Homework 1

CS6354 F2016 Computer Architecture
William Young
26 September 2016

1 System

OS: Ubuntu 16.04.1 (32-bit) (2-cores)
running atop macOS Sierra in VMWare Fusion v8.0.0

Memory: 4042MB

Chip

```
Package L#0 + L4 L#0 (128MB) + L3 L#0 (6144KB) + L2 L#0 (256KB) + L1d L#0
(32KB) + L1i L#0 (32KB) + Core L#0 + PU L#0 (P#0)

Package L#1 + L4 L#1 (128MB) + L3 L#1 (6144KB) + L2 L#1 (256KB)
+ L1d L#1 (32KB) + L1i L#1 (32KB) + Core L#1 + PU L#1 (P#1)
```

(Data obtained via hwloc v1.11.4)

2 Overview and Methodology

This project consists of a single binary executable, *benchmark*, which performs a series of microbenchmark tests in order to examine and record the characteristics of the memory hierarchy on a target system. Two fundamental algorithms form the underlying framework of the full suite of microbenchmarks:

- Memory latency: *Pointer-chasing*
  - Can be used to determine the relative difference in access latency when stride and memory usage is varied. Pointer chasing to some extent negates the performance optimizations introduced by features such as prefetching. As stride and array size exceed cache levels, cache misses are generated, adding noticeable latency when iterated over many times (greater than 1,000,000).

```
PSEUDOCODE:

while varying stride and array size:
       c = (char**) *c; // pointer chase
```

- Memory bandwidth: *Large reads and writes*
  - Useful for determining how much data can be read and written in a fixed interval of time. This approach requires that the target arrays be fully initialized prior to loop execution. This project utilized arrays filled with custom-defined structs of size 32-bytes, *bigblock*, to force more data to be transferred per operation. The

inspiration for the design of this algorithm comes from John McCalpin (University of Virginia).

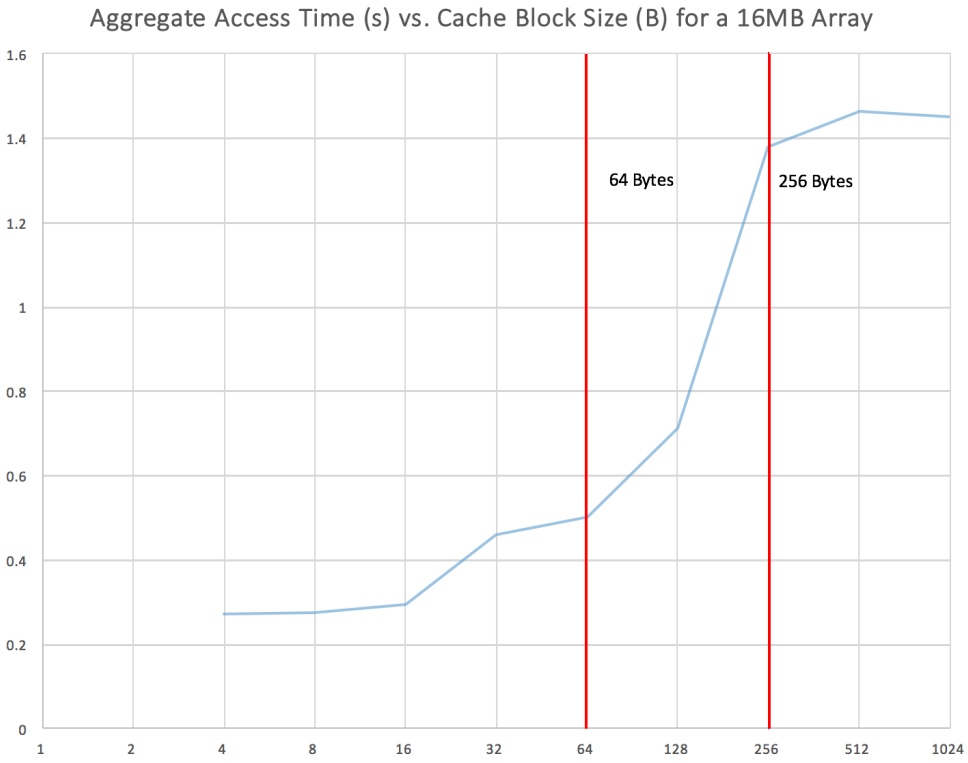
```
PSEUDOCODE:

large arrays a, b
for(i = 0; i < len; i++)
       a[i] = b[i]
```

These fundamental algorithms each allow a number of related tests to be performed in order to benchmark multiple architectural characteristics.

3 Cache Block Size

To benchmark cache block (line) size, we perform a cache latency test on an arbitrary 16-megabyte array with varying stride lengths, beginning at 4 bytes and ending at 1024 bytes. A noticeable increase in latency immediately follows the stride length equal in size to the cache block, as each stride will incur a cache line miss. In the figure below, it appears that latency begins to increase 64 bytes before plateauing at 256 bytes.

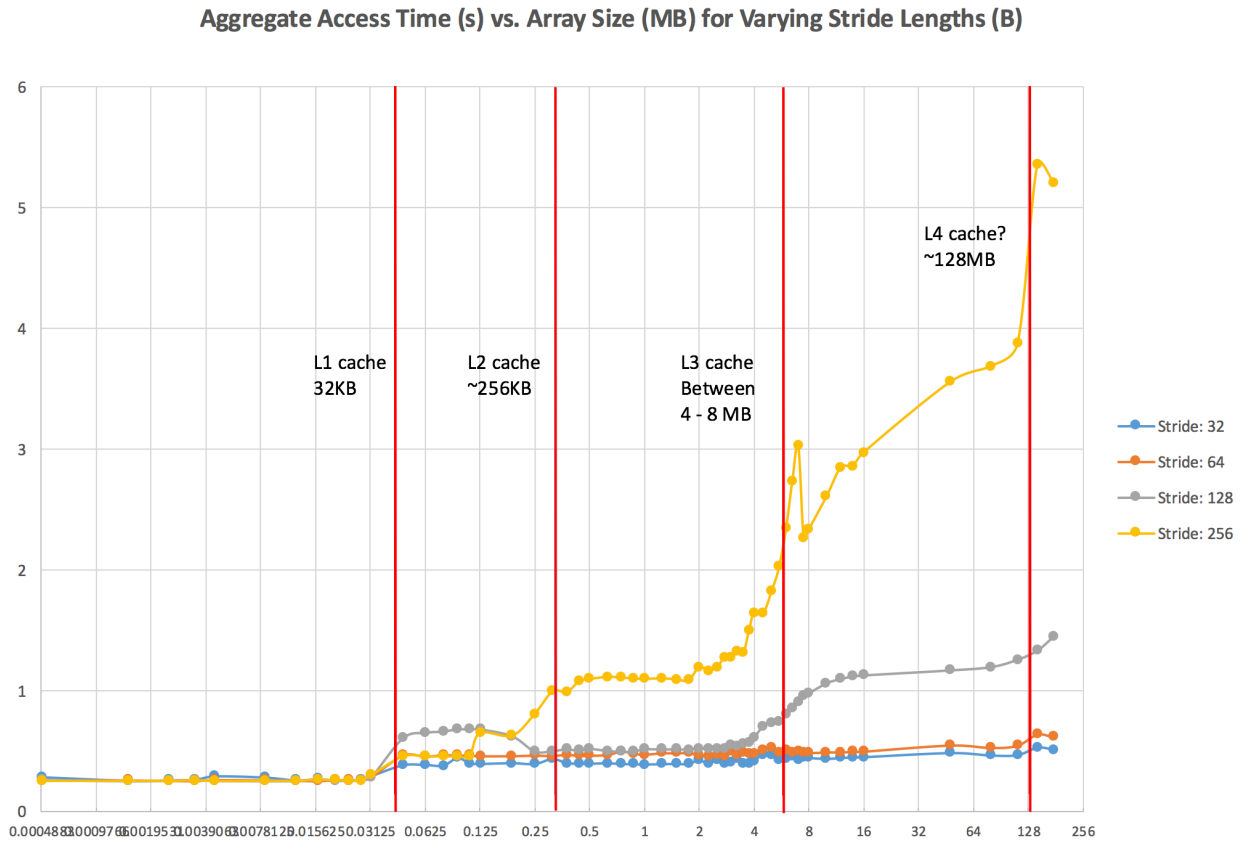
Measured Size: approx. 64 bytes
Actual Size: 64 bytes



4 Data Cache Size

To benchmark data cache sizes, we perform a cache latency test using arrays and strides of varying lengths. As array size and stride length both increase, we should observe a noticeable increase in latency when cache boundaries are crossed due to misses. These performance degradations should correspond with each successive data cache level. The existence of TLBs and victim caches, in addition to the unpredictable behavior of the L3 and L4 caches on this system, all have the potential to skew results.

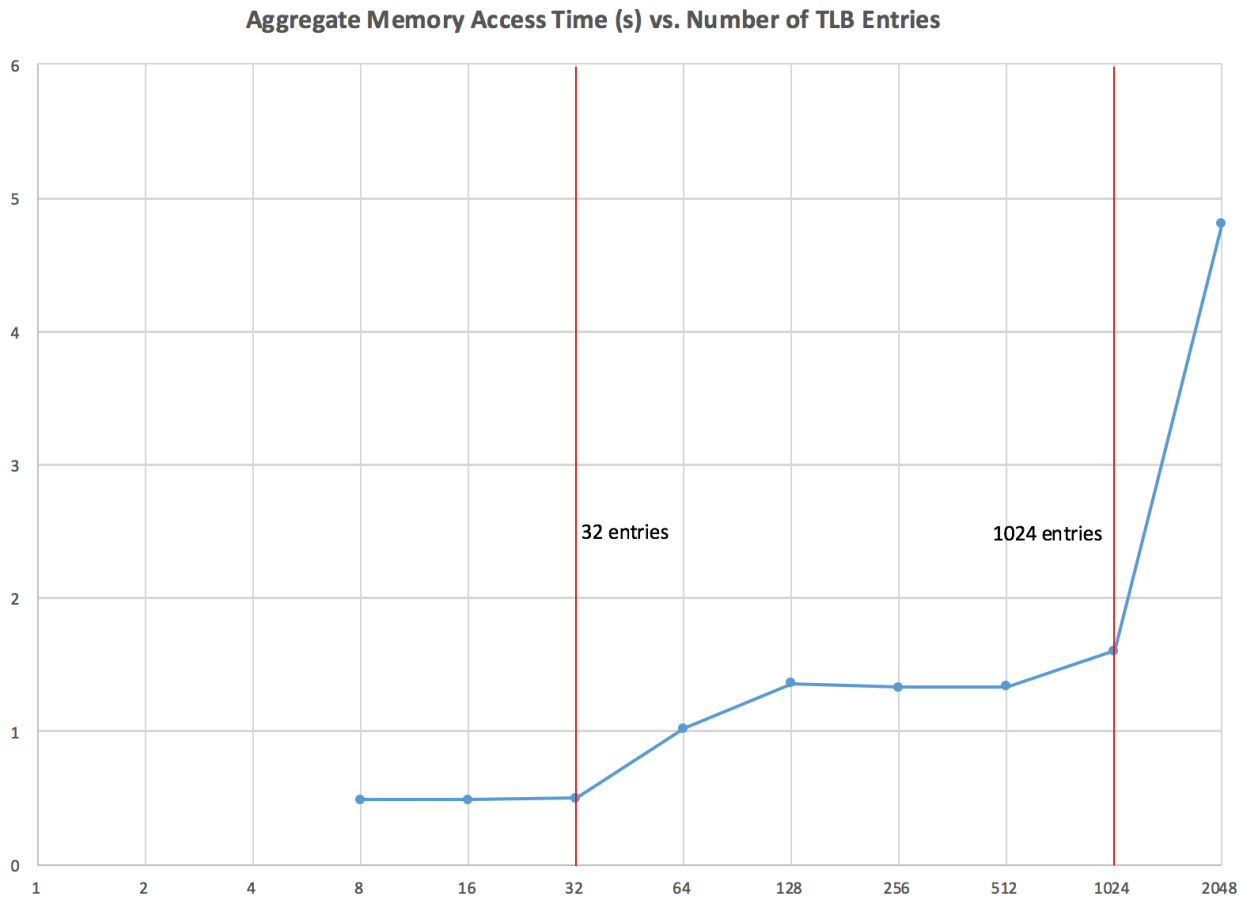
Measured Size:	L1d:	32 KB
	L2:	256 KB
	L3:	4-8 MB
	L4:	Indeterminate (between 64 MB and 256 MB)
		Hard to distinguish from main memory
Actual Size:	L1d:	32 KB
	L2:	256 KB
	L3:	6 MB
	L4:	128 MB



5 Data TLB Size

To benchmark data TLB size, we perform a cache latency test using arrays and strides of particular lengths; namely, each stride is a constant 4 kilobytes (page size, a priori knowledge) while each array size is equal to the page size times the number of estimated TLB entries, plus one extra page. We force the estimated TLB entry count to vary in the test, beginning with 8 entries and ending with 2048 entries. When we access a page not currently stored in the TLB, we will generate a TLB miss and incur a large increase in latency. The large increase at 1024 entries could be due to a number of reasons, though accessing main memory seems to be the most likely culprit.

Measured Size:	L1:	32 entries
	L2:	1024 entries (potentially)
Actual Size:	L1:	64 entries (macOS Sierra)
		32 entries (internal to Linux)
	L2:	1024 entries (macOS Sierra)
		Not visible to Linux



6 Memory Throughput

To benchmark main memory throughput, two 64-megabyte arrays of type *bigblock* are allocated and initialized in memory.

```
typedef struct bigblock {
       double a;
       double b;
       double c;
       double d;
} bigblock;
bigblock* a = malloc(len * sizeof(bigblock));
bigblock* b = malloc(len * sizeof(bigblock));
```

Main Memory - Sequential

For sequential main memory throughput, while iterating over each *bigblock* item in the two arrays, an item is read and copied between arrays according to the following:

```
for(i = 0; i < len; i++)
       x[i] = y[i];
```

Measured: 15,233 MB/s
Actual: approx. 19,500 MB/s (from *stream* benchmark)

Main Memory - Random

For random throughput, the loop iterates the same number of times, but the indices for the read and copy operations are selected at random. Measures were taken to reduce the impact of the costly call to `rand()`, but the random throughput remains substantially lower than sequential throughput. This is most likely due to implementation error in the project.

```
for(i = 0; i < len; i++) {
       r = rand();
       x[(i * r) & (len-1)] = y[(i * r) & (len-1)];
}
```

Measured: 1,218 MB/s
Actual: (no comparison available)

Main Memory – Multicore

To obtain multicore throughput, the technique used to obtain single-core sequential throughput is repackaged as a *pthread*-compliant *void** function and tasked to two independent threads. Each thread should execute the throughput test more or less simultaneously, though there is a wide margin of error. This method could be fine-tuned to obtain more precision. Performance results showed an average throughput nearly double that of the single-core throughput.

```
pthread_t thread1, thread2;
pthread_create(&thread1, NULL, bw_helper, (void*)r);
pthread_create(&thread2, NULL, bw_helper, (void*)r);
```

Measured: 29,780 MB/s

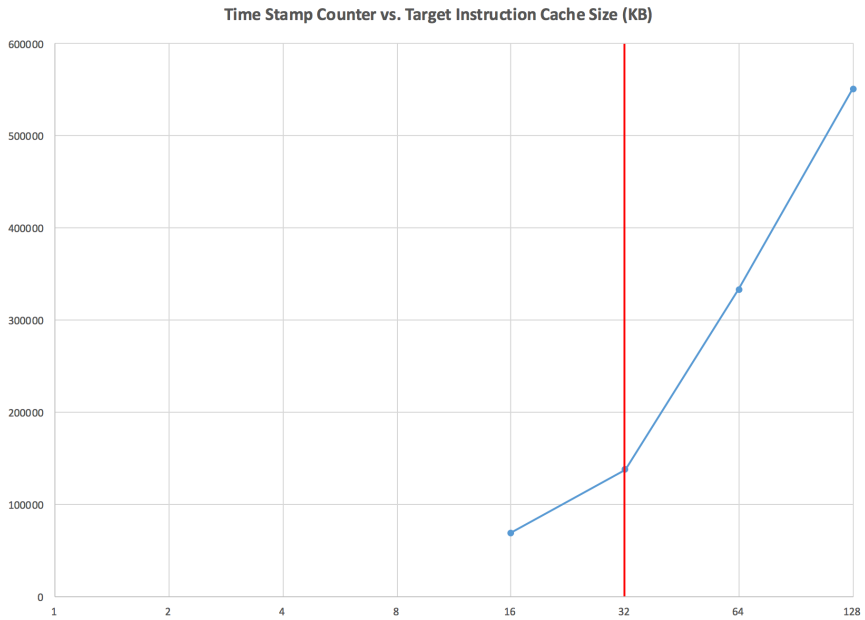
Cache

For cache throughput, the process is identical to sequential main memory throughput except with array sizes corresponding to the size of each data cache.

Measured: L1: 19,703 MB/s
L2: 19,308 MB/s
L3: 14,399 MB/s

7 Instruction Cache Size

To benchmark instruction cache size, the CPU is fed instructions in streams of 2048 no-operations, ranging from 2 to 128 kilobytes of total instructions. This is one benchmark that I found difficult to implement properly. I believe that no-operations must somehow be filtered-out by the compiler, or at the very least, the rate at which I send instructions to the CPU is not nearly as fast as the rate at which they are handled. Though the graph below shows a sharp uptick in latency at 32 kilobytes, I believe this is erroneous and more the result of timer variations than actual benchmark success.

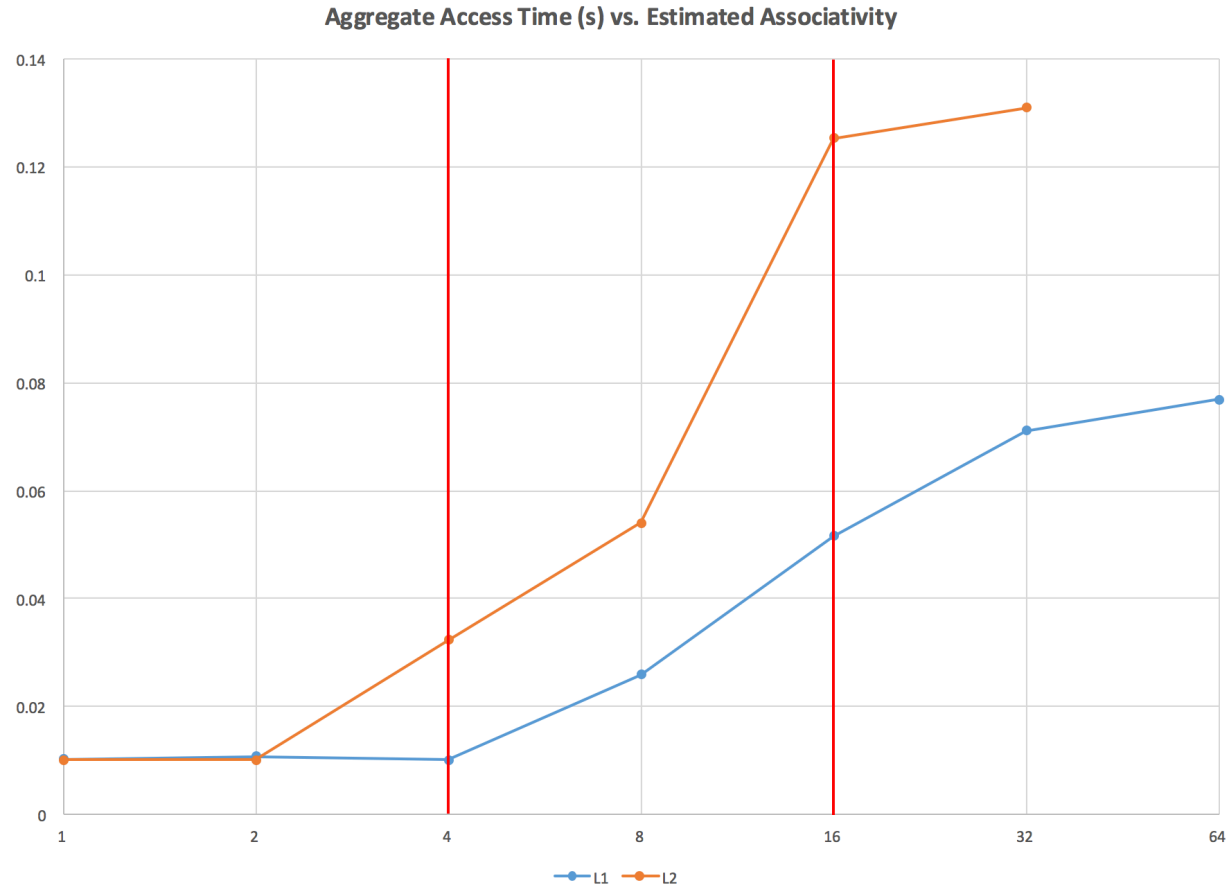


Measured: Inconclusive
Actual: 32 KB

8 Data Cache Associativity

To benchmark data cache associativity, my general strategy consisted of accessing n unique array elements repeatedly with a stride equal to the size of the cache in question. My intuition was that as the number of unique elements increases beyond the associativity of each cache's TLB, there should be an observable increase in latency. This approach is limited by a few factors. Primarily, it doesn't take into account virtual-to-physical mapping, which skews results (only virtual memory is adjacent).

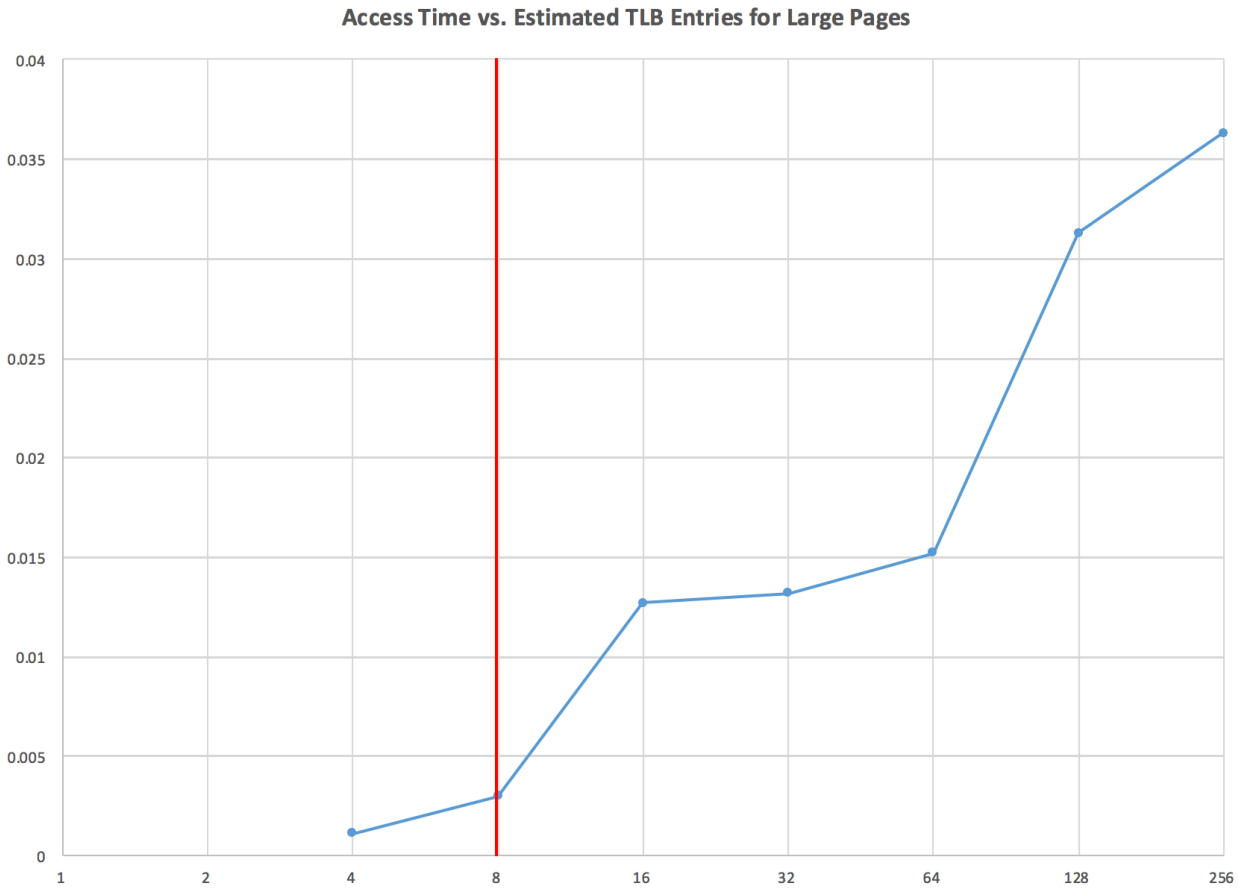
Measured:	L1:	Inconclusive
	L2:	Inconclusive (both seem to fall between 4 and 16)
Actual:	L1:	8-way
	L2:	8-way



9 Data TLB Size with Large Pages

To benchmark data TLB size when using large pages, the process is similar to benchmarking the data TLB size when using regular pages—the only thing that changes is stride (from 4KB to 4MB). A large increase in latency is observed at 8 entries and at 64 entries. Once again, a potential problem plaguing this test is access to main memory.

Measured: 8 entries (inconclusive)
Actual: 8 entries



10 Latency

To benchmark cache and memory latency, pointer chasing was conducted on three separate arrays, with each array size corresponding to a data cache size. The stride was held constant at 4 kilobytes (page size). As array size increases and more elements are accessed, we expect the performance decrease to reflect the increasing access latency to access an element in each cache level. Potential performance errors may have been present due to the use of loops to perform the chasing; overhead is exacerbated by fine granularity measurements. The latencies gathered from the results more or less reflect the published access values.

Cache Latencies

Measured:	L1:	1.4248 ns
	L2:	9.2217 ns
	L3:	17.6435 ns

Main Memory Latency

Measured:	98.7055 ns

11 Incomplete Benchmarks

Despite my best effort to find illuminating information, I was unable to devise methods to measure data TLB associativity and the existence of prefetching. The techniques to do so were beyond the scope of my abilities on and familiarity with low-level performance benchmarking.

12 References

POINTER CHASING - http://faculty.smcm.edu/acjamieson/s10/COSC251Lab2ish.pdf
POINTER CHASING -
http://www.ece.umd.edu/courses/enee759h.S2003/lectures/LabDemo1.pdf
CACHE SIZE; POINTER CHASING - http://igoro.com/archive/gallery-of-processor-cache-effects/
CACHE SIZE - http://www.cplusplus.com/forum/general/35557/
CACHE SIZE - http://stackoverflow.com/questions/12675092/writing-a-program-to-get-l1-cache-line-size
BANDWIDTH - https://www.cs.virginia.edu/stream/FTP/Code/
MISCELLANEOUS – LMBENCH test suite (lmbench.sourceforge.net)
PERFORMANCE - http://www.7-cpu.com/cpu/Haswell.html