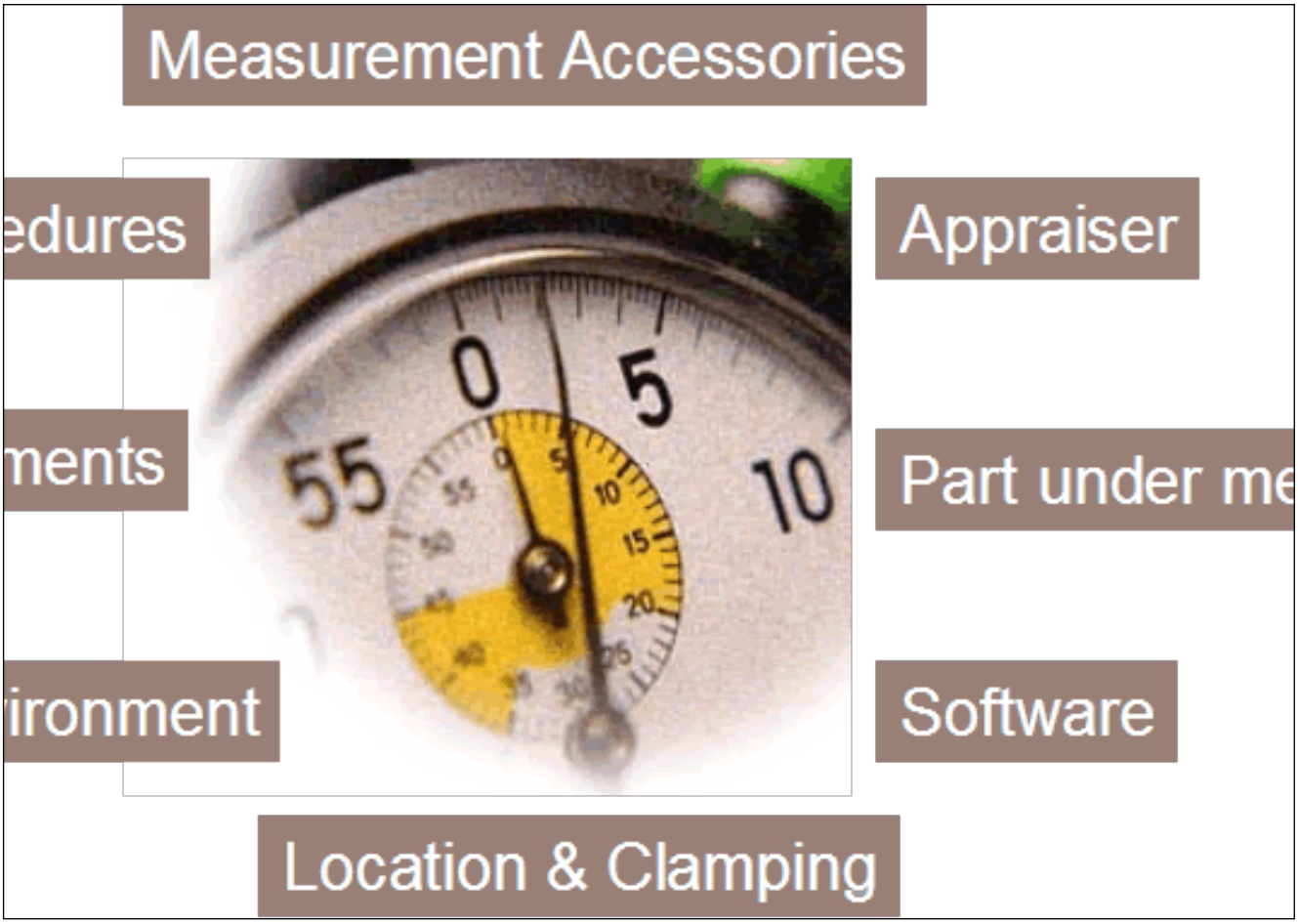
Systems Measurement

CSE 221, Fall '14

Kaushik Kalyanaraman
kkalyana@ucsd.edu

Priyanka Ganapathy
pganapat@ucsd.edu

November 12, 2014



Introduction

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

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

Most of these benchmarks have been implemented in **C++**, on **Mac OS X 10.10**, using **Apple LLVM version 6.0 (clang-600.0.51)**. Compiler optimizations in LLVM have been turned off using the flag -O0 and the system has been restricted to one active core, with HT disabled using the XCode Instruments.app.

Machine Description

Table 1: Hardware Information

Hardware Type	Description
Processor	Intel(R) Core(TM) i5-4258U@ 2.40GHz
Cores	2 Physical, 4 Logical
L1 I-Cache	32K 8-way set associative, 64-byte line size
L1 D-Cache	32K 8-way set associative, 64-byte line size
L2 Cache	256K 8-way set associative, 64-byte line size
Unified L3 Cache	3M 12-way set associative, 64-byte line size
Memory Bus Speed	1600 MHz
RAM	4096M Dual Channel DDR3 SDRAM 11-11-11-28
Disk Name	Apple SSD SD0128F (PCIe)
Disk Capacity	120.47GB
Disk Info	PCI, 256MB DDR DRAM buffer (Marvel-88SS9174 disk controller)
Network Card Speed	Apple Thunderbolt - Up to 20 Gb/s
Operating System	Mac OS X 10.10 (Yosemite)

CPU, Scheduling and OS Services

Measurement Overhead: For measuring the overhead of reading time, we use the mach_absolute_time() method mach/mach_time.h which provides the finest granularity of measuring time by providing the number of clock ticks since machine start. Two calls to mach_absolute_time() are initiated simultaneously, taking care to ensure that all other unnecessary applications have been shutdown, so that these two instructions are not reordered and are executed consecutively on the processor.

We call these two functions about **250,000** times in a loop and average the observations over all iterations in order to report a better statistical average overhead of reading time. Going by the implementation of mach_absolute_time() here [1], it is extremely fast and provides nanosecond level granularity. We estimate the overhead of retrieving time to be about 30 cycles, which is the time required to read the RDTSC register, perform the comparison and

the arithmetic and return the result and the actual value found is about 29.162 cycles on average.

The loop overhead involves a slightly complicated operation: even after turning off compiler optimizations, loops are being optimized either by being unrolled, by being eliminated as dead code by the hardware and/or by a two-level predictor with a 32-bit GHB on the Intel core i5. Running the same loop 250,000 times provides an average of 2 cycles per loop, which is logically not possible since the lower bound of the overhead of reading time is about 30 cycles. Instead, we observe a reasonable loop overhead to be of about 40 cycles (one initialization instruction, no loop body, one increment instruction and then one JNZ instruction) and we run a loop only once but run the program multiple times to obtain a reasonable estimate. We observe the loop overhead to be about 42.127 cycles.

Procedure Call Overhead: We theorize the cost of a procedure call to slightly increase as more arguments are added, because the CPU needs to use additional instructions in-order to move more values into the procedure and the corresponding activation record size of the procedure also increases as the number of arguments increases. Eight procedures have been created, each with an increasing number of 32-bit int arguments, starting with zero all the way up to seven with only a return statement in the body. We estimate the cost of a procedure call with zero arguments to be about 25 cycles which would include the cost of creating an activation record for the procedure, pushing it on to the stack, saving the address of the instruction after the function return into a temporary register, modifying the PC to be the first instruction in the procedure, loading the next address again into the PC and destroying the procedure's activation record. The cost of adding an additional argument would be the order of a cycle, since most of the above mentioned procedures are the same, with only the inclusion of one or two additional commands to handle the extra argument.

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

And an average increase of 1 cycle per argument.

System Call Overhead: We use the stat() system call to report the overhead of running a system call. Since idempotent system calls such as getpid() are cached by glibc, micro-benchmarking the system call overhead by running the same system call multiple times gives us incorrect results because only the first call is actually a trap into the OS, subsequent calls are just fetched from cache.

We estimate the cost of a system call to be much more than a procedure call because a system call is an expensive operation involving the following operations [2]:
1. Privilege Context Switch
2. A trap into the OS interrupt vector
3. A kernel supervisor routine now takes control and determines what system call has been executed, and if its arguments are legal.

Having taken these considerations into account, we estimate the cost of a (non-ultra fast [3]) system call to be about 1,000 cycles and our observations indicate that the cost of the system call stat() is about 1062.25 cycles.

Task Creation Time: We estimate the creation time of a thread to be at least 20x faster than that of a process because threads are lightweight process that share the same address space as that of the parent process but the creation of a new process requires setting up of its own address space, virtual memory etc. We use the fork() system call to create a new child process with the additional overhead of copying the parent processes's address space into the child process. We use pthread_create() for creating new threads and time is measured before creation and after pthread_join().

The estimates are in line with observed values. The process creation time is about 1742520 cycles and thread creation time is about 95560 cycles which is about a factor of 25x.

Context Switch Time: Context switch time, as described in [4] involves a lot of factors, such as cache pollution time, working set size, the resources held by a process etc. We do estimate the time for a thread context switch to be less than a process context switch because of the inherent light-weightness of a thread combined with the fact that it usually shares resources with other threads within the same process. About a 30% difference in the context switch times is a reasonable estimate as mentioned in [4] but we observe almost a 10x difference between the two.

We use fork() to create a new process, and we measure the process context switch time, averaged over 500 switches using a blocking pipe read. Data is written into one end of the pipe in the child process and the parent process blocks on read until the data is written into the pipe and the CPU is yielded to the parent, during which the child blocks on read from the parent. Once the CPU is yielded to the parent, the current clock ticks are measured and a value is written back into the pipe which unblocks the child. This continues for about 500 iterations, resulting in 1000 context switches.

We use pthread_create() to create a new thread and sched_yield() to switch between the two threads. Sched_yield() yields the CPU and the current thread is put at the back of the thread scheduling pool. Calling sched_yield() on the other thread once again brings back control to the original thread. Both threads were run on a single core, with multithreading disabled. These observations were also averaged over 500 iterations i.e., 1000 thread context switches.

The observed values are:
Process Context Switch - **3930.78 cycles**
Thread Context Switch - **322.482 cycles**

Memory Operations

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

The pseudo-code used for our measurement was:

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

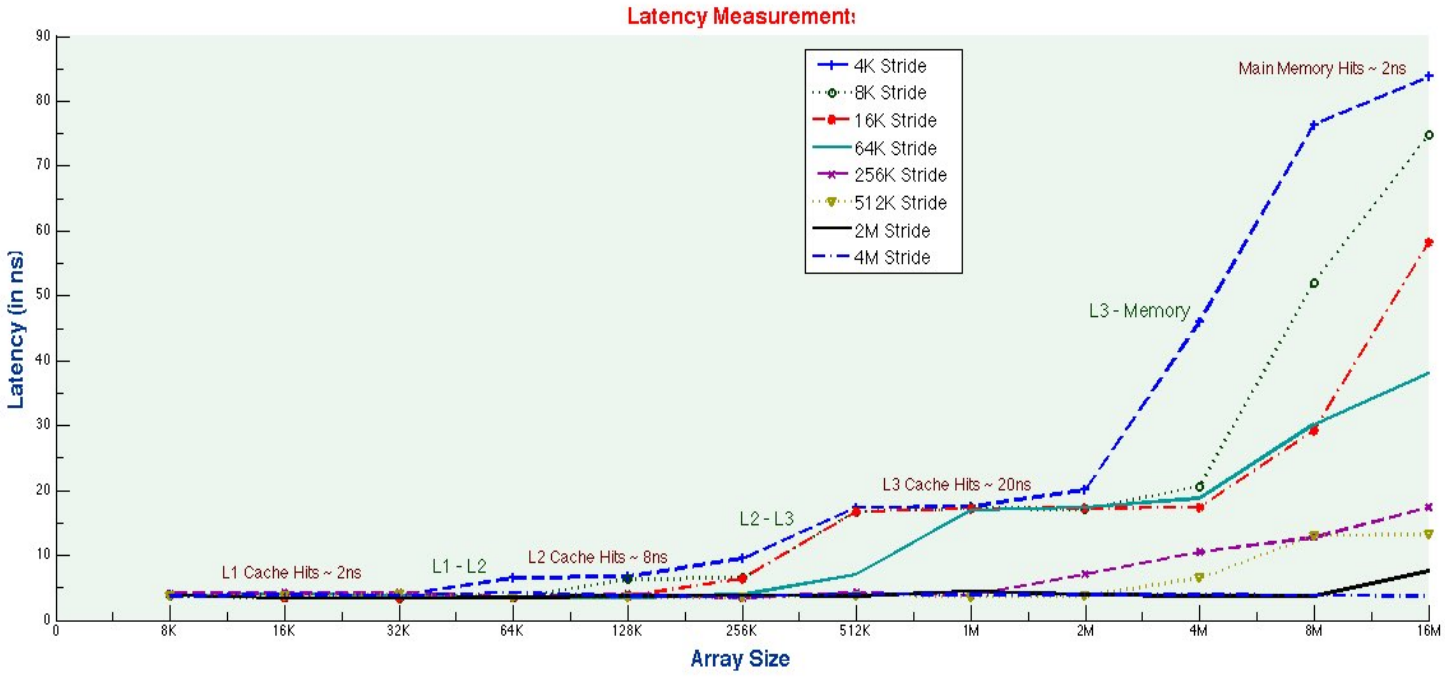
and then averaged all readings over NUM_SAMPLES to obtain the final result.
reinterpret_case is just the C++ style version of C style casting and hence has no overhead.

In fact, the operation int *p = reinterpret_cast<int*>('a') translates into the two assembly instructions,

movabsq $97, %rcx

movq %rcx, -16(%rbp)

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



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

An anomaly we notice in the graph is that for larger strides, the latency penalties are not as high as the smaller strides. We are wrapping around the array while striding and this could be a possible reason for lesser and lesser latency. For instance, the 4M stride almost always is

an L1 cache hit because a 4M stride wraps around the size of the array to access elements recently accessed, once again.

Intel's spec sheet for the Haswell micro architecture [8], reports that the L1 cache latency is about 4 cycles, the L2 cache latency is about 11 cycles which is very close to the observed values. The software visible cache latency is obviously dependent on access patterns and a lot of other factors and so we cannot micro benchmark precisely the latencies of different memory levels.

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

RAM Bandwidth: RAM bandwidth goes up to a maximum write bandwidth of 25.6 GB/s for a dual channel, 1600MHz DDR3 SDRAM [7] and a maximum read bandwidth of 51.2 GB/s for the same setup [8]. Memory bandwidth is computed by the product of the base DRAM clock, the number of data transfers per clock, the bus width and the number of interfaces [9].

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

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

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

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

The speed of the memory controller and DRAM page conflicts also play an important role in limiting the memory bandwidth. Since our bandwidth experiment was conducted on one thread, in a single process, it wasn't as efficient as it would be if multiple threads accessed

memory at the same time (which would be able to obtain the maximum possible throughput from the memory.)

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

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

and we observe a **read** bandwidth of about **22.213 GB/s.** Surprisingly, this is less than half the theoretical maximum read bandwidth of 51.2 GB/s and we surmise that the same problems affecting write bandwidth affects read bandwidths also i.e., absence of multithreaded sequential and random access reads, no guarantee of full channel bandwidth for our benchmark etc.

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

Our assembly instructions perform a uniform stride > 64B and so does not come under the hardware prefetcher. We disabled the adjacent cache line prefetch by setting **bit 19 of register 1A0h** on our machine. After this, we have taken reasonable assumptions as to the absence of the influence of the cache prefetcher on our readings. Another point to note is that having a small number of assembly instructions (1/2) ensures that only the first OPCODE fetch is a miss in the L1 instruction cache, subsequent fetches have a high probability of being a cache hit in the L1 I - cache or at some other higher level cache and so the overhead of the processor having to fetch the instruction OPCODE from memory is reduced too.

Page Fault Service Time: We measure the page fault service time using mmap to map a 1GB file into virtual memory using the command

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

where the file is divided into chunks of size chunkSize and and we load each chunk into memory, from the specified offset. By dereferencing the memory object returned by mmap and striding across this object by a size > pagesize (4K), we ensure that subsequent accesses are all page faults and the VM manager brings the pages from disk onto the main memory as well as the TLB. On the initial runs through the program, we calculate the average page fault service time to be about 16618.9 ns. But on going through frequent, multiple runs of the

program, the service time reduces significantly, to about 1445.86 ns and we postulate that this is due to two reasons:

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

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

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

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

References

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

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

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

[4] http://blog.tsunanet.net/2010/11/how-long-does-it-take-to-make-context.html

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

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

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

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

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

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

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

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

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

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