survey of other cache level latencies (www.anandtech.com), we predict the L2 to have a 10-15 cycle delay, with the L3 ranging from 25-50. Main memory should take around 150-200 cycles.

As the figure above shows, there are clear sections where the access patterns are falling within the caches. With the stride length equal to that of a cache line (or 64), the latency does not exceed 35 cycles! Even for those memory indexes that really should be hitting in main memory. This is because with the stride at 64, the hw prefecter is smart enough (Sandy Bridge has significantly increased the performance of the hw prefecther) to detect our access pattern, and is consequently prefetching the data needed. Thus, we don't actually go to main memory in most cases, but are still able to remain within the cache hierarchy! This is NOT the behavior that we desire for this test, so to overcome this we increase our stride length by a factor of 8 to now be 512.

By increasing the stride length to 512 we were able to avoid any benefit seen by the hw prefetcher. Indeed in the second figure we can see that the hw prefetcher no longer can effectively get the future memory references we will be using. This is shown directly in the graph below with the main memory access now reaching the 250 cycle threshold as expected. The L1, L2, and L3 cutoff points have been highlighted on the graph, but the results are in agreement with our predictions.



	L1	L2	L3	Main Memory
Stride = 64	10 cycles	16 cycles	25 cycles	30 cycles
Stride = 256	10 cycles	16 cycles	35 cycles	275 cycles

MEMORY BANDWIDTH (READ AND WRITE)

Another important metric that goes hand in hand with memory latency is the memory bandwidth. To test the bandwidths of the caches we created an array large enough to not fit within RAM but not so large as to cause it to be stored within virtual memory. Our cache hierarchy can support just over 8MB, we chose an array of 256MB to guarantee that our array was amply large to avoid cache effects. As described above, we avoided sequential reads and writes to prevent hw prefetching. Since we knew the size of the array we were going to read/write was 256MB, we wanted a stride that was large enough to overcome any prefetching in hw that might skew our results. We also unrolled the loop to increase our accuracy. The entries were summed in an attempt to prevent any compiler optimization from removing the instructions. Again, for accuracy we warmed up all caches before actually taking our measurements.

Based on the fact that our system support DDR3 1333 MHz RAM sticks, theoretically each stick has a maximum bandwidth equal to:

clock rate * data transferred per clock * bits transferred per clock / 8

Which when plugging in the specification numbers listed on our RAM module, we get a theoretical maximum of 10,664 MB/s. This is the absolute maximum that one stick of RAM of this type can handle, assuming a complete saturation of the bus at every cycle. This transfer rate is effectively doubled when in a dual-channel system, which ours is. So the maximum theoretical bandwidth is 21,328 MB/s. Based on the fact that the RAM chips will very rarely ever be fully saturated, even in a bandwidth test, we predict the observed bandwidth to be ~75% of the overall peak bandwidth, or that is to say we predict ~15-16GB/s. The write performance is generally slower (due to the physical nature of a write having to actually magnetize/change the physical state of some material) than read performance, and we estimate the performance to be roughly half that of the read.

The read bandwidth benchmark is very similar to that described within the lmbench paper. We created a large integer array, summed up a large series of values (this portion was unrolled). As noted in the lmbench paper, the addition will complete within 1 cycle, and the overall benchmark measurement will be dominated by the memory subsystem. The write bandwidth benchmark is very similar to the read, but instead of reading we are writing consecutive values. By writing/reading an array large enough, when repeating our benchmark we were able to avoid benefits that would have been observed if the array would have been able to fit within a cache level, as a result we were able to get results that are in line with our predicted values. The observed read performance is spot on with our prediction, the write performance we were a little too optimistic on, with the write performance measured being roughly 1/3 of the read performance. For both cases, the standard deviation was insignificant.

Read Bandwidth	Write Bandwidth
15005 MB/s	5642 MB/s
5.1 MB/s Standard Deviation	7.2 MB/s Standard Deviation

Servicing Page Faults

This experiment measures the time required to service a Page Fault. To enforce a page fault in a controlled manner, we used the mmap command, which creates a new mapping in the virtual address space of the calling process. Essentially, mmap creates the mapping for a new page and when the program attempts to access it for the first time, a page fault occurs and the page is loaded.

We verified that our program really introduces a page fault with the use of the /usr/bin/time tool which reports the number of page faults (among others) during another program's execution. We ensured that the number of page faults reported was increasing as expected when we were running toy examples of our code. We also ensured that we have a major page fault, meaning that the page will be loaded from the disk.

The page size in our system under test is 4KB. To enforce page faults, we created a file of size 40GB. With the use of mmap(), we mapped that file into pages. After mapping, the file can be accessed as a traditional array of characters. Accessing index 0 the first time produces a page fault, however every following access to the same page (up to index 4K-1) does not raise a page fault. Accessing index 4K however creates a page fault and the second page is loaded.

Our program maps the large file and then performs two accesses which in turn cause two page faults. The first access reads index 0 and the second reads a random index up to the limit of 40 GB. We measure the time for two page faults to be served and report the time. We executed this program one hundred thousand times to get the average value for servicing a page fault. Our results demonstrate that page fault servicing requires 15228 ns (0.015 ms) (~53300967 clock cycles).

NETWORK

When measuring the network's characteristics, we implemented benchmarks to measure both the loopback and remote interfaces for every aspect we were asked to measure. To write network related benchmarks, we used sockets to create client-server connections.

Round Trip Time

Loopback Interface:

To measure the network's round trip time for the loopback interface, we implemented a simple client-server benchmark. To keep things simple, we implemented both the client and the server in the same source file. With this approach, we don't need to run two programs on the same system and as such, our results are not subject to error because of context switching and other aspects. To write both the client and server codes in one file, we had to make the server waiting for connections asynchronously, so the program can continue into the client's code. This optimization does not alter the behavior of our benchmark, since we start measuring time long after the connection is achieved.

When a connection is established, the client sends a message to the server. Upon receiving the message, the server simply writes it back to the socket so the client can receive it back. We measure the time between the client's first send and after receiving the message back. To add more confidence to our measurement, we repeated the same procedure 1000 times, with each time measurement stored in an array. Then the array of measurements was used to compute the average time for the Round-Trip time on the loopback device. Since we are measuring the RTT, creation and initialization of the required buffers are not included in the measurements.

We also compared our benchmark's results with the results of the ping command. For a fair comparison, we set the packet size of our benchmark to 56 bytes in order to match ping's packets. According to ping's manual page, "The default [size] is 56, which translates into 64 ICMP data bytes when combined with the 8 bytes of ICMP header data".

Running ping on the loopback interface, we got an average of 23 µs. Based on this knowledge and on the fact that we are eliminating the ICMP overhead in our benchmark, we predict that RTT time will be about half the ping's RTT time.

The following table shows our benchmark's measurements for the round trip time in cycles and microseconds, as well as a comparison against ping's results.

Prediction (µs)	RTT (cycles)	RTT (µs)	STDV (cycles)	Ping (µs)
10	14191	4.05	526	23

<u>Remote Interface:</u>

To measure the RTT of remote interfaces, we implemented the same benchmark, however this time we created separate files for the server and client to allow us to run each one on its own machine. Again, the client connects, sends a message and received the message back. Like before, we set the message size to 56 Bytes for a fair comparison between our benchmark and ping.
To eliminate the risk of external factors altering our results, we run both the client and server programs on machines with identical specs (presented in the first chapter), both hardware-wise and software-wise. Like before, based on ping's performance and the elimination of ICMP protocol's overhead, we estimate the RTT time for remote interfaces to be in about half of that measured with ping.
The following table shows our benchmark's measurement for the round-trip time in cycles and in microseconds, and a comparison against ping results.

Prediction (μs)	RTT (cycles)	RTT (μs)	STDV (cycles)	Ping (μs)
100	1627965	132.9	118923	213

Peak Bandwidth

The second part of the network evaluation was to measure our system's peak bandwidth. To achieve this we implemented a client/server benchmark similar to the one we used before, with some modifications. Since we are using identical machines, it does not make a difference on which machine the server runs or the client.

First, to measure the peak bandwidth, all we have to do is to get the time needed for the server to perform a read call on a socket. The rest is simple math. Again, we have the client sending a message to the server. The server simply measures the time it needs to read the socket. We repeat this procedure 10K times for each experiment and report the average results.

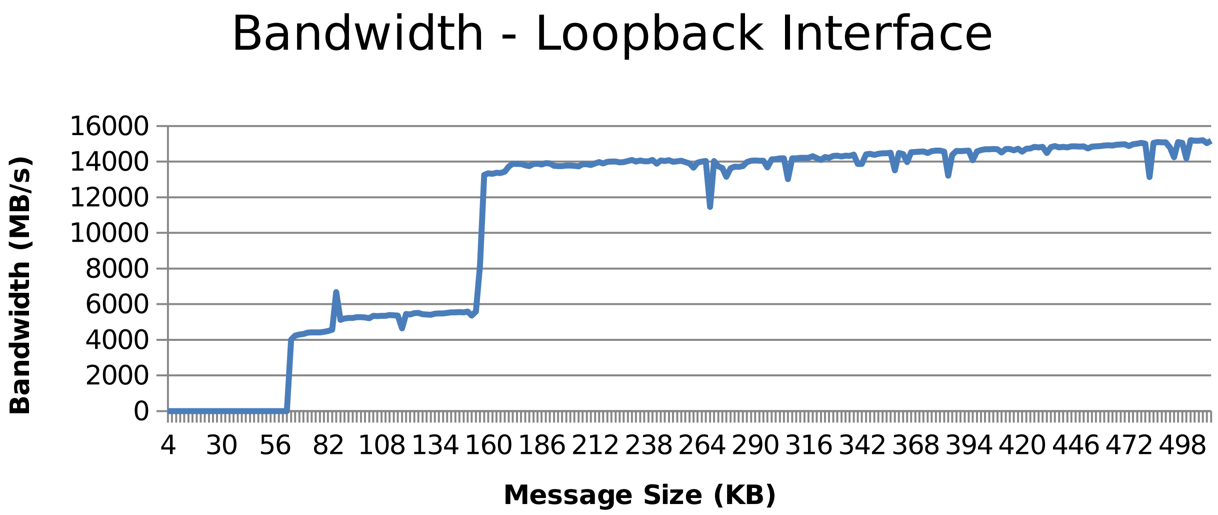
The size of data being sent should affect the measured bandwidth. Under-utilizing the available resources could lead to pessimistic measurements. To overcome this issue, we needed a methodology that allows us to saturate the connection to its limits. Our benchmarks increment the size of the messages in steps in order to get a better view of our system.

<u>Loopback Interface</u>

For the loopback interface, the system's NIC is bypassed and because of that we expect the limit to be our memory's bandwidth (measured in earlier section to be ~15GB/s). Again for simplicity, we implemented both the client and server in the same source file to prevent context switching from taking place and altering our results.

The following table presents the measurements and the figure visualizes the measured peak bandwidth versus the size of the message. We can see that indeed the results show the peak bandwidth to be very close to the memory's bandwidth, as expected.

Prediction (GB/s)	Peak Bandwidth (GB/s)	STDV (MB/s)
15	15.2	53.1

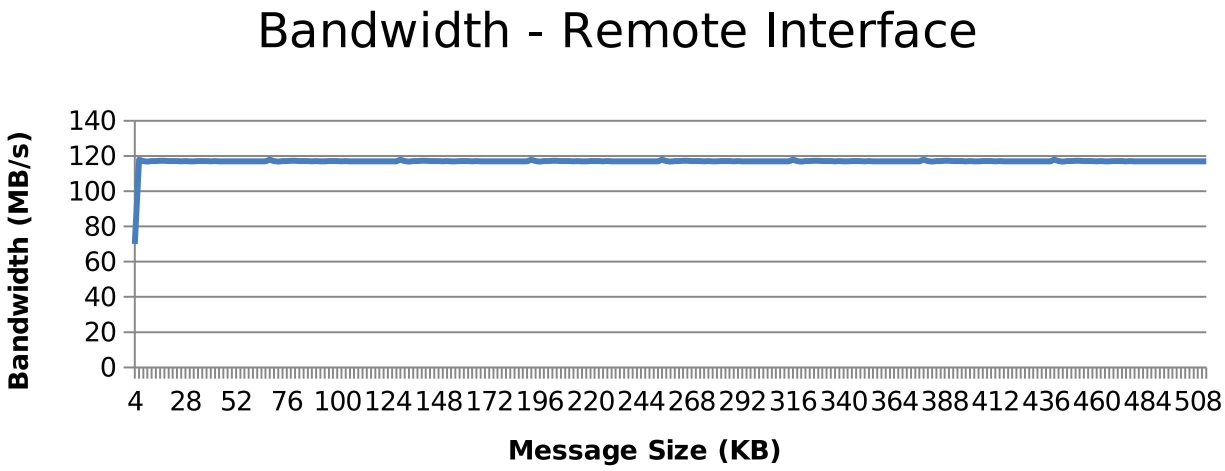


Remote Interface

For this case, the server and client are running on two identical machines, performing the exact same steps described earlier. The two machines are part of the same LAN, in which the routers and CAT cables support 1Gbps (128 MBps). This is much lower than the memory's bandwidth, and we expect this number to be the bottleneck of the communication.

We present our measurements in a similar way as the Loopback Interface part of this experiment.

Prediction (MB/s)	Peak Bandwidth (MB/s)	STDV (MB/s)
128	117.8	2.95



Connection Overhead

To measure the time required to setup and tear down a connection, we implemented very similar benchmarks as those used so far for our network analysis. This time, we measure the time that the server was waiting for a connection (Setup time) and the time required to close the connection socket (Tear-down time).

Loopback Interface

We used the client/server benchmark described in the previous sections, with both the server and client in the same source file. For the setup time we measure the time difference between the non-blocking connect instruction (client) and right after the accept instruction (Server). To ensure that the server is always waiting for a connection and it doesn't skew our measurements, we repeated the connection procedure 10K times and ignored the first hundred measurements as a warm-up period.

The tear down time is easy to measure, since it only involves closing the connection socket. We are measuring the time the server needs for one close (socket) instruction.

We expect the connection setup time to be in the range of 20-30 µs. We base this estimate on the fact that we measured 4 µs of RTT time earlier and our expectation that connection setup is a much longer procedure. We also expect the tear down time to be much smaller than the setup time, in the range of 10-15 µs.

The table below presents our results.

	Prediction(µs)	Overhead (cycles)	Overhead (µs)	STDV (cycles)
Setup	20-30	33284	9.5	335
Tear down	10-15	26997	7.71	224

Remote Interface

With the server and client benchmarks running on two identical machines, we measured the connection setup time as the time before and after the blocking connect instruction of the client. Like in the loopback interface, we included a warm-up phase, to ensure that the server does not alter our results. In a loop, we create a connection from scratch and close it right before the end of the loop. This allows us to measure both the setup and tear down time in one run.

The setup/teardown procedure was repeated in a loop for 10K times for more accurate measurements. In our results, we present the average values. The table below presents our results.

We predict the setup time to be in the range of 90-100 µs, simply because the measured RTT time for the remote interface was roughly 10 times larger than the RTT time for the loopback interface. Extrapolating this information and based on our earlier measurement, our expectation is justified.

However, tear down time is simply closing a socket, and as such we expect about the same overhead as the loopback interface.

	Prediction(µs)	Overhead (cycles)	Overhead (µs)	STDV (cycles)
Setup	90-100	2666817	762	20463
Tear down	7	23423	6.7	198

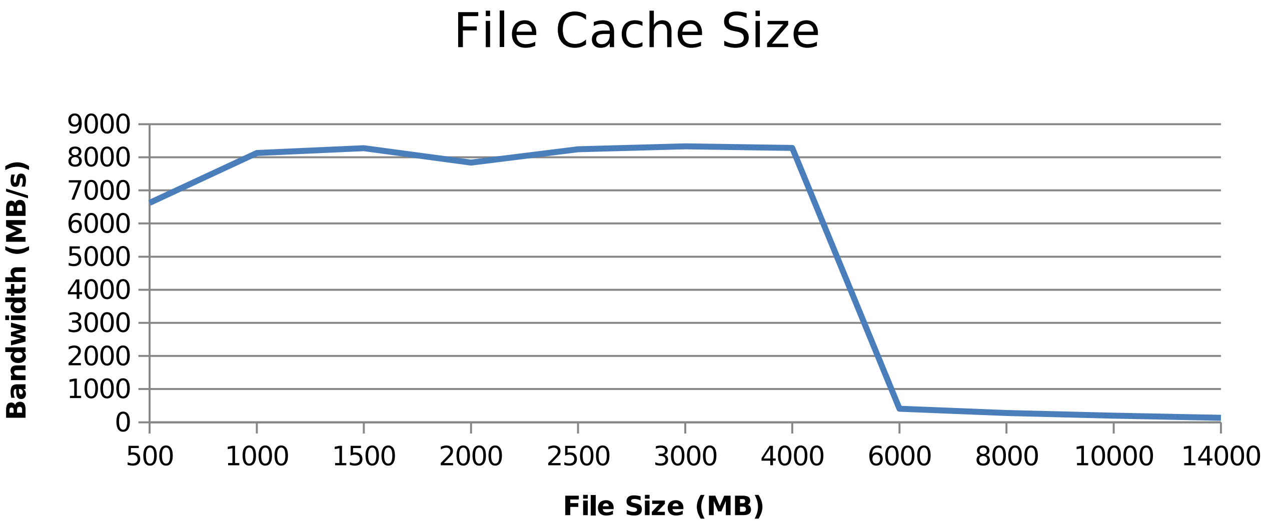
FILE SYSTEM

File Cache Size

File cache is used to temporarily store files accessed from the disk in order to service future requests on the same files faster. Our goal was to measure the size of this file cache. To achieve this goal, we implemented a benchmark that repeatedly reads a file (in a loop). We expect subsequent reads to be serviced from the file cache, unless the file size is larger than the amount the cache can store.

We created temporary files of sizes ranging from 0.5 GB to 14 GB (in 0.5 GB increments), which we fed to the benchmark and measured the bandwidth obtained from reading the files. The temporary files were omitted from the delivered files of this project due to their large size.

Our system has 32 GB of main memory and as such, we expect the file cache size to be in the order of Gigabytes. This is the reason we had to create so large input files. The following figure plots the measured bandwidth against the input file size.



As we can see from the plot above, files with size less than 4GB measured a very high bandwidth (~8GB/s), while the rest of the files provided an extremely low bandwidth. This suggests that the size of the file cache is equal to 4GB.

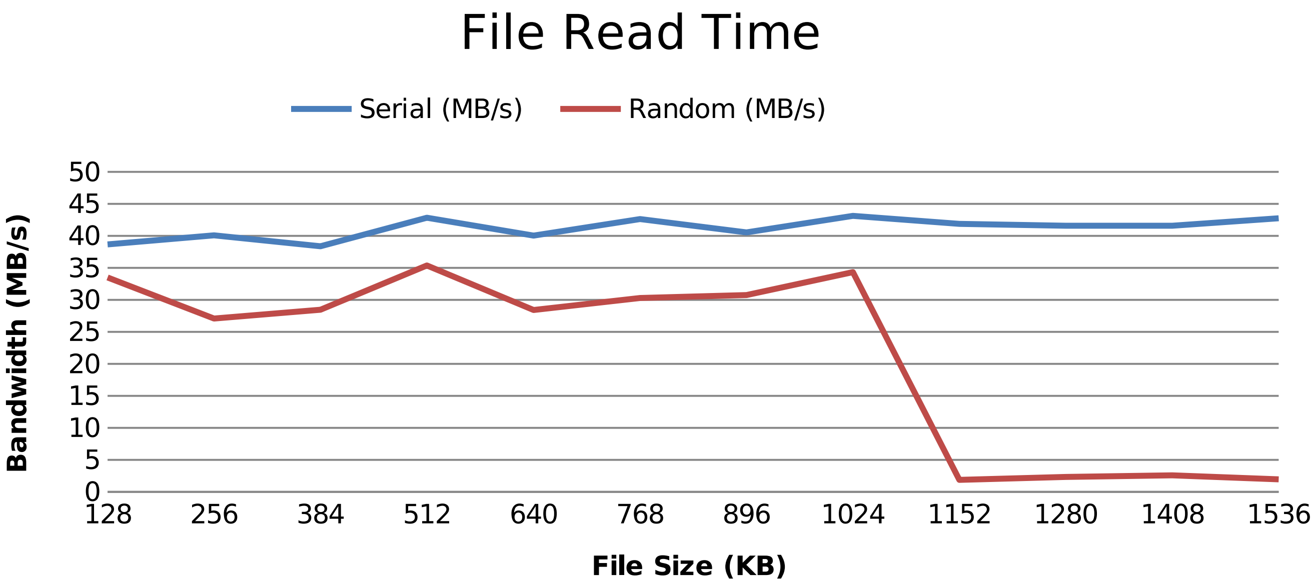
File Read Time

To measure the bandwidth of our hard drive, we had to disable file caching. We achieved that with the use of the O_DIRECT flag when opening the file. This experiment is very similar to the previous one. A file is opened and read multiple times. The average bandwidth is reported in our figure below.

We should note that since we are not utilizing the file cache, we expect reading times to be very large. To make our benchmark more efficient, we are now working on much smaller file sizes (128KB - 1.5MB) and since the file size is so small, we edited our benchmark to generate the file of the desired size on-the-fly.

One of our goals was to compare the impact of sequential and random file access. For the former, after a file is opened with the O_DIRECT flag, we sequentially read blocks of 4KB. To create random access patterns, we use a random number generator to offset the next access to a random page.

As seen on the following figure, for the case of sequential access, we observe that the measured bandwidth remains roughly the same. Since we are accessing the file in 4KB segments, the file size cannot affect the performance. However, this is not the case for random accesses. The reason is that as the file size grows, a random access pattern will increase the overall seek time required. We can observe how the bandwidth degrades for larger file sizes and random accesses, while for small files the effect is small.



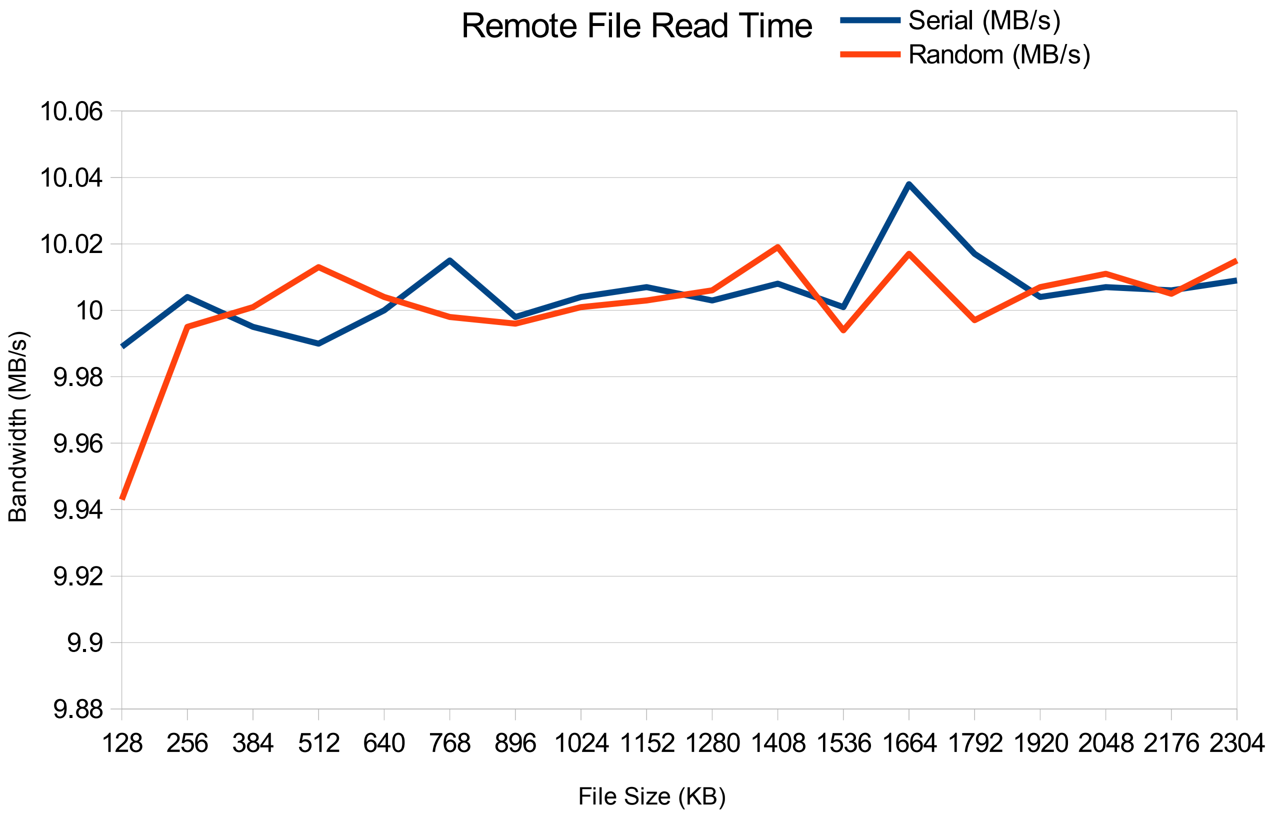
Even though our experiment reports a constant bandwidth for sequential accesses, in theory it is possible to observe different results. If a file is not stored sequentially due to disk fragmentation, it can have the same degradation as the random access. When moving to even larger sizes, the possibility for fragmentation increases.

Remote File Read Time

To measure the remote file read time, we used a NFS server and performed the same experiment as earlier for both sequential accesses and random. This time, the file sizes we used ranged from 128 KB to 3 MB. Again, file cache was disabled. The following figure demonstrates the obtained bandwidth.

We observe that the bandwidth in this case is limited by the network. The machines of this server are connected with a 100 Mbps Ethernet cable and as such it can provide a maximum bandwidth of 12.5 MBps. Our benchmark's results are near the connection's limit. We estimate that the reason the bandwidth is not very close to 12.5 MB/s is possible contention by other jobs running on the server.

In the following figure, note that the Y-axis is zoomed in by a lot. The two lines are roughly identical.

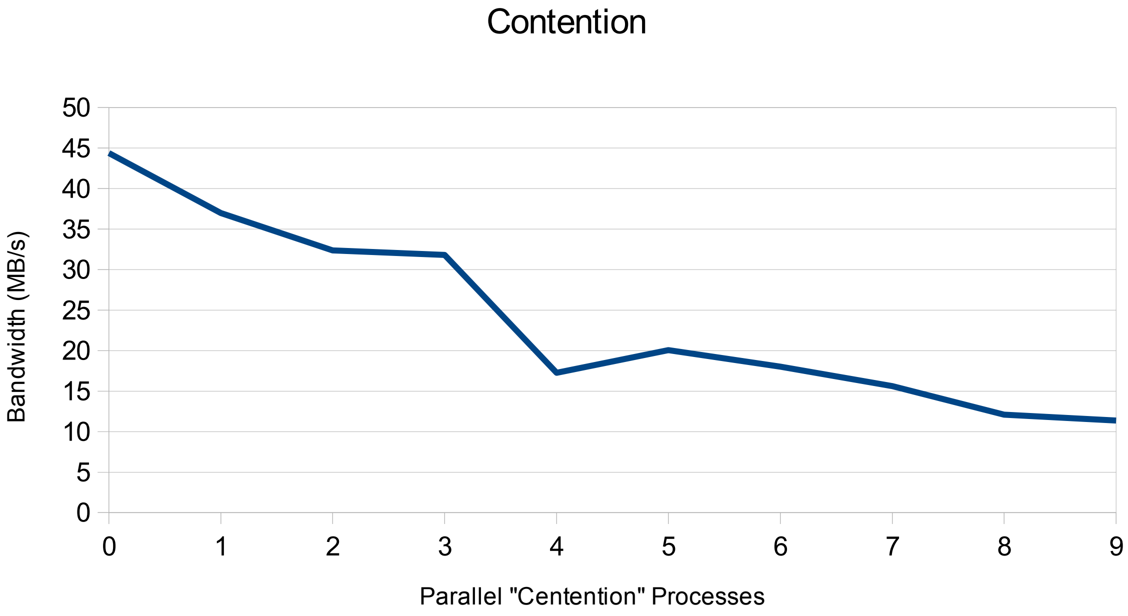


Contention

As a final step in our evaluation, we had to measure the system's reaction under contention. Contention occurs when multiple processes compete one another for the same resource. To obtain this metric, once again we disabled the file cache and measured the time required to read a 4KB block off the disk, while multiple processes are accessing the disk simultaneously.

In our benchmark, each "contention" process creates a 64KB file and reads it indefinitely in an infinite loop. This way we ensure that each process will constantly be accessing the disk. The main process is responsible for reporting the results. In parallel with the "contention" processes, the main process accesses a 4KB block off the disk for a thousand times and reports the average time required.

In this experiment, we varied the number of "contention" threads from 0 to 9. The following figure demonstrates the results. As expected, performance degrades as more processes access the disk simultaneously.



Conclusions

The goal of this project was to characterize our system, based on several metrics. We feel like the project's goals were met successfully.

Working on this project we learned a lot of new things about how a system is constructed and how it works. Also, collaborating efficiently with a team, as well as learning the process of creating a benchmark to measure something very specific, will be a valuable asset in our future research.