CSE 221 - PROJECT REPORT

Submitted by,
Prabu Dhakshinamurthy
Apurva Kumar

Introduction:

The goal of the project is to predict the performance of various system components, measure the performance and the compare the outcome with the prediction.

Language used: C
Compiler version: gcc 4.5.2 compiler
Optimization flags used: Default optimization level which is 'Oo'
Who performed what:

Time spent on the project: 150-200 hrs

Machine Description:

Component	Description
Processor	intel i5-2410M, 2,3GHz;
Cache	L1 - 32KB instruction, 32KB data; L2 - 256KB; L3 - 3MB
Memory	4GB
Disk	320GB, 7200 RPM, Controller cache - 16MB Avg Seek time: 12ms Seek time track-track: 1ms
Network	1000Mb/sec
I/O Bus	66MHz
Operating System	Ubuntu 11.04
Kernel	GNU/Linux, 2.6 kernal

Experiments

Cycle counter:

For all the experiments, we use the rdtsc instruction to measure the cycles elapsed for an operation. We initially calculated the cycle counter overhead by simply starting and stopping the counter and measuring the difference. We repeated this a number of times, took the average and found out the cycle counter overhead. Since the overhead thus found varies for each run, we calculated this overhead separately at the beginning of (in the same source file) each of our experiments.

For all the experiments, we have used the *'taskset'* command inorder to pin the process to a particular core. Also, we disables the speed step.

Measurement Overhead

Counter Overhead:

Methodology:

We use the rdtsc machine instruction to measure the clock cycle count. We have inline assembly code that executes the rdtsc instruction and records the CPU clock cycle in a long integer. To measure the overhead of our cycle counter, we get the clock cycle count twice, one after the other and measure the difference between the two. This will give the cycles actually spent in the cycle counter.

We inlined the cycle counter procedure in order to avoid the extra cost associated with procedure call.

Number of samples is configurable in our program and we repeated the experiment with varying sample count and settled on a result and exhibited minimum standard deviation.

Prediction:
Base hardware performance:

rdtsc instruction:

From the intel site and other online sources, we came to know that the rdtsc instruction takes around 100-200 clock cycles to report the timestamp.

Left shift operation:

We do a left shift operation to join the high and low 32bits of the timestamp returned in the registers.This will take 1-2 cyles.

Software overhead:

There is no software overhead associated with the measurement.

So we predict that our counter will take somewhere between **100 to 200 clock cycles**.

Results:

All units are in #clock cycles

	Base hardware	Software overhead	Prediction	Experiment result	Std. Deviation
Counter overhead	100-200	0	100-200	163	7

Discussion of Results:

Source: We referred to various online intel resources to predict the rdtsc overhead time

Comparison: Our results are very close to the predicted performance

Evaluation of success: Since our results are in line with the expected performance, we think our measurement is pretty accurate.

Units: All units reported in the results section are the clock cycle count

---

Loop overhead:

Methodology:

To measure the loop overhead, we record the timestamp before beginning the loop and then record the timestamp immediately after completing the loop; The different between these two will give the time it takes to run a loop of the given number of iterations. We deducted the overhead incurred by the cycle counter from the results to get more accurate results.

In our program, the number of loop iterations and the number of samples are configurable and we repeated the experiment by varying these two parameters and finalized the result which had minimum standard deviation. We did the experiment with ‘for loop’.

Pseudo code:

```
for ( #samples ) {
        start = count()

        for( #iterations ) {
                // Do nothing
        }

        end = count()
        result[currentSample] = end-start-counterOverhead
}

getMeanStdDev(result);
```

Prediction:

Base hardware performance:

For each iteration of the loop, the following operations need to be done:

1. Update the loop iteration variable: 3 cycles
2. Termination condition checking: 2 cycles
3. Jump to the beginning of loop body/Exit the loop: 1 cycle

Software Overhead:

There is no software overhead

So for each iteration of the loop, it will take around 6 clock cycles.

Result:

All units are in #clock cycles

	Base hardware	Software overhead	Prediction (1000 iterations)	Experiment Result (1000 iterations)	Standard Deviation (1000 iterations)	Experiment Result (1 iteration)
Loop overhead	6	0	6000	20208	47	20

Discussion of Results:

Source: Manufacturer manual to find out the #clock cycles to execute an instruction

Comparison: We see a difference of around 14 cycles for each iteration of the loop between the prediction and the actual result. We feel that either our prediction is not accurate or the other system processes that are running in the system affects our measurement because in that case, our measurement includes the cycles spent while other processes are executing in the system.

Evaluation: Though our results are different from the prediction, since the standard deviation that we got is very very less, we feel that our results consistent during all the samples and close to the actual performance

Units: All units are in number of clock cycles.

Procedure Call Overhead

Methodology:

We created eight different procedures whose parameters vary from zero to seven integers. We then called each procedure in a loop for N (#samples) iterations and calculated the average time to make one procedure call. The procedure body is empty and the procedures start executing and the control returns to the caller immediately.

Since we call the procedure in a loop, we also calculate the loop overhead in advance (like we calculate counter overhead) and then subtract it from the results to discount the cost associated with using loops.

Pseudo code:

```
46 void procedureTwoArg(int arg1, int arg2) {
47 }
…
...
156     start = count();
157     for (index = 0; index < samplecount; index++) {
158         //Two arg procedure call
159         procedureTwoArg(arg1, arg2);
160     }
161     end = count();

130     meanArray[2] = (end-start-mean_counteroverhead - mean_loop)/samplecount;
```

Prediction:

Base hardware performance:

During a procedure call the following operations take place:

1. Create a new stack frame - 1 cycles
2. Allocate space and copy the parameters to the stack frame - 3 cycle * no. of parameters
3. Copy the return address to the new stack frame - 1 cycle
4. Update the stack pointer and instruction pointer - 1 cycle
5. Remove the stack frame - 1 cycle

Total: 4 + (3*#parameter)

Software overhead:

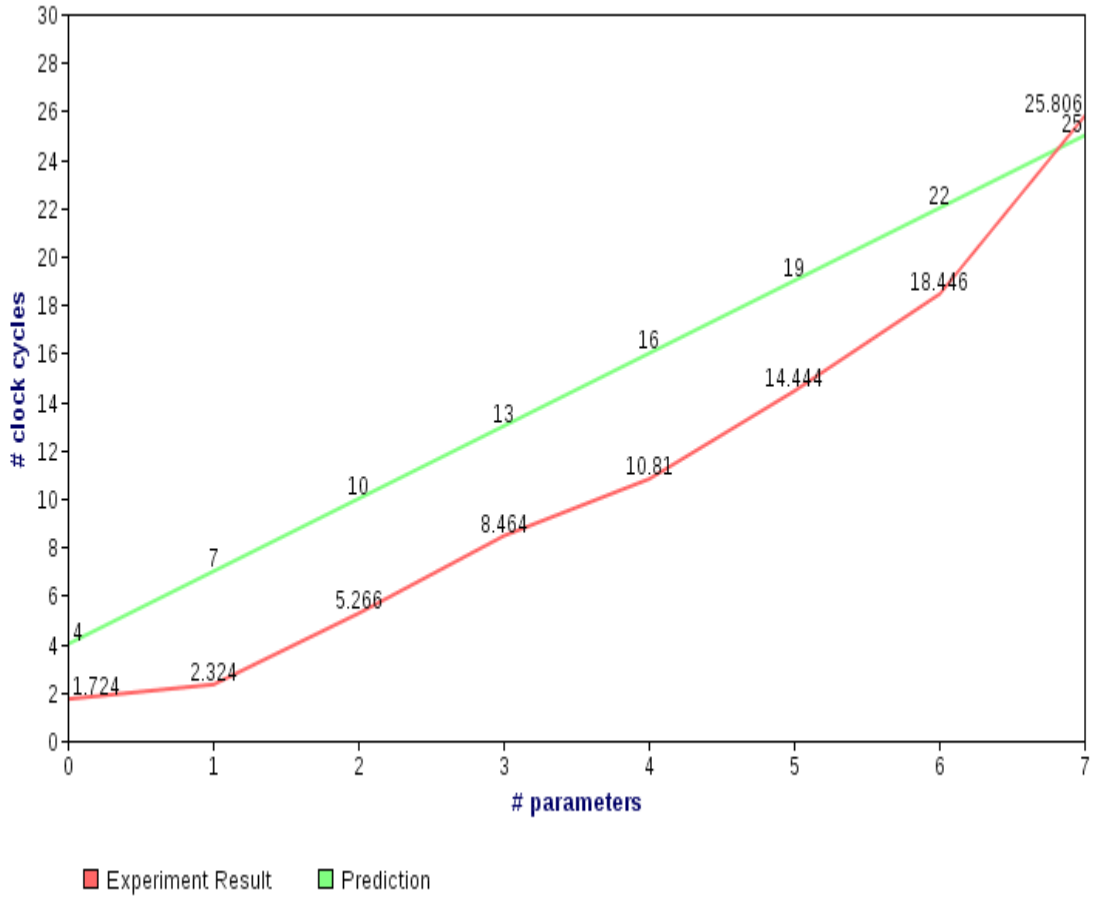
Software overhead is zero.

So we predict that it will take around 7 clock cycles for a procedure call with zero parameters and the number of clock cycles will increase by three for each additional parameter.

Results:

All units are in #clock cycles

# parameters	0	1	2	3	4	5	6	7
Predicted call time	4	7	10	13	16	19	22	25
Measured call time	1.724	2.324	5.266	8.464	10.810	14.444	18.446	25.8



Discussion of Results:

Source: Manufacturer manual to find out the #clock cycles to execute an instruction

Comparison: Our measurements show that, the time measured is slightly less than the predicted time when the number of parameters to the procedure call is lesser. But, as the number of parameters increase, our measurements slowly catch up with the predicted results. One possible explanation could be pipelining of instruction execution. Our prediction was based on the assumption that the instructions will be executed one after the other. But in reality, the instructions may be executed in parallel through pipelining mechanism and thus, because of the parallelism, our measurements take less time.

Evaluation: The obvious expectation is that, as the number of parameters to the procedure call increase, the time it takes to make the procedure call should also increase. In this sense, our results meet the expectation and we feels its a good sign that our results are correct. May be the numbers reported for each call is different from the actual performance but the time it takes for each procedure is relative to the number of arguments.

Graph: As seen in the graph, time for a procedure call increases linearly as the number of parameters increase.

Units: All units are in number of clock cycles.

For each additional parameter, it takes approximately around three clock cycles more.

System call overhead

Prediction:

We are planning to measure the overhead in calling the ‘dup’ system call to duplicate the standard output descriptor.

Base hardware performance:

Time to execute the dup system call: 700-1000 cycles

Software overhead:

Time to switch between user mode and kernel mode to execute the system call and then return back: 6-7 cycles

So as per our prediction, it **will take around 700-1000 cycles** to successfully execute the dup system call.

Methodology:

We measured the time to execute the dup system call by calling the system call repetitively in a loop, each time measuring the elapsed time; We repeated this experiment many times, each time varying the sample count and recorded the time where the standard deviation was less.

Pseudo code:

```
44    for (index = 0; index < samplecount; index++) {
45        start = count();
46        dup(1);
47        end = count();
48        syscallOverheadArray[index] = end-start-mean_counteroverhead;
49    }
```

Results:

All units are in #clock cycles

	Base hardware performance	Software overhead	Prediction	Measured performance	Mean deviation
dup system call	700-1000	6-7	700-1000	817	46

Discussion of Results:

Source: Referred to online discussion forums to find out the average time it takes for system calls and the time to switch between user and kernel mode.

Comparison: Our measured performance and the predicted performance are very close.

Evaluation: Since our measured performance and the predicted performance are very close, we feel that our experiment result is accurate.

Units: All units are in #clock cycles

The cost of a system call is very very high comparing to procedure call - system call nearly takes eighty times more time than a simple procedure call. This is because, the procedure call happens within the same address space whereas, a system call will lead to a protection domain crossing which is an expensive operation.

Task creation times

Prediction:

Hardware performance:

Processor is capable of executing every instruction in one clock cycles and the additional overhead while creating a new process is, to switch from user mode to kernel mode and then return back to user mode.

Switch from user mode to kernel mode and the return back: 6-7 cycles

Process creation:

Creating a new process is generally an expensive operation. When a new process is created, the execution environment for the new process needs to be setup and it involves the following steps (and a lot more).

- assigning a new process id
- allocating memory
- initializing stack
- initializing various register values
- copying various descriptors from the parent process
- initialize various resource monitors to track the new process's cpu utilization, execution time etc

So all these operations will take few hundreds of microseconds of CPU time.

Thread creation:

Comparing to process creation, thread creation is a less expensive task. Thread creation involves the following minimal steps:

- create a new stack
- create thread local space

New thread created exists within the address space of the parent process that created it and shares the global variables and the code section. We expect that, thread creation will take few tens of microseconds.

Methodology:

Process creation:

We use the fork system call to create a new process. We record the time just before calling the fork system call (start time) and to make the scheduler immediately switch execution to the child process, we call the wait system call in the parent process. Once the child process starts executing, we immediately record the time in the child process and this time will mark the end time. The difference between the end and start time will give the time elapsed to successfully create and start to execute a child process.

To communicate the end time that is recorded in the child process, we write to a pipe that is shared between the parent and the child process.

There is a little bit of overhead involved because of the 'wait' system call that we invoke to force the scheduler schedule the new process but, comparing to the process creation cost, this overhead will be very very minimal and we can safely neglect the cost associated.

Pseudo code:

```
54    // MEASURE THE PROCESS CREATION TIME
55    for ( index = 0; index < samplecount; index++) {
56        // start the counter before forking - this will account the process creation time
57        start = count();
58        if ((processId = fork()) != 0) {
59            exitId = wait();
60            read(readwritedescp[0], (void*)&end, sizeof(end));
61        }
62        else {
63            // as soon as new process starts executing, stop the timer
64            end = count();
65            write(readwritedescp[1], (void*)&end, sizeof(end));
66            exit(1);
67        }
68        processOverheadArray[index] = end-start-mean_counteroverhead;
69    }
```

Kernel Thread creation time:

We follow the same approach as above to measure the kernel thread creation time - instead of fork system call, now we use ‘pthread_create’ call to create a new thread. Also, to schedule the newly created thread, we execute, pthread_join from the main thread and this will make the scheduler schedule the new thread immediately.

New thread, once it starts executing, it records the time and this will mark the end time. Since, global variables are shared between the threads in the same address space, we communicate this end time to the main thread by means of shared variable.

There is a slight overhead involved in invoking the ‘pthread_join’ but, this overhead will be very minimal comparing to thread creation cost. So this can be safely neglected.

Pseudo code:

```
30 void *threadFunc( void *ptr ) {
31    end = count();
32    pthread_exit(NULL);
33    //return;
34 }

57    // MEASURE THE THREAD CREATION TIME
58    for ( index = 0; index < samplecount; index++) {
59        start = count();
60        threadId = pthread_create( &aThread, NULL, threadFunc, NULL);
61        pthread_join(aThread, NULL);
62        threadOverheadArray[index] = end-start-mean_counteroverhead;
63    }
```

Results:

All units are in micro seconds

	Base hardware performance	Software overhead	Predicted performance	Measured performance	Standard deviation percentage
Process creation	6-7	400	406	148	4.7%

Kernel thread creation	6-7	30 - 40	36-46	28	17.8%

Source: Referred to text books to know the steps involved in creating a new process and we referred to sources on internet to find out the average time for each step.
Comparison: Our results are slightly deviated from the the predicted performance. We think we overestimated the time it takes to create a new process.
Evaluation: It is expected that process creation should be expensive than thread creation and it is evident from our results. So, we believe that our results are good and accurate.

Creating a thread is nearly five times faster than creating a new process.

Context Switch Times

Prediction:

Base hardware performance:
Processor is capable of executing every instruction in one clock cycles and the additional overhead during context switch is to switch from user mode to kernel mode and run the scheduler. Also, processor need to handle the situation when a process/thread is blocked and notify the kernel to schedule another entity.

Switch from user mode to kernel mode and the return back: 6-7 cycles
Notify kernel of a blocked process: 2-3 cycles

Process switch time:
Switching from one process to another process involves the following steps:

- Save the execution context (Process control block) of the currently executing process
- Loading the execution context of the second process and start executing the process

We predict that this will take few tens of micro seconds.

Kernel thread switch time:
Switching from one kernel thread to another kernel thread involves the following steps:

- Save the executing context of the current thread: the execution context of a thread is much lesser compared to a process because, all threads created inside a process live in the same address space and they share lots of data and code with other threads. So only thread specific data and stack need to be saved.

- Load the execution context of the second thread and start executing it. Again, the context that need to be loaded is much lesser.

We predict that this operation will be much quicker than the process context switch and should take few micro seconds.

Methodology:

Process switch time:
We first created a new process by using fork system call. Once the new process is started, we record the time in the parent process (start time) and make the parent process wait for the the child process. This ‘wait’ call make the scheduler to do a context switch and execute the child process. As soon as the child starts executing, as a first statement in the child body, we record the time and this time marks the end time. After that the child process will write this end time to a pipe that is shared with the parent and exits. This the difference between the start time recorded in the parent process and the end time recorded in the child process will give the context switch time.

Comparing to the context switch time, the overhead associated with the wait system call us negligible and it can be discounted.

Pseudo code:

```
77    // MEASURE THE CONTEXT SWITCH TIME
78    for ( index = 0; index < samplecount; index++) {
79        if ((processId = fork()) != 0) {
80            // as opposed to process creation time, start the timer just before doing a context switch
81            start = count();
82            exitId = wait();
83            read(readwritedescp[0], (void*)&end, sizeof(end));
84        }
85        else {
86            // stop the counter as soon as the child starts executing
87            end = count();
88            write(readwritedescp[1], (void*)&end, sizeof(end));
89            exit(1);
90        }
91
92        processswitchOverheadArray[index] = end-start-mean_counteroverhead;
93    }
```

Thread switch time:

Same as the above method except for the changes below:

1. We create a new thread by ‘pthread_create’
2. We force a context switch by executing ‘pthread_join’
3. Communication between threads happens by means of shared global variable

Pseudo code:

```
71    // MEASURE THE THREAD CONTEXT SWITCH TIME
72    for ( index = 0; index < samplecount; index++) {
73        threadId = pthread_create( &aThread, NULL, threadFunc, NULL);
74        start = count();
75        pthread_join(aThread, NULL);
76        threadswitchOverheadArray[index] = end-start-mean_counteroverhead;
77    }
```

Results:

All units are in microseconds

	Base hardware performance	Software overhead	Prediction to do a single context switch	Measured time to do context switch	Standard deviation
Process switch time	8-10	20-40	28-50	27	7.4%
Thread switch time	8-10	2-5	10-15	4.08	19%

Discussion:
Source: Referred to online sources to determine the hardware performance and we calculatd the software overhead based on our intuition.
Comparison: Our results are very close to the predicted performance for process switch and its slightly less for thread switch. We think that the results are accurate but our prediction for thread switch time somehow went wrong.
Evaluation: The major expectation with this experiment was to compare process context switch time and thread process switch time. As expected, the process switch time is expensive and takes nearly seven times more time than thread context switch time.

Thread switch time is nearly seven times faster than process switch time.

Measure the latency to access an integer from L1 cache, L2 cache and RAM

Machine description:

L1 - 32KB
L2 - 256KB
L3 - 3MB
FSB - 1333MHz
Word size - 4 bytes

Prediction:
(All access times are to access a single integer [4 bytes])

L1 access time:

base hardware: 2 cycle
software overhead: 0
prediction: 2 cycle

L2 access time:

base hardware: L2 access time = 4 cycles
harware overhead: L1 miss penalty = 2
Software overhead: 0
prediction: 6 cycles

L3 access time:

base hardware: L3 access time =12 cycles
harware overhead: L1 miss penalty + L2 miss penalty = 2+4 = 6
Software overhead: 0
prediction: 18 cycles

Main memory access time:

base hardware: Main memory access time = 70 cycles
hardware overhead: L1 miss penalty + L2 miss penalty + L3 miss penalty = 2+4+12 = 18 cycles
Software overhead: 0
prediction: 88 cycles

Experiment: Measure the latency to access an integer from L1 cache, L2 cache, L3 cache and RAM.

Methodology:

High level idea is to allocate certain amount of memory in the heap and then access the memory in a random fashion by defining stride size.

1. Create an array of integer pointers of length L (and it will have size S = L * size(int*))

```
int **a = NULL;
a = (int**)malloc(startsize);
```

2. Fill the array such that, each element will have the reference to some other element in the array. Also inorder to avoid prefetching, we used stride of size > 3MB (size of L3 cache is 3MB - we wanted to keep the stride size more

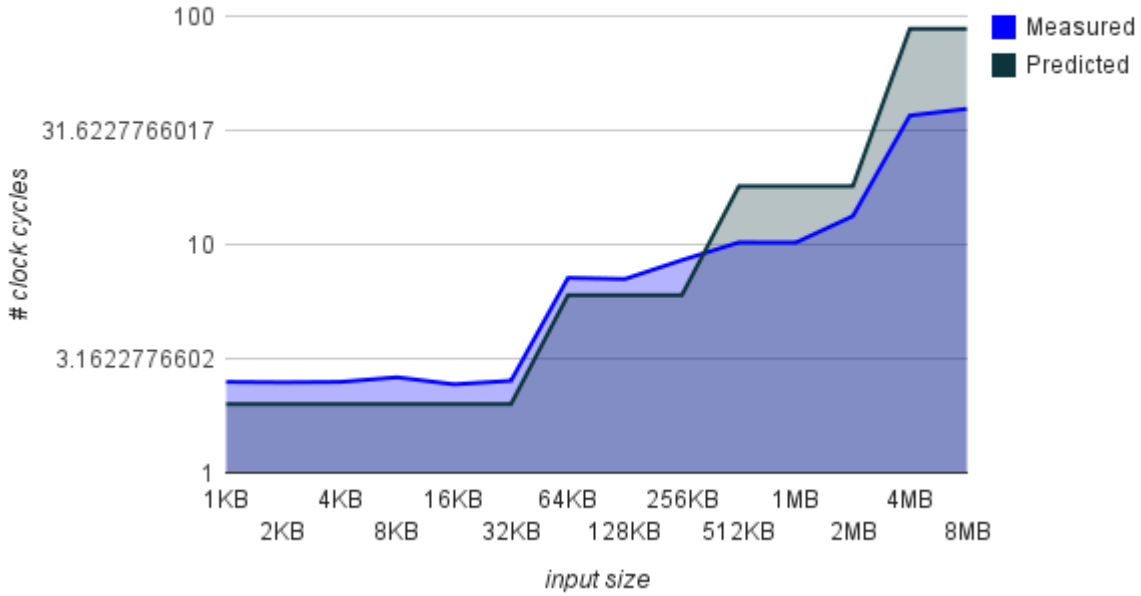
than the size of the cache).

```
 for (i = 0; i<numitems; i++) {
 a[i] = (int*)&a[(i+stride)%numitems];
}
```

3. Walkthrough the array and access all the elements (in random fashion because of the stride) and measure the time to do a complete walk

```
start = count();
 for (i=1; i<=numiters; i++) {
    p = (int*)*p;
}
 end = count();
```

4. Now calculate the loop overhead for step (3) by running an empty loop
5. Subtract loopoverhead from the value calculated in step (3) and it will give the latency to access L integers
6. Run steps (1) through (5) for N number of times (we took 100) so that after the first itertation, data will be served from cache
7. Vary the array length L (and size S) and repeat steps (1) through (6).



Results:

Discussion of results:

1. Source:

1.To identify the cache sizes, we use the system information which is found in /sys/devices/system/cpu/cpu0/cache/index*

2. We referred to various sources in the internet to indetify the access time for various levels of cache and for DDR3 RAM

(But we are not very sure whether the values that we got are the accurate since its not from the manufacturer)

2. Result vs Prediction:

For all the caches and the main memory, calculated cycles are slightly more than the predicted cycles in for L1, L2 caches and for L3 and Main memory, our measurement yields better results than the predicted performance. We believe that this inconsistent behaviour is because of the other processes that are running in the

system which affect the calculated cycles. Though we killed all other processes and ensured that we do not disturb the experiments, there are always some administrative processes running in the system like the windowing system, taskbar, terminal program (which we use to launch our experiment) and the networking system.

We believe that we can close the gap between the experiment results and the predicted results if we can get an ideal environment where only our experiment processes are run.

3. Evaluation:

As said in the previous point, we believe our results are close to the actual performance and hence, we are ~80% successful in our experiments.

4. Analysis of graph:

As seen in the graph, at the points where the data size exceeds the current cache capacity, the number of clock cycles required jumps to a higher value - this is because the access time for the next level cache is higher than the previous level. Jump happens at 32 KB, 256 KB and 3MB.

L1 access time: 2-3 cycles
L2 access time: 8-9 cycles
L3 access time: 10-14 cycles
RAM access time: 35-40 cycles

Though the jump from L1 to L2 and L3 to Main Memory are clearly visible, the jump from L2 to L3 is not apparant. One possible reason could be that the access time for L2 and L3 are almost same.

5. Units used:

All unites are in clock cycle count. Our processor clock speed is 2.3GHz; So a count of 115 clock cycles is roughly equivalent to 0.05 micro second.

RAM Read and Write Bandwidth

Machine description:

Main memory size: 4GB
FSB 1333Hz
Bus width: 32bits (actually 64bit; but since 32bit OS is run, we assume 32bit width)
Number of bits per bus line cycle: 2

Prediction:

From the memory details given above, the theoretical bandwidth can be calculated as follows:

Theoretical bandwidth = 1333 million hertz * 32 * 2 = 85210 Mbits/sec = 10664 MB/sec = 10.39 GB/sec
Software overhead = 0

Predicted bandwidth = 10.39 GB/sec

We referred to the wikipedia article, http://en.wikipedia.org/wiki/Memory_bandwidth to calculate the theoretical memory bandwidth.

Experiment: Measure the RAM read and write bandwidth

Methodology:

Write bandwidth:

Our approach to measure the write beandwidth is to allocate an array of integer of size 1GB in the heap memory and then access the elements of the array and write it to the memory location. To prevent from locality of reference (spatial locality), we chose our stride size (3MB) which is greater than L1, L2 and L3 cache.

We initially calculated the loop overhead and than calculated the total time to write all elements of the 1GB sized array. Then we duducted the loop overhead from the measured time and the result gives the time to write 1GB of memory to the main memory.

Code snippet:

```
46  int stride = 886432;
```

```
….
….
81      for (i=0; i<numiters; i++) {
82          a[i] = randInt;
83      }
84      end = count();
```

Read bandwidth:

We followed the same approach that we used for measuring the write bandwidth. Here, instead of writing the array elements, we read the array elements and measure the time to read the complete array of size 10GB.

Code snippet:

```
91      start = count();
92      for (i=0; i<numiters; i++) {
93          t = a[i];
94      }
95      end = count();
```

Write/Read bandwidth with stride size > 3MB:

We also just wanted to test the bandwith by defining stride size of >3MB. But in this case the bandwidth measured is very less. We have shown this in the table.

Code snippet (write):

```
a[(i*stride)%numiters] = 540000000;
```

Code snippet (read):

```
t = a[(i*stride)%numiters];
```

For the purpose of measuring the read and write bandwidth, sequential access of memory is fine. There will be some prefetching because of the sequential access, but still, even the prefetching time is accounted in the measured end-to-end time delay to read/write 1GB of memory.

	Base HW performance	Software overhead	Predicted performance	Measured performance	Measured performance - with stride
Read bandwidth	10.39 GB/sec	0	10.39 GB/sec	1963.27 MB/Sec	129 MB/sec
Write bandwidth	10.39 GB/sec	0	10.39 GB/sec	916 MB/Sec	100 MB/sec

Discussion of Results:

Source:

We referred to the manufacturer manual to find out the FSB speed and referred to sources on the internet to find out the number of bits transferred on a bus line per bus clock cycle.

Result Vs. Prediction:

There is a gap between our prediction and the measurement. We think this is mainly because of the fact that the theoretical bandwidth can not be reached practically and the theoretical bandwidth calculation just gives the maximum attainable bandwidth - which is possible only when we completely know the operating environment and choose the input data that favors the operating environment.

Evaluation:

We feel that 1963MB/sec read bandwidth and 916 MB/sec write rate is quite reasonable. But we are striving to understand the exact reason for the gap between our prediction and the actual results.

Units:

All units mentioned in this experiment are either in GB/sec or MB/sec - respective units are given alongside the actual numbers.

Page Fault Service Time computation

Disk Info:

Rotation speed: 7200 rpm
Seek time (avg): 12ms
Seek time (track to track): 1ms
Transfer rate: ~50MB/sec
File system block size: 4KB

Prediction:

Hardware performance:

Time to read one disk block:

Seek time: 12ms
Transfer time: 0.08 ms
Total: 12.08 ms

Software overhead:

When a page fault occurs for a page which is mapped in the virtual memory but not found in the main memory, then the following actions have to take place:

1. Save the current execution context: ~ 10s of cycles
2. Switch to kernal mode: ~5 cycles
3. Pagefault handling overhead:
   System call to read the faulting page from the disk: ~1000 cycles
   Handling the completed read (handling interrupt from the hardware): ~100 cycles

4. Switch to user mode: ~5 cycles
5. Resume the execution of the interrupted user process

So the total software overhead is around 1150 cycles.

With the overhead, it will take around 0.5 micro seconds to bring an entire page from the disk to main memory.

Overall prediction per pagefault:

Hardware performance: 12.08 ms
Software overhead: 0.0005 ms

Comparing to hardware performance, software overhead is very less.

Experiment: Measure the Page Fault Service time for an entire page from disk

Methodology:

Page Fault Service Time:

The approach we have taken is, memory map a file which contains some characters and then access the elements in the memory mapped file in a loop such that each iteration tries to access an element from a new page. In order to offset the effect of pre-fetching, we define a jump size which defines the number of pages to skip before accessing the next character. We have changed this jump size until we get a satisfactory standard deviation (skipping 185 pages gave good distribution). Thus, each access to a character will generate a pagefault.

Code Snippet:

```
92    for (i=0; i<samplecount; i++) {
93        start = count();
94        c = buffer[(i*skipSize)%(fileSize-1)];
```

```
95        end = count();
96        pagefaultOverheadArray[i] = end-start-mean_counteroverhead;
97    }
```

Results:

- All units are in micro seconds
- Page size: 4KB

	Base hardware performance (ms/ block)	Software overhead (ms/ pagefault)	Prediction (ms/block)	Measured pagefault service time (ms/block)	Standard deviation percentage
Page fault service time	12.08	0.0005	12.0805	7.89	24%

Discussion of Results:

Source: Manufacturer manual to get the disk parameters like average seek time, transfer rate. Also used the 'hdparam' linux command to find the hard disk performance.

Result Vs. Prediction: Comparing to predicted performance, our measurement is faster (12 ms Vs 7.89 ms). This could be because of the variations in the disk seek time. The prediction is made based on the average seek time of 12ms but, the track-to-track seek time is only 1ms. In our experiment, since we have lots of free disk space and the disk is not fragmented, the blocks will be placed in the nearby cylinders and as a result, the tracks sought were nearby and did not need the complete 12 ms seek time; So the average seek time would have been much lesser than 12 ms.

Evaluation: We feel that the results are accurate since, during every page fault, the disk has to seek to locate the data and the seek is an expensive operation. So pagefault service time of 7.89 ms/block sounds reasonable.

Units:
Units are mentioned alongside with numbers.

Comparison with memory read:
Time to read one byte from memory (based on the memory bandwidth 1963.27 MB/sec): 0.0004 usec
Time to read one byte from disk (random access): 1.92 usec

So, memory is nearly 4500 times faster than disk.

NETWORK OPERATIONS

Round Trip Time:

It is the time required to send a packet over the network to a remote machine (server) and get a reply back. A packet that is addressed to the loopback interface is delivered immediately at the loopback interface.

Prediction:

Network card speed: 1Gbit/sec.
Source: Unix command 'lshw'

Hardware performance:

Loopback interface: There is no hardware overhead since the loopback interface is implemented in software.

Remote interface: Comparing the local area network speed, network interface will not be a bottleneck.

Local Area Network delay: Our LAN is capable of transferring data at 150MB/sec. So we expect a small fraction of a millisecond as network delay.

Software overhead

Create a packet to send and to interpret the received packet (including the processing over the protocol stack) - 10s of micro seconds

Protection domain crossing - 100s of CPU cycles (0.5 to 1 micro second)

Interrupt handling when a packet arrives at the network interface: ~ 5us

Predicted performance:

Based on the base hardware performance and software overhead, RTT should be less than a milli second say 0.3 milli second.

Experiment:

Methodology:

We create two processes, a server and a client process by using sockets and establish a TCP connection between the processes. Once the TCP connection is successfully established, then, we send 1 byte packet from the client to the server and then receive the packet sent by the server in response. The server, sends a 1 byte packet immediately after receiving a packet from the client. We thus calculate the time elapsed to send and receive a packet and it gives the RTT.

We repeat the experiments again and again until we get a satisfactory standard deviation, say below twenty percent. The number of times to repeat the experiment is a configurable value and thus it is easier to change the value and repeat the experiment.

Code snippet:

Server:

```
64        // read and write to the socket
65        while((n = read(newsockfd,buffer,strlen(buffer))) > 0) {
66            //printf("Read: %s\n", buffer);
67            n = write(newsockfd,buffer_wr,strlen(buffer_wr));
68        }
```

Client:

```
81    for (i=0; i<samplecount; i++) {
82        start = count();
83        n = write(sockfd,buffer_wr,strlen(buffer_wr));
84        n = read(sockfd,buffer_rd,strlen(buffer_rd));
85        end = count();
86        pingOverheadArray[i] = end-start-mean_counteroverhead;
87    }
```

Results:

 # all units are in milli seconds

	BaseHW	Software Overhead	Network Overhead	Prediction	Measured Performance	Mean Deviation	PING output
Loopback interface	0	0.05	0	0.05	0.05	0.008	0.042

Remote interface	0.05	0.05	0.2	0.3	0.22	0.03	0.3

Discussion:

Source: Referred to manufacturer manual and used the 'lshw' command to find the network card speed; Used speed test softwares to find the LAN speed.
Comparison: Our predicted value and the measured value are similar.
Evaluation: Since our measured performance is very close to our prediction, we trust that our measurement is accurate.
Comparison with ping: Comparing to ping, our RTT is slightly higher (~8 micro sec). This is acceptable since, the ICMP packets handled at the kernel level whereas our packets should reach our process running in user mode. Also, ICMP packets are smaller in size compared to the IP packets and as a result, the time to construct IP packets is higher.
Comparison - loopback and remote interface: It is expected that the tests with the remote interface will be slower than the loopback interface because of the additional overhead in network and the delay in processing the packets through the network interface card.

From our measurements, we realize that, the OS software adds an overhead of 0.05 ms for the baseline performance of 0.17ms which is about 30% of the baseline network performance.

Bandwidth

Prediction:

Source: Unix command 'lshw'

Hardware performance:
Network card speed: 1Gbit/sec.
Bus speed: ~1GB/sec
Transfer rate of a the network: 150MB/sec

Software overhead:
Create a packet to send and to interpret the received packet (including the processing over the protocol stack) - 10s of micro second
Protection domain crossing - 100s of CPU cycles (0.5 to 1 micro second)
Interrupt handling when a packet arrives at the network interface: ~ 5 us

Predicted performance:
Remote interface: Based on the baseline hardware performance and software overhead, bandwidth will be ~150MB/sec
Loopback interface: Depends on the bus speed which is about 1GB/sec

Experiment:

Methodology:
As before, we create two processes, a server and a client process by using sockets and establish a TCP connection between the processes. Once the TCP connection is successfully established, sever continuously keeps writing data into the network and client keeps reading the data sent by server. We supply a size parameter to the program and once the supplied size of data is received by the client, we record the time taken to receive the said size and then the server sends the same data all over again for the next sample.

The number of samples is configurable and we repeat the experiments with varying samples until we get a satisfactory standard deviation, say below twenty percent.

Code Snippet:

Server:

```
70      // read and write to the socket
71      for (i=0; i<samplecount; i++) {
72          for(index = 0; index < sizeInGB; index++ ) {
73              n = offset = 0;
74              //printf("for the next gig\n");
75              while ((n = write(newsockfd, buffer, PACKETSIZE)) > 0) {
76                  offset += n;
77                  if (offset == STREAMSIZE) {
78              //      printf("Transferred 1G; Brekaing...\n");
79                      break;
80                  }
81              }
82          }
83      }
```

Client:

```
85    for (i=0; i<samplecount; i++) {
86        start = count();
87        for(index = 0; index < sizeInGB; index++ ) {
88            offset = n = 0;
89            //printf("for the next gig\n");
90            while((n = read(sockfd, buffer_rd, PACKETSIZE)) > 0) {
91                offset += n;
92                if (offset == STREAMSIZE) {
93                //  printf("Received 1G; Breaking....\n");
94                    break;
95                }
96            }
97        }
98        end = count();
99        pingOverheadArray[i] = end-start-mean_counteroverhead;
100   }
```

Results:

	Base HW performance	Software Overhead (per GB of data)	Network performance	Prediction	Measured Performance (MB/sec)	Mean Deviation (percent diff)
Loopback interface	1 GB/sec	~10 ms	0	1 GB/sec	760 MB/sec	1.5
Remote interface	1 Gbit/sec	~1s	150 MB/sec	146 MB/sec	115 MB/sec	7

Discussion:

Source:
- Referred to manufacturer manual and used the ‘lshw’ command to find the network card speed
- Used opensource software to find out the LAN speed

Comparison:

Loopback interface: Our measurement is 25% lower than the expected performance and we attribute this degradation to other loads in the system.

Remote interface: Our measurement is 21% lower than the expected performance and this could be due to unexpected delays encountered and packet loss/retransmission in the network. Also, the software that we used to find the LAN speed might not have given accurate network speed.

It is also possible that the OS overhead is more than our predicted value of 1s/GB. In this case, the overhead will bring down the bandwidth.

Evaluation:
Though our results are little off from the prediction, the standard deviation value shows that, our measurements are consistent. So we believe that our results are accurate.

Comparison - loopback and remote interface:
From our measurements, we realize that the OS is adding ~2s/GB of data received. It means that, OS is adding approximately around ~29% overhead to the base network performance, which is 6.8 seconds to transmit 1 GB data (derived from 150 MB/sec speed) . As a result, the peak bandwidth that we achieve is only 115 MB/sec as opposed to the baseline network performance of 150 MB/sec.

Connection Overhead (Setup and teardown)

TCP connection establishment and teardown each require 3 packets to be transmitted between the two parties establishing/closing the connection. So the connection overhead is the time it takes to exchange these packets and establish a successful connection.

Prediction:

Hardware performance:
Network card speed: 1Gbit/sec
Local Area Network delay: Our LAN is capable of transferring data at 150MB/sec. So we expect a small fraction of a millisecond as network delay.

Software overhead
Create a packet to send and to interpret the received packet (including the processing over the protocol stack) - 10s of micro seconds
Protection domain crossing - 100s of CPU cycles (0.5 to 1 micro second)
Interrupt handling when a packet arrives at the network interface: ~ 5us

Prediction:

Loop back:
Connection setup: ~ 0.035 ms (30+1+5 = 35 us)
Teardown: ~0.005 ms (just to initiate the close system call)

Remote:
Connection setup: ~ 0.235 ms (0.035 + 0.2 ms for transferring the packet through the network interface and network delay)
Teardown: ~0.055 ms (0.005ms + 0.05 ms for transferring the packet through the network interface)

Connection teardown is a less expensive operation since, the kernel will take care of closing the tcp connection on behalf of the process which initiates the connection close. The process does not block for any FIN/ FIN+ACK from the server; Kernel will handle these packets for the process.

Experiment:

Methodology:
The server first creates a socket, bind the socket to a port and listens (blocking) in that port for any incoming connection requests. The client, on the other side, also creates a socket and calls the connect system call and sends a SYN request packet to the server. Upon receiving the request, the server creates a new socket to serve the client and sends a SYN+ACK packet to which the client replies with an ACK immediately. Thus it is sufficient to measure the time to execute the ‘connect’ system call at the client side.

Connection tear down is similar to connection setup and here instead of SYN packets, FIN packets are

transmitted by the client. Thus the connection tear down time is the time to successfully execute the ‘close’ system call on the socket descriptor.

Code Snippet:
Server:

```
// connection setup
 59        newsockfd = accept(sockfd,
 60                (struct sockaddr *) &cli_addr,
 61                &clilen);

// connection close
 66        // read and write to the socket
 67        n = read(newsockfd, buffer, 1);
…
 73        close(newsockfd);
```

Client:

```
// connection setup
 71        start = count();
 72        if (connect(sockfd,(struct sockaddr *) &serv_addr,sizeof(serv_addr)) < 0)
 73            error("ERROR connecting");
 74        end = count();

// connection close
 79        start = count();
 80        close(sockfd);
 81        end = count();
```

Results:

Connection Setup	Hardware performance	Software Overhead (per packet)	Network delay	Prediction (ms)	Measured Performance (ms)	Standard Deviation percentage
Loopback	1 Gbit/sec	0.1 ms	-	0.035	0.01	23%
Remote	1 Gbit/sec	0.1 ms	150 MB/sec	0.235	0.31	16%

Connection teardown	Hardware performance	Software Overhead (per packet)	Prediction (ms)	Measured Performance (ms)	Deviation
Loopback	1 Gbit/sec	0.1 ms	0.02	0.004	22%
Remote	1 Gbit/sec	0.1 ms	0.055	0.02	5%

Discussion:

Source: Referred to manufacturer manual and used the ‘lshw’ command to find the network card speed
Comparison: Our measurement for connection setup is close to our prediction. But for connection teardown, measurements are much lesser than the predicted value and this could be because of favourable network connection inaccurate prediction.
Evaluation: We believe that our results are pretty good because of the consistent values that we get across differet runs.
Comparison - loopback and remote interface:
Comparing local and remote connection, connection setup has slowed by 30 times and connection teardown slowed by 5 times. This is in line with our expectation.

File Cache Size

Prediction:

The machine that we use to run the experiment has 4GB of physical memory and we are run a 32bit linux. So the physical memory will be split as 3GB for user process and 1GB for kernel space. The design of linux allows all available free memory to be used as file cache. So if the experiment is run on an ideal machine with no other user level processes running, i.e., the 3GB of user space memory is free, then we predict that, the 3GB space will be used as file cache.

Predicted file cache size: 3GB

Methodology:

In this experiment we will read files of varying size from the disk and measure the read bandwidth. Also, the file caching is enabled and we will read each file multiple number of times. So for the first time the file is read, the program will read it from the disk. But from the second time onwards, it will be read from the file cache. When the file size goes bigger than the cache size, the complete file can not be kept in cache and as a result, for certain blocks there will be cache misses and those blocks will be read from the disk. To accommodate the new blocks, certain blocks in the file cache will be replaced and the blocks thus replaced will be read from the disk during the next read.

As a consequence of reading from disk, there will be a significant drop in the bandwidth when the file size exceeds the file cache size. Another thing to notice is that, once the file size exceeds the cache size, the number of cache misses will be directly proportional to the file size because, only a small portion of the file can be kept in the file cache.

We read each file multiple number of times and the first read will always be from the disk (since it will not be cached). So we omit the readings for the first read and as a result, for files which can be accommodated in the file cache, the resulting bandwidth will be very high (read from file cache). But for files which grows larger than the file cache, each subsequent reads will also need to go to disk and as a result, the time to read a file block will see a sudden jump.

Note: Before we move on to read the next file of bigger size, we manually flush the file cache in order to guarantee the maximum file cache for the file that is to be read.

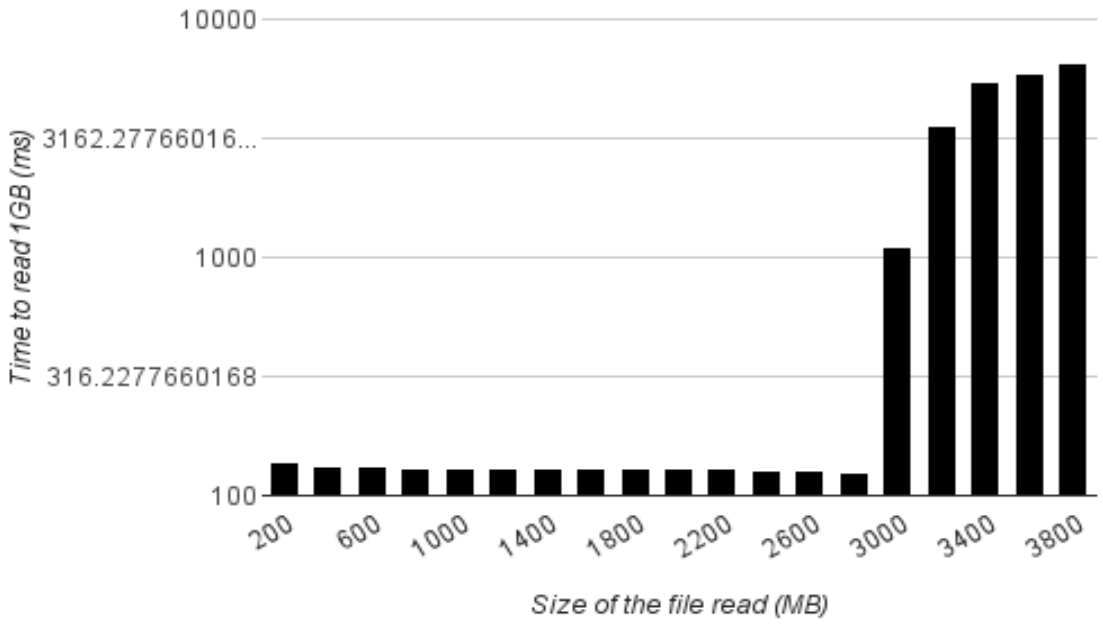
Result:

Predicted file cache size: 3GB

File Size (MB)	Measured time to read a 1GB data (ms)
200	137.19
400	133.45
600	131.39
800	130.69
1000	128.79
1200	129.56
1400	130.23
1600	128.81
1800	129.22
2000	130.53
2200	130.47
2400	127.73
2600	126.67
2800	125.94

3000	1105.22
3200	3582.96
3400	5471.86
3600	5914.94
3800	6580.29

Chart:



Source: OS text book and ubuntu forums to predict how much of memory will be used as file cache
Comparison: Our measured file cache size is exactly same as the predicted file cache size
Evaluation: The cache size found by our experiment concurs with our prediction. So we feel that, the results are 100% accurate.

Graph:

As the file size increases, the average time to read remains constant until 3GB and there is a huge jump in read time afterwards. This is because, until 3GB, the files can be held in file cache and hence the constant and lesser read time. But once the file size is bigger than cache (at least a little), it triggers disk activity and thus the read time suddenly increases because the file is now read from disk. So, after this point, as the file size increases, the read time also keeps increasing with the file size and this is because of the increasing cache misses (increased disk activity).

So from the graph, it is very clear that the maximum possible file cache size is 3GB.

File Read Time

Prediction:

Base hardware performance:

Disk Info:
Rotation speed: 7200 rpm
Seek time (avg): 12ms
Seek time (track to track): 1ms
Transfer rate: ~50MB/sec
File system block size: 4KB

We are measuring the time to read one disk block by sequential and random access.

Time for sequential access:

We assume that there will not be much of disk seeks to access the disk blocks for a sequential access. So, the time to read a disk block depends on the transfer rate.

Sequential read time: 0.08 ms/block

Time for random access:

Random access will trigger disk seek and hence, seek time will slow down the disk read time.
Random read time: 12ms + 0.08 ms = 12.08 ms/block

Software overhead:

Read system call: ~1000 cycles
Handling the interrupt from the disk controller after the read is complete: ~10000 cycles
Copying data to the user buffer: ~200 cycles

Software overhead: 0.005 ms

Comparing to the disk performance, the software overhead is very small.

Prediction:

Sequential read: 0.085 ms/block
Random read: 12.085 ms/block

The machine that we use to run the experiment is a relatively new machine (2 months of usage) and the hard drive has lots of free space and not much of disk activity (file creation, file deletion etc). So we expect that the disk is not fragmented and even larger files will not be scattered much. As a result, the sequential and random read time should be similar for varying file sizes. But, when the file size is smaller than the disk controller cache, the subsequent reads after the first read will be served from the cache and as a result, for these files, the average read time should be very small comparing to bigger files.

Experiment:

Methodology:

In this experiment, we want to measure the disk read performance and hence, we skip the file cache while reading the file from disk i.e., the content read will not be cached in the file cache so, successive read requests for the same file will go to disk. To do this, we use the ‘O_DIRECT’ flag with the open system call.

Sequential read:

To simulate sequential read, we continuously keep reading the file using read system call until there are no more bytes to read.

Random read:

We have generated random read by doing a seek after each read (each read reads a single block) to skip a certain number of blocks before next read. So every successive read attempt will try to access a disk block that is not near the disk head and hence it will generate disk seek. There is a slight overhead with the iterative read and seek system call for every read and we have taken care to deduct these overheads from the measurement.

The number of blocks to skip and the number of samples to take is configurable in our program. We ran the program many times altering these parameters until we get a minimum standard deviation of the measurements.

Code snippet:

Random read

```
142     //read randomly
143     for (index = 0; index < samplecount; index++) {
144
145         if (lseek(inputFileDesc, 0, SEEK_SET) == -1)
146             error("Error seeking");
147
148         numBlocksReadRand = 0;
149
150         start = count();
151         while (1) {
152             bytesRead = read(inputFileDesc, buffer, BLOCKSIZE);
153
154             if (bytesRead <= 0)
155                 break;
156
157             numBlocksReadRand++;
158
159             if (lseek(inputFileDesc, jumpSize*BLOCKSIZE, SEEK_CUR) == -1)
160                 error("Error seeking");
161         }
162         end = count();
163
164
165         printf("Blocks read rand: %llu\n", numBlocksReadRand);
166         randOverheadArray[index] = end-start-mean_counteroverhead\
167                                 -(mean_loopoverhead*numBlocksReadRand)\
168                                 -(mean_seekoverhead*numBlocksReadRand);
```

Sequential Read

```
173     //read sequentially
174     for (index = 0; index < samplecount; index++) {
175
176         if (lseek(inputFileDesc, 0, SEEK_SET) == -1)
177             error("Error seeking");
178
179         numBlocksReadSeq = 0;
180         start = count();
181         while (1) {
182             bytesRead = read(inputFileDesc, buffer, BLOCKSIZE);
183             //if ((++numBlocksReadSeq == numBlocksReadRand) || bytesRead <= 0 )
184             if (bytesRead <= 0 )
185                 break;
186             ++numBlocksReadSeq;
187
188         }
189         end = count();
190         printf("Blocks read seq: %llu\n", numBlocksReadSeq);
191         seqOverheadArray[index] = end-start-mean_counteroverhead\
192                                 -(mean_loopoverhead*numBlocksReadSeq);
```

Results:

Prediction:

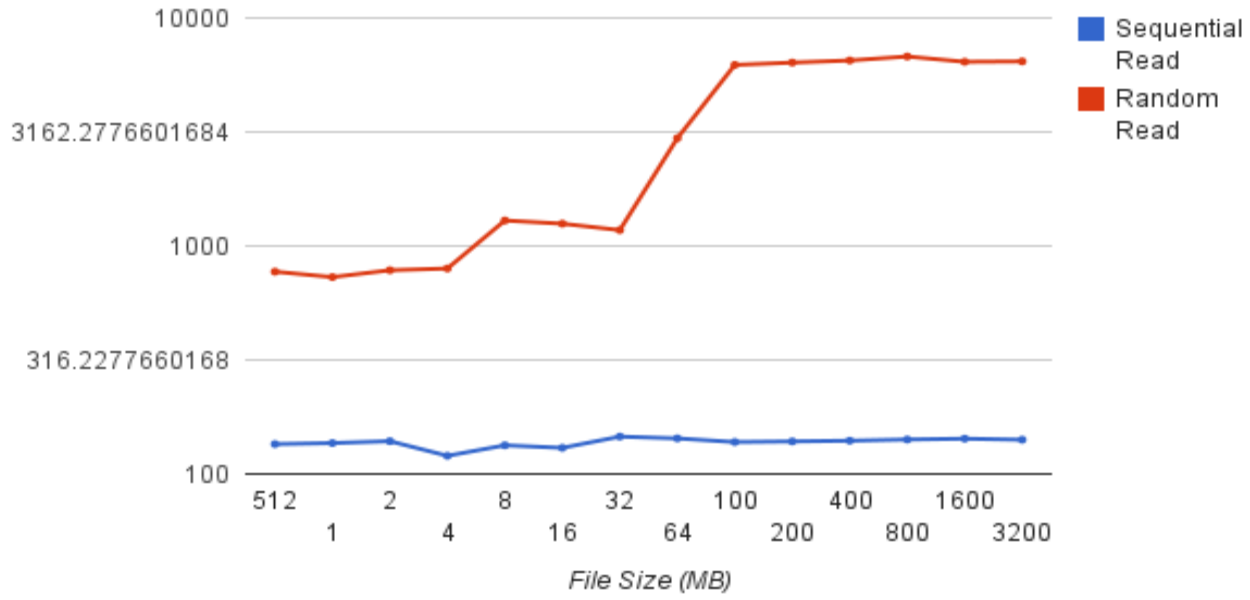
Sequential read (for all file sizes): 85 us/block
Random read (for all file sizes): 1208 us/block

File size MB	512 KB	1	2	4	8	16	32	64

Sequential read (us/ block)	135.220	136.651	139.270	120.146	133.580	130.215	145.871	143.249
Seq read Standard deviation	22.135	12.732	6.892	12.576	1.621	19.66	6.236	32.754
Random read (us/ block)	771.439	730.311	782.801	796.630	1293.563	1252.641	1173.959	2968.806
Rand read standard deviation	161.145	138.852	163.196	285.296	163.196	039.243	99.418	49.558

File Size MB	**100**	**200**	**400**	**800**	**1600**	**3200**	**6400**
Sequential read (us/ block)	138.087	138.949	139.944	141.637	142.828	141.345	139.772
Seq read Standard deviation	4.645	4.523	1.958	2.693	1.238	001.253	0.930
Random read (us/ block)	6226.170	6373.426	6512.719	6777.178	6424.973	6450.120	6868.527
Rand read standard deviation	196.274	130.083	220.606	416.642	61.565	77.203	79.945

Graph:



Discussion:
Source: Manufacturer manual to get the disk parameters like average seek time, transfer rate. Also used the ‘hdparam’ linux command to find the hard disk performance.

Comparison: As expected, the random read time per block is very small for files which are smaller than the disk cache (retrieved from cache and hence avoids disk seeks) and for other files, it remains same irrespective of the file size. This is because, the disk has lots of free space and the extent to which blocks are scattered are same for small files as well as large files.

The predicted sequential read time is 0.085 ms/block and the measured time is 0.13 ms/block. This could be because the practical data transfer rate acheivable is different from the advertised data transfer rate.

The predicted random read time is 12.085 ms/block and the measured time is ~6.5 ms/block - prediction is nearly double of the obtained result. This could be because of the variations in the disk seek time. The prediction is made based on the average seek time of 12ms but, the track-to-track seek time is only 1ms. In our experiment, since we have lots of free disk space and the disk is not fragmented, the blocks will be placed in the nearby cylinders and as a result, the tracks sought were nearby and did not need the complete 12 ms seek time; So the average seek time would have been much lesser than 12 ms.

Evaluation: Since we get consistent results across different file sizes, we believe that our measurements are close to accurate.

Graph:
There is huge gap between the sequential read and random read and this is mainly because of the time spent in seeking the disk which is an expensive operation. Sequential read is nearly 45 times faster than random read.

Also, when the files size start to exceed the disk controller cache, the random read time also starts to increase. Sequential reads does not benefit much from the disk controller cache since sequential reads are already faster and does not need expensive seeks.

When a sequential read may not be a actual sequential read:

If the disk is heavily fragmented and has very little free space left, then it is likely that there are no contiguous free space. As a result, the file system can not optimally lay all the blocks of a file and the blocks will get scattered into different tracks. As a result, even a sequential access will need to do lots of seeks resulting in a reduced throughput that is same as random read.

Remote File Read

Remote machine configuration:

The read time (both sequential and random) of the remote machine (from which we read the file) hard drive is similar to the local machine on which we run all the experiments.

Component	Description
Processor	Intel(R) Core(TM)2 Duo CPU CPU MHz : 2526.957 address sizes : 36 bits physical, 48 bits virtual
Cache	L1 Instruction : 32KB L1 Data : 32KB L2 : 3072KB
Memory	4 GB RAM FSB : 1066 Mhz
Disk	Rotational Speed : 5,400 RPM Buffer Size : 8 MB Read Seek Times: 12ms Track-To-Track read time: 2ms
Network	Intel 82567LM Gigabit Network Ethernet
I/O Bus	66Mhz
Operating System	Ubuntu 11.10
Kernel	3.0.0-12-generic-pae

Prediction:

Hardware performance:

The objective of this experiment is to find out the network penalty while reading files from a remote machine. So we measure the performance of remote machine and network and use that as baseline hardware performance for our prediction.

Network bandwidth (measured): 115 MB/sec
Sequential Read bandwidth of remote machine (measured): ~29 MB/sec
Random read bandwidth of remote machine (measured): 0.6 MB/sec

Since the network speed is relatively high comparing to remote disk read speed, we expect that the network should not impact the read time from remote machine.

Software overhead:

The OS has to process the incoming packets and it will involve some extra overhead.
OS interrupt processing: 100 us/block

So our prediction is same as the local disk read times.

Remote sequential read: 0.234 ms (0.134 ms + 0.1 ms OS overhead)
Remote random read: 6.6 ms (6.5 ms + 0.1 ms OS overhead)

Experiment:
Methodology:

We have done previous experiment (local file read) on machine X and now we are trying to read from the

disk of machine Y. So, Y is the remote machine and X is the local machine. IOW, earlier, on X, we were reading from the X's hard disk and now, on X, we are reading from Y's hard disk.

We have setup and nfs server on the remote machine Y and mounted the filesystem of Y on machine X. After this step, the experiment is same as the previous experiment but with one difference: now files are read from the remote machine's hard drive through the network.

The program we used remains same.

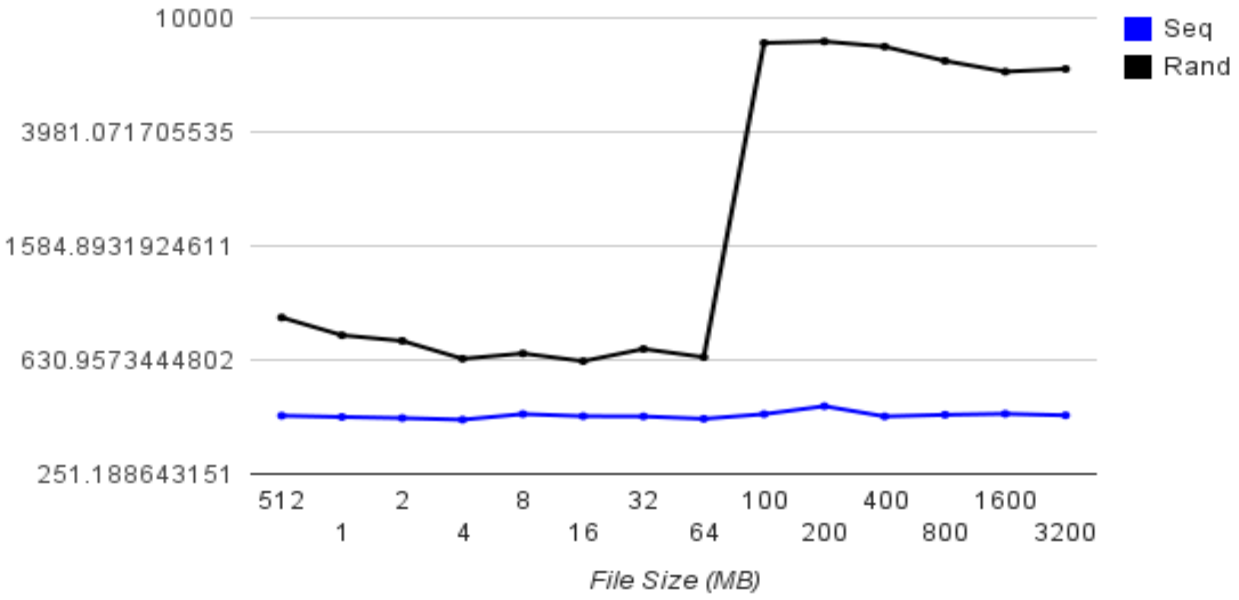
Results:
Prediction:

Sequential read (for all file sizes): 234 us/block
Random read (for all file sizes): 6600 us/block

	512	1	2	4	8	16	32	64
Sequential read (us/ block)	402.237	398.185	394.602	389.5	407.67	400.708	399.969	392.273
Random read (us/ block)	889.263	770.676	736.222	637.354	665.102	665.102	689.678	645.655

File Size (MB)	100	200	400	800	1600	3200
Sequential read (us/ block)	407.432	434.389	400.001	405.208	408.468	403.509
Random read (us/ block)	8174.680	8273.163	7933.989	7070.477	6481.920	6624.631

Graph:



Discussion:

Source: For prediction purpose, we measured the performance of the remote machine and used that as baseline performance.

Comparison:

Sequential read: We predicted that the sequential remote read will take 0.234 ms/block but the measurements show that, it takes ~0.4 us/block. This could be because of some additional overhead in the processing of the incoming the network packets. Comparing to local read, network read is nearly three times slower and we believe that this is because, sequential reads are faster and hence the delay caused by network shows up.

Random read: We predicted that the remote random read will take 6.6 ms/block but as per the measurements, it takes between 6.6 ms/block to 8.2 ms/block. Again, we believe that this is because of some unexpected delay in the network transfer. Comparing to local read, on average, network read remains comparable to local read. This is because, random disk read is really slow and thus, the disk becomes the bottleneck for reading through network thus offsetting the penalty caused by network.

Evaluation: For sequential reads, there is ~185% increase (almost triples) in read time where as for random reads, at the worst case, there is an increase of 26% in the read times. We feel that this increase is acceptable for remote read; So we believe that our results are accurate.

Graph:

Two important features:

1. The read time for sequential access remains constant across different file sizes - this is because, sequential reads does not benefit from the disk cache since there is no seek.

2. The random read time is very less for smaller files because of the disk controller cache and for larger files (bigger than cache), the random read time decreases a little bit (around 1 ms) with respect to the increase in file size. This is because, when the file size is smaller, the network pipe is not fully packed and as a result, the overhead of transferring through the network shoots up the read time. But when the file size grows larger, the network pipe will be fully packed with data and there will be continuous flow of data in the network; Hence, the delay caused by the network is offset by the slower disk read and thus, the read time is slightly lesser for as the size of the file grows.

Average Network penalty for sequential read: 0.26 ms/block
Average Network penalty for random read: less than 0.7 ms/block

Contention

Prediction:

As the number of processes increase, the disk read time will also increase in proportion. Because, when different processes are accessing different files in the system, which are probably placed in different locations on the disk, during every context switch, the disk has to seek in order to access the file the current process wants to read.

During the course of a file read, the number of context switches will increase with the number of new additional processes and for each context switch, the disk will incur a seek. So during each time quantum, time for an additional seek will be added to the actual time to read a particular number of blocks.

Suppose that in a given time slice the process reads 100 disk blocks, earlier when the process was running alone, it will take 100*block_read_time. Now, it will be 1 seek time + 100*block_read_time.

Hardware performance:

Disk Info:
Rotation speed: 7200 rpm
Seek time (avg): 12ms
Seek time (track to track): 1ms
Transfer rate: ~50MB/sec
File system block size: 4KB

We are measuring the time to read one disk block by sequential and random access.

Time for sequential access:
We assume that there will not be much of disk seeks to access the disk blocks for a sequential access. So, the time to read a disk block depends on the transfer rate.

Sequential read time: 0.08 ms/block

Time for random access:
Random access will trigger disk seek and hence, seek time will slow down the disk read time.
Random read time: 12ms + 0.08 ms = 12.08 ms/block

Software overhead:
Process context switch time: few micro seconds
For every additional process, there will be a cost one one additional seek

Prediction:

Sequential read:
100 us/block increase for every additional process
Random read:
Around one additional seek time for every additional process for every block access

Experiment:

Methodology:
For this experiment, we follow the approach that we did for the ‘File Read’ experiment. In addition, we create new contention processes, which do nothing but keep reading files in an infinite loop - sole purpose is to generate a lot of disk activity. So we measure the file read performance by increasing the number of contention processes.

Pseudo code - contention process:

```
53    while(1) {
54
55        if (lseek(inputFileDesc, 0, SEEK_SET) == -1)
56            error("Error seeking");
57
58        if (doRandom) {
59            while (1) {
```

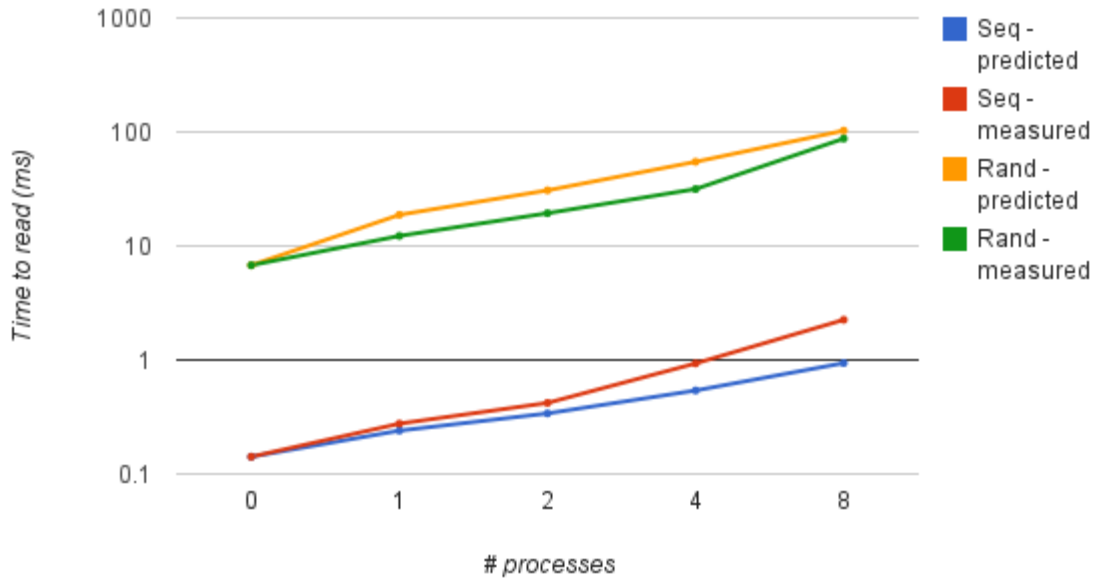
```
60              bytesRead = read(inputFileDesc, buffer, BLOCKSIZE);
61
62              if (bytesRead <= 0)
63                  break;
64
65              if (lseek(inputFileDesc, jumpSize*BLOCKSIZE, SEEK_CUR) == -1)
66                  error("Error seeking");
67          }
68      }
69      else {
70          while (1) {
71              bytesRead = read(inputFileDesc, buffer, BLOCKSIZE);
72              if (bytesRead <= 0)
73                  break;
74          }
75      }
76
```

As seen in the code, the contention process simply makes some disk activity by continually reading a file. Different contention processes will be reading different files thus generating a lot of disk seek and rotation.

Results:

	0	1	2	4	8
Predicted Sequential Read time (ms/block)	0.141637	0.24	0.34	0.54	0.94
Sequential Read time (ms/block)	0.141637	0.276480	0.420096	0.933163	2.251665
Predicted Random read time (ms/block)	6.777178	18.77	30.77	54.77	102.77
Random read time (ms/block)	6.777178	12.255274	19.387028	31.585334	87.482935

Graph:



Discussion:
Source: Manufacturer manual to get the disk parameters like average seek time, transfer rate.

Comparison: For sequential read, till the number of processes are 4, our prediction of linear increase matches well with the measurement. But, as the number of processes increase, the time to do sequential read increases exponentially as seen in graph. We reason that this is because of the increased overhead (because of increased number of processes) in context switch timings.

For random read, measurements are very close to our prediction.

Evaluation: Both the sequential and random read time increases in proportion with the number of contending processes. So, it looks like our results are good.

Summary

	Base hardware performance	Software overhead	Predicted time	Measured time
Counter overhead (# clock cycles)	100-200	0	100-200	163
Loop overhead per iteration(# cycles)	6	0	6	20
Procedure call - additional cycles per argument	3	0	3	3-4
System call overhead (# cycles)	700-1000	6-7	700-1000	46
Process creation (usec)	6-7	400	406	148
Kernel thread creation (usec)	6-7	30-40	36-46	28

Process switch (usec)	8-10	20-40	28-50	27
Kernel thread switch	8-10	2-5	10-15	4.08
L1, L2, L3 acccess time (cycles)	2, 4, 12	0, 0, 0	2, 6, 18	3, 8, 11
Main memory access (cycles)	88	0	88	55
Read bandwidth	10.39 GB/sec	0	10.39 GB/sec	1963.27 MB/sec
Write bandwidth	10.30 GB/sec	0	10.39 GB/sec	916 MB/sec
Page fault service time (ms)	12.08	0.0005	12.0805	7.89
Ping (lb, remote) (ms)	0, 0.25	0.05, 0.05	0.05, 0.30	0.05, 0.22
Bandwidth (lb, remote)	1GB/sec, 1Gbit/sec	10ms (per gb), 1s (per gb)	1GB/sec, 146 MB/ sec	760 MB/sec, 115 MB/sec
Connection Setup (lb, remote) (ms)	0, 150 MB/sec (nw delay)	0.1 ms, 0.1 ms	0.035 ms, 0.235 ms	0.01, 0.31
Connection tear down (lb, remote) (ms)	0, 0	0.1 ms, 0.1ms	0.02 ms, 0.055ms	0.004 ms, 0.02 ms
File cache	-	-	3GB	3GB
File Read per block - sequential	0.08 ms	0.005 ms	0.085 ms	0.13 (ms)
File Read per block - random	12.08 ms	0.005 ms	12.085 ms	6.5 (ms)
Remote read per block- sequential	115 MB/sec n/w delay	0.1 ms	0.234 ms	0.4 ms
Remote read per block - random	115 MB/sec n/w delay	0.1 ms	6.6 ms	6.6 ms to 8.2 ms
Contention - sequential read - increase in read time per additional process	-	100 us	100 us	130 us
Contention - random read - increase in read time per additional process	-	12.085 ms (1 seek time)	12.085 ms (1 seek time)	6 ms