Determining L2 Cache Parameters

Methodology:

To determine the L2 cache, dynamically allocate a huge integer array too big to fit in L2 cache. Try to access elements in the array in the steps of 2^n bytes and benchmark the results. We can determine the cache size by observing significant spike in the results.
In order to access elements in the array and to ensure that the compiler doesn't try to optimize, we used a *volatile* variable sum and we tried to sum up the array accessed.
The *'volatile'* keyword ensures no compiler optimization.

Cache line size:
Since cache line size is the amount of data that gets transferred on a cache miss, inorder to determine *L2 cache size* , we need to first determine *Cache line size* and determine Cache size as a multiple of Cache line size.
The main idea to determine the cache size, is similar to how we determine cache line size.

1. Dynamically allocate a huge integer array too big to fit in cache.
2. Initialize it sequentially so that the memory address is mapped.
3. Try to access the array in the steps of 2^n bytes and benchmark the time spent per array access in usec. Since the cache line size is generally small (less than 128 Bytes), we repeat step (3) from range of 4B to 128 B and we benchmark the results.
4. Observe the largest spike in time as Cache line size.

Note: For each iteration, we ran same number of array accesses for uniformity.

Cache Size:
For determining cache size, we determine it as a multiple of cache line size.

1. Dynamically allocate a huge integer array too big to fit in cache.
2. Initialize it sequentially so that the memory address is mapped.
3. Try to access the array in the steps of 2^n bytes and benchmark the time spent per array access in usec. We repeat step (3) from range of multiple of 1 Cache line size to 262144 times cache line size and we benchmark the results.
4. Observe the largest spike in time as Cache size.

Note: For each iteration, we ran same number of array accesses for uniformity.

Cache Miss Penalty:
For determining cache miss penalty, we need to access array elements in such a way that there is a cache miss every time. In order to ensure this, we always jump 2 times the cache size address space.

1. Dynamically allocate a huge integer array too big to fit in cache.
2. Initialize it sequentially so that the memory address is mapped.
3. Try to access the array in the steps of 2 times the cache size address space so that there is a miss every time.
4. Return the (total time)/(number of array accesses) as the average cache miss penalty.

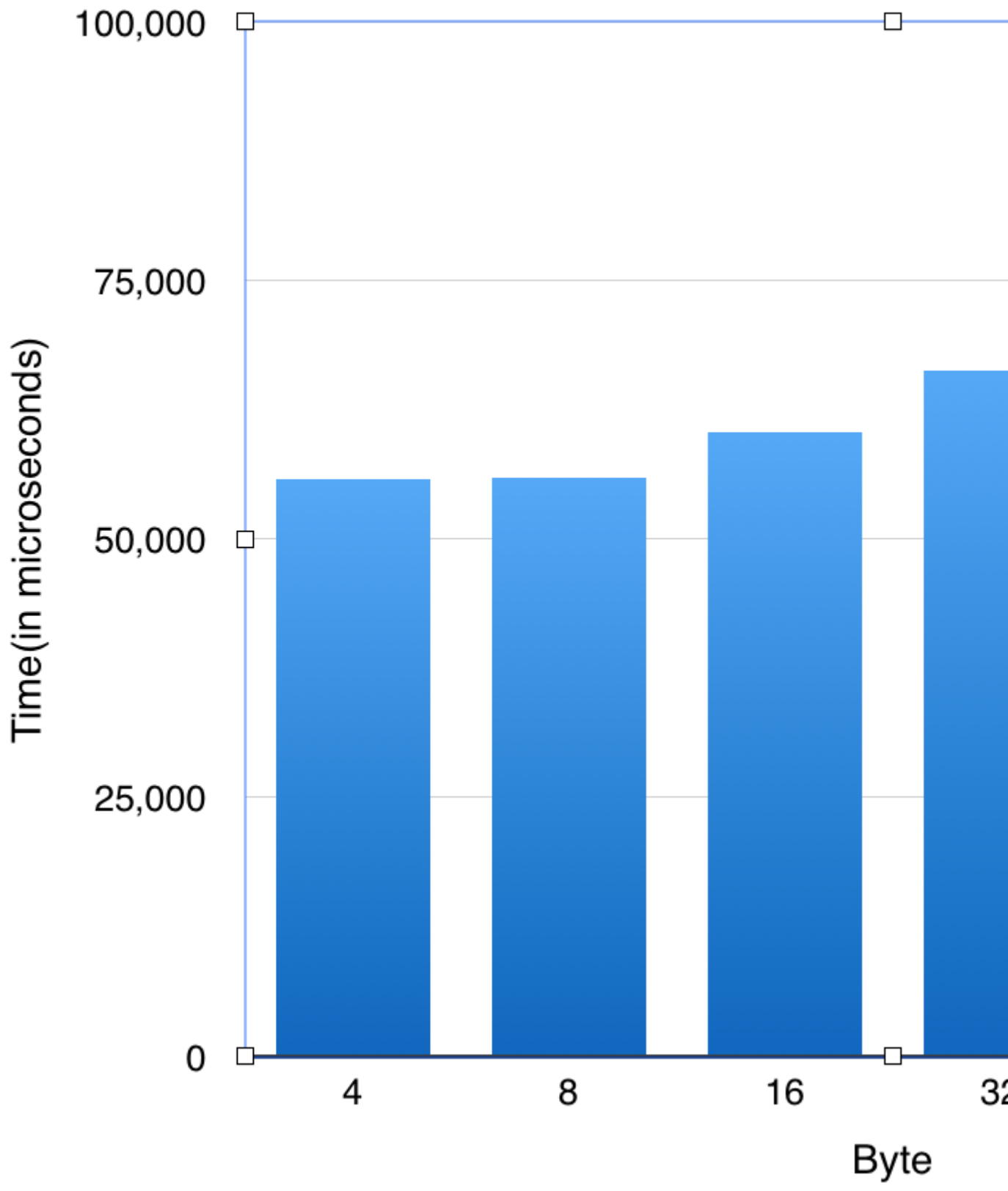
Results:

We tested our results on the below machine:
CPU: AMD A10-4655M APU with Radeon(tm) HD Graphics
L1d cache: 16K
L1i cache: 64K
L2 cache: 2048K

Cache Line size:
For Cache Line size, spike was observed after 32 B. Hence that is the cache line size for the machine.

Byte	Time(in microseconds)
4	55,811

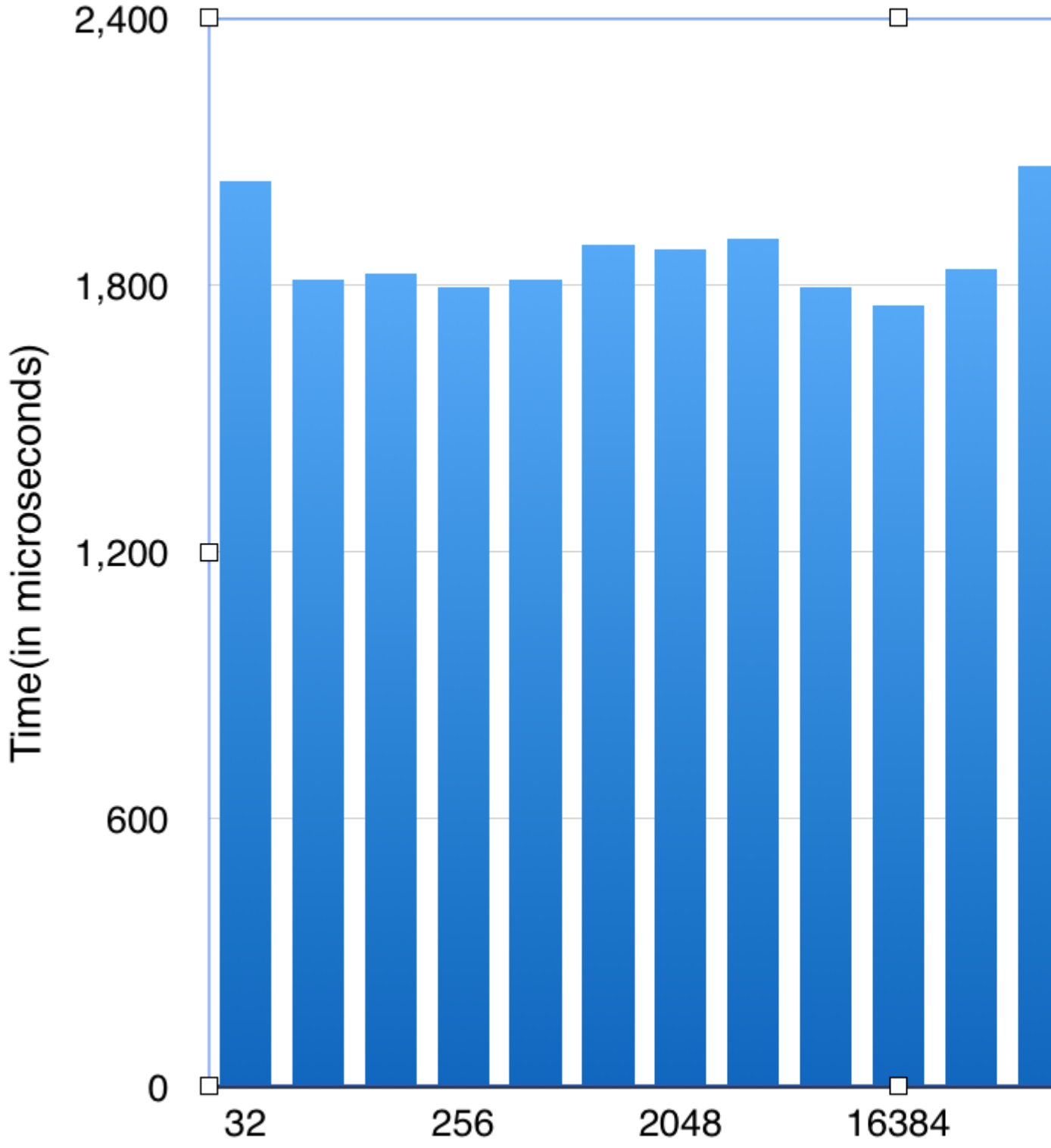
8	55,912
16	60,226
32	66,294
64	92,474
128	94,063



Cache Size:

For Cache size, spike was observed after 2048 KB. Hence that is the cache size for the machine.

Bytes	Time(in microseconds)
32	2,031
64	1,811
128	1,824
256	1,793
512	1,812
1024	1,890
2048	1,878
4096	1,902
8192	1,794
16384	1,753
32768	1,835
65536	2,068
131072	1,771
262144	1,678
524288	1,674
1048576	1,677
2097152	1,666
4194304	2,391
8388608	2,375



Final Result
Cache Block/Line Size: 32 B
Cache Size: 2048 KB
Cache Miss Penalty: 0.156000 us

Note: Our program sometimes even detected Cache size as 64K as L1i cache size.

We even tested the code on MacBookPro 11,1 with Intel Core i5 2.4 GHz and iLABs machine.

Code:

analyzecache.c

1. float analyze_cache_line_size(int* a, int cache_line_multiplier)

This method returns the time in microseconds. a is the reference to the array allocated (following the methodology defined above). *Cache_line_multiplier* is the assumed cache line size iteration we use to benchmark the result.

2. float analyze_cache_size(int* a, int cache_multiplier, int cache_line_size)

This method returns the time in microseconds. a is the reference to the array allocated (following the methodology defined above). *Cache_multiplier* is the assumed cache line size iteration we use to benchmark the result. *Cache_line_size* is line size of the cache we obtained as result from the *analyze_cache_line_size*().

3. float calculate_cache_miss_penalty(int* a, int cache_size)

This method returns the time in microseconds. a is the reference to the array allocated (following the methodology defined above). *Cache_size* is size of the cache we obtained as result from the *analyze_cache_size*().

4. void resetArray(int* a)

Initializes all the elements of array a with value 1.

5. int main(int argc, char *argv[])

This function main first allocates an array a and initializes value to each element using *resetArray()* runs a loop of step size 2^n and passes it as an input to the analyze_cache_line_size() and henceforth calls *analyze_cache_size*() and *calculate_cache_miss_penalty*(). This method determines the spikes and prints the results.

analyzecache.h

no code changes