PROJECT REPORT CSC-501

Title: Performance Measurement

Group Member: Syed Amer Zawad Student id: 200132685 (sazawad)
Group Member: Sourabh Saha Student id: 200157857 (sssaha2)

Index

Table of work allocation

Operation	Performed by
Measuring time overhead	Sourabh Saha
Measuring Loop overhead	Syed Zawad
Procedure call overhead	Sourabh Saha
System call overhead	Syed Zawad
Process/Thread creation overhead	Sourabh Saha
Context switch overhead	Syed Zawad
RAM access latency	Sourabh Saha
RAM Bandwidth	Syed Zawad
Page fault service time	Syed Zawad & Sourabh Saha
Report creation	Syed Zawad & Sourabh Saha

Introduction

The project concerns itself with benchmarking various aspects of the underlying hardware components of a computer like CPU, RAM, Caches etc. The main intent of the project was to understand the metrics of performance and through this understanding, implement a set of measurement programs for the components mentioned in the subsequent sections of this report.

The language in which the programs have been implemented is C, since it is considered to be one of the fastest executing languages when it comes to benchmarking hardware. The compiler used is gcc compiler for ubuntu (version 5.4.0 20160609) with default optimization settings.

The amount of time spent on this project would be no less than 60 Hours for each group member.

Machine specifications

Processor model:

Intel core i7-4710HQ (8 cores)

Cycle time:

2.50GHz

Cache sizes:

L1: 32KB (Instruction)
L1: 32KB (Data)
L2: 256KB
L3: 6144KB

RAM

width: 64bits
Size: 8192MB
Type: DDR3
Synchronous Speed: 1600MHz
Total slots: 4 (used 1)

OS Details:

Kernel: 4.2.0-42-generic x86_64 (64 bit gcc: 5.2.1)
Desktop: MATE 1.10.2 (Gtk 3.16.7-0ubuntu3.3)
Distro: Ubuntu 15.10 wily
Laptop: Lenovo Y50-70

Methodology common to all experiments

1. A shell script is run to execute the benchmark operation
2. The operation is run for 100000 to 200000000 times depending on when it is appropriate to get mean clock cycle value
3. The first 1 million iterations is to warm-up the CPU and the next million values are averaged to get final output
4. A “result” file is generated and this is given as input to a Java file which calculates the mean and standard deviation
5. Values beyond a realistic range are not considered as they significantly affect the mean
6. Standard deviation is calculated so as to best get the variance in the resultant values
7. CPUID and RDTSCP instructions are used to measure time (as they result in the most granular values) as per the paper [1]
8. All processes are confined to one single CPU core through the use of setAffinity function
9. The priority of the processes are boosted using “nice” property
10. The assembly code was viewed (using -S flag of gcc compiler) and the order of instructions was used to predict execution time

sched_setaffinity() sets the CPU affinity mask of the thread whose ID is *pid* to the value specified by *mask*. If *pid* is zero, then the calling thread is used. A thread's CPU affinity mask determines the set of CPUs on which it is eligible to run. On a multiprocessor system, setting the CPU affinity mask can be used to obtain performance benefits.[2]

nice is a program found on Unix and Unix-like operating systems such as Linux. It directly maps to a kernel call of the same name. nice is used to invoke a utility or shell script with a particular priority, thus giving the process more or less CPU time than other processes.[3]

Cold cache: When the cache is empty or has irrelevant data, so that CPU needs to do a slower read from main memory for your program data requirement.
Hot cache: When the cache contains relevant data, and all the reads for your program are satisfied from the cache itself [4]

RDTSC - returns 64-bit timestamp counter, which is increased on every clock cycle. It is one of the most precise counter available on x86 architecture [1][5]

CPUID - Returns processor identification and feature information to the EAX, EBX, ECX, and EDX registers, according to the input value entered initially in the EAX register [1][6]

Evaluation of experiments

1. While timing the experiments we have not excluded the effect of inherent OS optimization policies and mechanisms like scheduler preemption or guaranteeing exclusive ownership of the CPU programmatically
2. The code includes positions which hints the OS to boost process priority and setting affinity of the process to one core
3. Any operation specific overheads have been mentioned in their respective sections
4. All code was executed using the -O flag as mentioned in this paper [7]

Measurement overhead

For measuring overhead regarding reading the time itself and the overhead of a single loop iteration,we have:

1. Analyzed the code sample presented in the intel paper [1]
2. Used CPUID, RDTSCP instructions

Measuring time:

Base hardware performance	Software overhead estimation	Predicted operation time	Measured operation time
1 cpu cycle = 0.4 ns (2.5 GHz processor)	This is the most basic operation. Any overhead would be generated only by assembly instructions	5 cycles or 2 ns	23 cycles or 8 ns

We have referred to a manual [8] to obtain average latencies of different assembly instructions and to */proc/cpuinfo* to gather other details of our system (For example CPU Clock speed)

Measurement of the overhead of measuring time itself was done through through finding difference between the start and end timestamp counter values when there is no code written in between. The difference in predicted and actual might be due to scheduling delays in between RDTSC instructions.

The prediction of the operation time was done considering the overhead of looking at the assembly code (using the -S flag)

```
CPUID
RDTSC
mov %edx, %edi
mov %eax, %esi
movl    %edi, -96(%rbp)
movl    %esi, -92(%rbp)
RDTSCP
mov %edx, %edi
mov %eax, %esi
CPUID
```

There are 4 mov instructions between the two RDTSC instructions and hence we predicted 5 cycles considering that MOVL takes 1 cycles and loading the timer takes 1 cycle

Measuring loop overhead (1 iteration):

Base hardware performance	Software overhead estimation	Predicted operation time	Measured operation time
1 cpu cycle = 0.4 ns (2.5 GHz processor)	23 cycles or 8 ns	9 cycles or 3.6 ns	2 cycles or 0.8 ns

Measured operation time (and Predicted operation time) = Value returned by code - timing overhead

The prediction was done on the premise that for initialization and running a for loop, the constituent assembly level instructions will have their overhead, along with the overhead of reading the time itself.

```
/*For loop body*/
   movl    %edi, -112(%rbp)
    movl    %esi, -108(%rbp)
    movq    $1, -88(%rbp)
    jmp    .L7
.L8:
    sall    -116(%rbp)
    addq    $1, -88(%rbp)
.L7:
    movq    -88(%rbp), %rax
    cmpq    -56(%rbp), %rax
    jl    .L8
```

A for loop consists of two MOVL, two MOVQ, one JMP, one JL, one CMPQ, one SALL and one ADDL instruction giving an expected total of 9 CPU clock cycles. But the measured value turns out to be 2 CPU clock cycles and the reason for this might be the innate optimization strategies applied by the CPU architecture.

Current Intel 64 architectures (since the PC in which we performed the experiment has this architecture) have optimization strategies like out-of-order execution and the scheduler within the processor core can process upto eight micro-operations per clock cycle. This, we assume, would cause the different instructions in the for loop to be executed parallely (For example, the MOVL statements) [9]

Procedural call overhead

Methodology:

For measuring time regarding overhead of a procedure call,we have:

1. Measured the time taken to execute a procedural call of type void
2. Measured the time taken by adding arguments incrementally
3. Plotted a graph depicting the execution time in y-axis and the number of arguments supplied in the x-axis

Base hardware performance	Software overhead estimation	Number of arguments	Predicted operation time	Measured operation time
1 cpu cycle = 0.4 ns (2.5 GHz processor)	23 cycles (read time) = 8 ns	0	9 cycles	2 cycles
		1	10 cycles	2 cycles
		2	11 cycles	3 cycles
		3	12 cycles	3 cycles
		4	13 cycles	4 cycles
		5	14 cycles	5 cycles
		6	15 cycles	5 cycles
		7	16 cycles	6 cycles

Measured operation time (and Predicted operation time) = Value returned by code - timing overhead

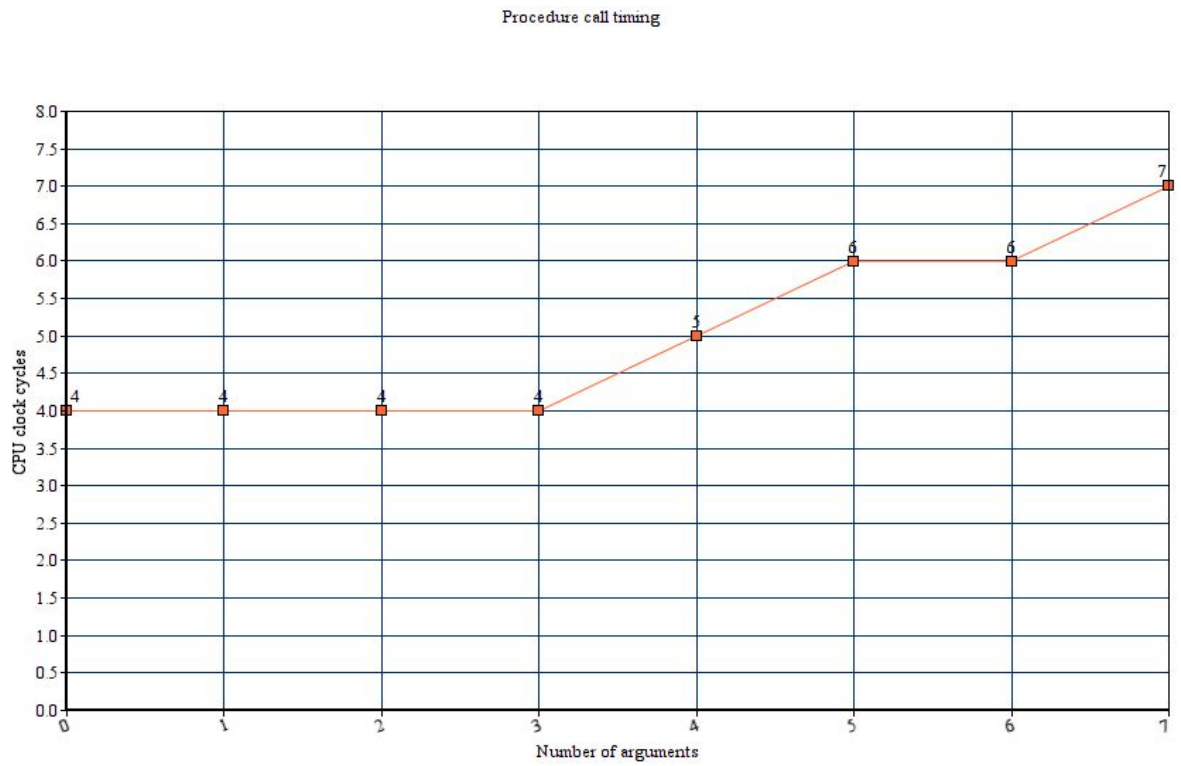
The prediction was done on the premise that for a procedure call, the constituent assembly level instructions will have their overhead, along with the overhead of reading the time itself.

```
/*calling the procedure with args 0*/
   movl    %edi, -96(%rbp)
   movl    %esi, -92(%rbp)
   movl    $0, %eax
```

```
    call    procedure0
/*Actual called procedure body*/
    .cfi_startproc
    pushq    %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset 6, -16
    movq    %rsp, %rbp
    .cfi_def_cfa_register 6
    nop
    popq    %rbp
    .cfi_def_cfa 7, 8
    ret
    .cfi_endproc
```

A procedure call consists of one MOVQ, two MOVL, one CALL, one RET, one NOP, one PUSH, one POP and n additional MOVL instructions (depending on the n arguments) giving an expected total of 9+n CPU clock cycles. But the measured value turns out to be non-linear variation CPU clock cycles

The out-of-order execution engine and the scheduler within the processor (the one we have used has 4 ALUs) would cause, we assume, the different instructions in the call body to be executed parallely (For example, the MOV statements) [9]



.

System call overhead

Methodology:

For measuring overhead regarding overhead of a system call,we have:

1. Measured the time taken to execute the system call getppid()
2. Iterated the process over 1 million times to get correct reading

getppid() was chosen as it had the same cost as getpid() (considered to be the cheapest linux system call) and it does not get cached every iteration

Base hardware performance	Software overhead estimation	Predicted operation time	Measured operation time
1 cpu cycle = 0.4 ns (2.5 GHz processor)	23 cycles or 8ns (read time) and privilege switching time (once for user-kernel and for kernel-user)	36 cycles or 14 ns	107 cycles or 42 ns

Measured operation time (and Predicted operation time) = Value returned by code - timing overhead

The prediction was done on the premise that for a system call, the constituent assembly level instructions will have their overhead, along with the overhead of reading the time itself and switching in and out of privilege levels.

```
movl    %edi, -96(%rbp)
movl    %esi, -92(%rbp)
call    getppid
```

The call contains two MOVL instructions and one CALL instruction which gives a total of 4 cycles along with time taken for switching privilege levels. The referenced paper [10] claims 1.68 microseconds or 223 cycles for a 133 MHz processor, which led us to assume that the kernel system call would take 90 nanoseconds or 36 cycles for our 2.5 GHz processor (excluding the calculation overhead of reading time).

The difference in the actual and predicted value might be due to the reason that increasing frequency would not necessarily result in a linear decrease in number of cycles required to execute. There are some unavoidable kernel overheads present when timing the system call.

Process/Thread creation overhead

Methodology:

For measuring overhead regarding creation of a process:

1. Measured the time taken for fork and execvp, between the parent process and the child process
2. The parent process starts the timer and the child process ends the timer
3. Iterated the process over 1 million times to generate a set of values in terms of cpu cycles
4. Java program calculates the mean number of cpu cycles and the value entered in the table below is the result of multiplying the mean with 0.4 ns (2.5 GHz processor)

fork() creates a new process by duplicating the calling process. The new process is referred to as the *child* process. The calling process is referred to as the *parent* process. The child process and the parent process run in separate memory spaces [11]

The **exec**() family of functions replaces the current process image with a new process image. **execvp**() functions provide an array of pointers to null-terminated strings that represent the argument list available to the new program. [12]

For measuring overhead regarding creation of a kernel thread:

1. Measured the time taken for thread create and join, between the main thread and the created thread
2. The main program starts the counter and the created thread stops the timer before joining the main program
3. Iterated the process over 1 million times to generate a set of values in terms of cpu cycles
4. Java program calculates the mean number of cpu cycles and the value entered in the table below is the result of multiplying the mean with 0.4 ns (2.5 GHz processor)

In the UNIX environment a thread:

- Exists within a process and uses the process resources
- Has its own independent flow of control as long as its parent process exists and the OS supports it
- Duplicates only the essential resources it needs to be independently schedulable
- May share the process resources with other threads that act equally independently (and dependently)
- Dies if the parent process dies - or something similar
- Is "lightweight" because most of the overhead has already been accomplished through the creation of its process. [13]

Base hardware performance	Software overhead estimation	Predicted operation time	Measured operation time
1 cpu cycle = 0.4 ns (2.5 GHz processor)	sys call overheads for fork and exec	200 microseconds for fork and exec combined	174 microseconds for a fork+exec process
	thread allocation overhead (pthread_create and pthread_join)	2 microseconds	2 microseconds for a kernel thread creation

The prediction was done on the basis of guessing the overheads involved in making the system calls (fork and exec). Since the code only creates a child process and exits immediately, the total time taken for the operation would include the time for starting the timer, switching from user mode to kernel mode for fork(), cloning the memory space of the parent, switching back to user space, executing the system call execvp with similar overhead and lastly stopping the timer.

The referenced paper [14] reported a time of 0.3 milliseconds for a fork() on a 933 MHz Pentium III processor, which (if we assume linear inverse-proportionality) yields 100 microseconds for a fork() on a 2.5 GHz Intel Core-i7. So, we predicted fork+exec would take approximately 200 microseconds.

For the kernel thread creation, the only considerable overhead is the switching of user mode to kernel mode and back, since the creation of the thread is inexpensive as it uses the same memory space as the process. Hence, we predicted the kernel thread execution time would be considerably cheaper than creating a new process.

The prediction for pthread creation was based on this source [13], which (for 2.4 GHz Intel Xeon) states an approximate fork() to pthread_create() ratio of 50:1. This led us to the estimate that for 100 microseconds taken by fork() process (for our processor) would yield 2 microseconds for pthread creation.

The measured values between process and a kernel thread is so different because when a process is created:

- System calls like fork and exec generate their own overhead
- There is an overhead to duplicate the memory space for the child process

And when a kernel thread is created the above overheads are not present.

Context-switch overhead

Methodology:

For measuring overhead regarding context switch of a process:

1. Created a child process using fork()
2. Created a pipe in the parent process before the fork()
3. Started the timer in the child process
4. We wrote a byte to the pipe buffer
5. Inside the parent, the pipe was read (this step indicates the context switch)
6. The timer is stopped once the child exits and the timer difference is given to an output file
7. Java program calculates the mean number of cpu cycles and the standard deviation from this output file and the value entered in the table below is the result of multiplying the mean with 0.4 ns (2.5 GHz processor)

For measuring overhead regarding context switch of a kernel thread:

1. We used the pthread library to start the kernel thread
2. The main program starts the counter and the created thread stops the timer before joining the main program
3. Inside the main program a pipe is created before creating the thread
4. Then inside the main thread, a byte is written to the pipe
5. Inside the thread function, the pipe is read (this step indicates context switch)
6. The timer is stopped after the pipe is read and the thread exits
7. The timer difference is given to an output file which is read by the Java program
8. Java program calculates the mean number of cpu cycles and the standard deviation from this output file and the value entered in the table below is the result of multiplying the mean with 0.4 ns (2.5 GHz processor)

The procedure for using blocking pipes as a method of timing context switches was suggested by Ousterhout's paper [15]. The paper mentioned passing a byte between the process (child and parent) or between threads and calculating the amount of time it takes.

Pipes and FIFOs (also known as named pipes) provide a unidirectional interprocess communication channel. A pipe has a *read* end and a *write end*. Data written to the write end of a pipe can be read from the read end of the pipe. [16]

Context switch time is defined here [7] as the time needed to save the state of one process and restore the state of another process

Base hardware performance	Software overhead estimation	Predicted operation time	Measured operation time
1 cpu cycle = 0.4 ns (2.5 GHz processor)	1. System call overheads for fork (i.e overhead of creating the child process) 2. Overhead of writing to the pipe and reading from it (since it is a blocking pipe)	Difficult to measure exactly	67 microseconds for a process context switch excluding system call overhead
	1. thread allocation overhead (pthread_create and pthread_join) 2. overhead of writing to the pipe and reading from it (since it is a blocking pipe)	1.34 microseconds (given time for process context switch)	1.25 microseconds for kernel thread context switch

The operation of measuring a context switch of a process includes the overhead of writing to and reading from a pipe. Since the pipe is a blocking communication method, the delay between a write and a successful read can vary depending upon the scheduling of the child process by the operating system.

But given the time for measuring process context switch we predicted the time of the thread switching by using the earlier ratio of 50:1 between user level processes and kernel level threads (since the overhead of reading and writing pipes will also decrease proportionately)
The predicted and measured time have been stated by subtracting the overhead of thread/process creation beforehand.

RAM access time

Methodology:

For measuring bandwidth of writing to memory, we have:

1. Considered max memory size to be 9MB
2. Created a circular linked list
3. Varying the size of the linked list for every iteration (1 million iterations for every size)
4. Traversed the linked list by accessing various memory locations
5. Measured the time for traversing the entire list
6. Generated a graph with x axis as log of Array size and y axis as the latency in nanoseconds

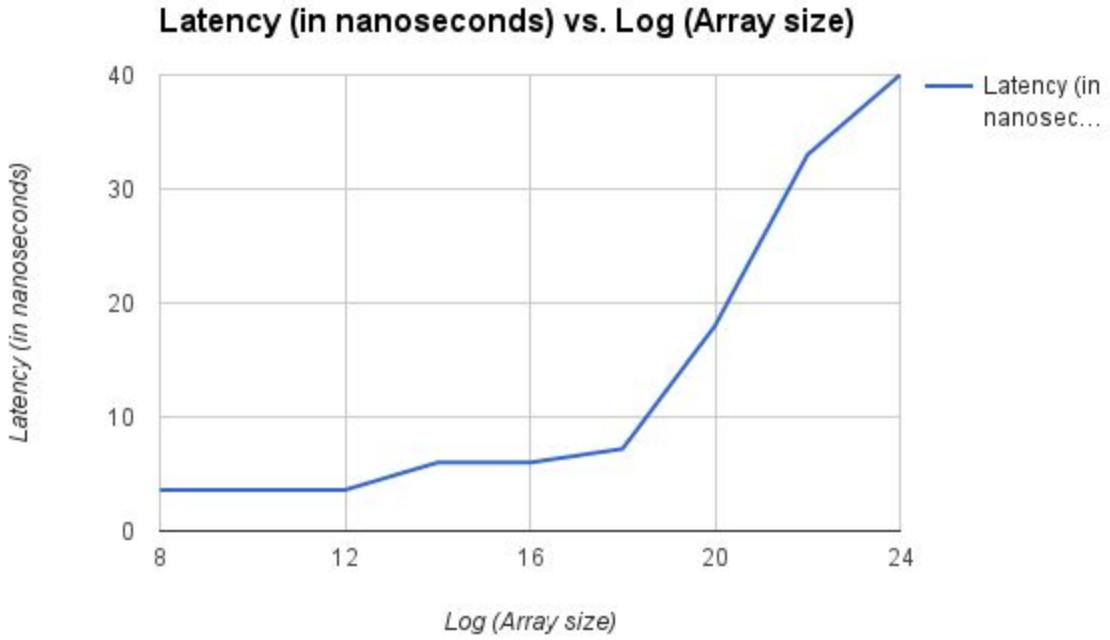
The method followed was based on the Back-to-back-load latency as mentioned in the lmbench paper [7]. This measurement was used as it is easily done through software. A linked list is traversed, by accessing various memory locations. The size of this linked list is varied over time. There arise, at specific time points, where the memory access time increases drastically. This spike in time occurs due to switch from lower level cache to a higher level cache. The graph indicates several plateaus and the point where each plateau ends and the line rises marks the end of that memory hierarchy.

The system that was used in this particular experiment was **different** than the ones for the other measurements. The reason was that clear distinction between caches was not visible in the former machine. The new system details are as stated below:

CPU details:

Processor:	AMD A10 8700p (4 cores)
Clock:	1.8GHz
L1 Cache:	32KB
L2 Cache:	4 MB

Since the file taken into consideration was 9MB, we expected to see latency spikes at the 32KB limit as well as the 4MB limit. The graph below plots the values of the latency measured as function of the log(Array Size). Logarithmic values were taken as the real values would be too big to be displayed effectively in the graph. As expected, the graph clearly demonstrates the spikes:



At log(Array Size) = 18 : L1 Cache ends (320Kb) (2^18 = 256KB nearest to 320KB)
At log(Array Size) = 22 : L2 Cache ends (4 MB) (2^22 = 4096KB)
Beyond log(Array Size) = 22 : Main memory (2^23 = 8MB nearest to 9MB)

From this reference [19], which shows the various latency periods for 2.5GHz processor (5-30 nanoseconds), we predicted the latency rates to be somewhere in the range of 5-50 nanoseconds for file size of 5 MB (since the processor used was 1.8GHz)

RAM Bandwidth

Methodology:

For measuring bandwidth of writing to memory, we have:

1. Considered the array size to be 4KB
2. Written to the array in 512 byte chunks
3. For every iteration of measurement we have covered 4096 bytes in 512 byte chunks
4. Reset the array after every iteration
5. Divided the timer difference obtained by the total number of loop iterations considered to get cpu cycles for 1 loop iteration for writing a 4096 bytes to memory

The method followed was based on the source cited below. Memory writing is measured by storing a value into an integer in an unrolled loop. The unrolling was done in order to avoid unnecessary overhead of a incrementing and storing another loop variable as it would have slowed the performance.

For measuring bandwidth of reading from memory, we have:

1. Considered the array size to be 4KB
2. Read from the array in 512 byte chunks
3. Added the array to a sum variable
4. For every iteration of measurement we have covered 4096 bytes in 512 byte chunks
5. Reset the array after every iteration
6. Divided the timer difference obtained by the total number of loop iterations considered to get cpu cycles for 1 loop iteration for writing a 4096 bytes to memory

The method followed was based on the lmbench paper[7]. Memory reading is measured by adding a series of integers in an unrolled loop. The unrolling was done in order to avoid unnecessary overhead of a incrementing and storing another loop variable as it would have slowed the performance. The addition operation was performed because of compiler optimizations which heavily optimize loops which perform no operations.

Base hardware performance	Software overhead estimation	Predicted operation time	Measured operation time
1 cpu cycle = 0.4 ns (2.5 GHz processor)	Reading time, loop overhead and integer sum overhead	71GB/s(56 cycles for reading 2KB)	83GB/s (48 cycles for reading 2KB)
	Reading time, loop overhead and integer assign overhead	35GB/s	55GB/s (72 cycles for writing 2KB)

The prediction for memory read was done by assuming that it would take 7 cycles (approx 3 ns) to access 256 bytes (64 indexes*4 bytes of integer array) of data (based on the values returned for previous experiment) and to access this 8 times in the loop gives 56 cycles.

The prediction for memory write was based on the statement that in Intel architectures, memory read time is much greater than memory write as while writing the memory is first read then written [7]. Hence, depending on this fact, we guessed that writing memory might be the half speed with which the memory can transfer bits to the CPU.

Page fault service time

Methodology:

For measuring page fault service time, we have:

1. Considered a file of size 5 MB
2. Created a new mapping of the file in the virtual memory using mmap()
3. Retrieved the values of each page mapped (this results in a page fault)
4. Timed the retrieval operation
5. Calculated the mean time for bringing one page into memory by dividing by the number of pages considered
6. Used munmap() to undo the mapping done and create a fresh mapping for the next iteration

mmap() creates a new mapping in the virtual address space of the calling process. The starting address for the new mapping is specified in *addr*. The *length* argument specifies the length of the mapping. [17]

munmap() system call deletes the mappings for the specified address range, and causes further references to addresses within the range to generate invalid memory references. [17]

msync() flushes changes made to the in-core copy of a file that was mapped into memory using mmap(2) back to the filesystem. Without use of this call, there is no guarantee that changes are written back before munmap(2) is called. [18]

Base hardware performance	Software overhead estimation	Predicted operation time	Measured operation time
1 cpu cycle = 0.4 ns (2.5 GHz processor)	The only significant overhead will be the overhead of fetching the page from the memory into the CPU	Prediction was not possible	250 cycles or 100 ns

The procedure takes input as 5 MB file as this file is large enough not to be stored entirely in the cache. Hence, when the new mapping is created through the map function, any and every fetch of the page results in a page fault. There are certain optimization strategies implemented like cache prefetching to increase the locality of reference so we have considered the mean of the timings in order to get the proper values.

The prediction was not possible as the only significant overhead incurred will be the access time for the page (in case of page fault) and that is precisely what is to be measured.

Summary

Operation	**Base Hardware Performance**	**Estimated Software Overhead**	**Predicted Time ***	**Measured Time ***
Measurement of time overhead	0.4 nanoseconds per cycle	Machine instruction overhead (5 ns)	5 nanoseconds	8 nanoseconds
Measurement of loop overhead	0.4 nanoseconds per cycle	Timing overhead and machine instruction overhead (	3.6 nanoseconds	0.8 nanoseconds
Procedure call	0.4 nanoseconds per cycle	Machine instruction overhead	9-16 nanoseconds	2-6 nanoseconds
System call	0.4 nanoseconds per cycle	Kernel privilege switch overhead	14 nanoseconds*	42 nanoseconds
Process creation	0.4 nanoseconds per cycle	System call overhead of fork+exec	200 microseconds	174 microseconds
Thread creation	0.4 nanoseconds per cycle	Kernel privilege switch overhead, Pthread create function overhead	2 microseconds	2 microseconds
Process context switch	0.4 nanoseconds per cycle	System call overhead, pipe creation overhead, blocking read write overhead of pipe	Difficult to predict	67 microseconds
Thread context	0.4	Kernel privilege	1.34	1.25

switch	nanoseconds per cycle	switch overhead, Pthread create function overhead, pipe creation overhead, blocking read write overhead of pipe	microseconds	microseconds
RAM access time	0.56 nanoseconds per cycle,	Timing overhead, machine instruction overhead	5-50 nanoseconds	5-40 nanoseconds
RAM read bandwidth	0.4 ns per cycle	Timing and machine instruction overhead, RAM access time overhead for large files (uncached)	71GB/s	83GB/s
RAM write bandwidth	0.4 ns per cycle	Timing and machine instruction overhead	35 GB/s	55 GB/s
Page fault service time	0.4 ns per cycle,	Page retrieval overhead (i.e RAM access time overhead)	63 nanoseconds	100 nanoseconds

* The reasons for prediction and measured values have been explained in their respective sections

References

1. Gabriele Paoloni: How to Benchmark Code Execution Times on Intel IA-32 and IA-64 Instruction Set Architectures (September 2010)
2. http://man7.org/linux/man-pages/man2/sched_getaffinity.2.html
3. https://en.wikipedia.org/wiki/Nice_(Unix)
4. http://stackoverflow.com/questions/22756092/what-does-it-mean-by-cold-cache-and-warm-cache-concept
5. www.strchr.com/performance_measurements_with_rdtsc
6. http://x86.renejeschke.de/html/file_module_x86_id_45.html
7. Larry McVoy and Carl Staelin, lmbench: Portable Tools for Performance Analysis, Proc. of USENIX Annual Technical Conference, January 1996
8. http://www.agner.org/optimize/instruction_tables.pdf
9. http://www.intel.com/../manuals/64-ia-32-architectures-optimization-manual.pdf
10. The Performance of µ-Kernel-Based Systems, Liedtke et al, 1997, http://os.inf.tu-dresden.de/pubs/sosp97/
11. http://man7.org/linux/man-pages/man2/fork.2.html
12. https://www.gnu.org/software/libc/manual/html_node/Executing-a-File.html
13. https://computing.llnl.gov/tutorials/pthreads/
14. https://www.usenix.org/legacy/event/usenix04/tech/general/full_papers/ruan/ruan_html/node13.html
15. John K. Ousterhout, Why Aren't Operating Systems Getting Faster as Fast as Hardware?, Proc. of USENIX Summer Conference, pp. 247-256, June 1990
16. https://linux.die.net/man/7/pipe
17. http://man7.org/linux/man-pages/man2/mmap.2.html
18. http://man7.org/linux/man-pages/man2/msync.2.html
19. http://www.extremetech.com/wp-content/uploads/2014/08/latency.png