CS211Spring 2018
Prof. Abhishek Bhattacharjee
Programming Assignment 5: Analyzing Caches Performance
Due: 11:55 PM, May 5th, 18
Jeeho Ahn, CS211

Part 1:

The goal of this assignment is to analyze cache performance and creating a most optimizing code for matrix multiplication. In a sense of reducing the miss rate on memory, it should be carefully built by checking how the memory is reviewed through the cache memory. By given the code, the result is shown on table 1 and the time taken to finish the code with sample1.txt is about 35 seconds.

Performance counter stats for original matrix multiplication		
cache-misses	756,825,190	(34.47%)
L1-dcache-loads	50,559,218,136	(49.99%)
L1-dcache-load-misses	11,495,216,243	# 22.74% of all L1-dcache hits (49.99%)
L1-dcache-stores	9,815,784,917	(50.00%)
L1-dcache-store-misses	<not supported>	
LLC-loads	3,470,810,999	(50.04%)
LLC-load-misses	250,394,549	# 7.21% of all LL-cache hits (50.01%)
LLC-stores	4,653,385	(24.99%)
LLC-store-misses	1,629,975	(24.99%)

Table 1

[Loop Fusion]

One way to optimize the program is by combining multiple loops into one. In the matrix multiplication, it can be done by bringing two initializations of matrix into one since the matrix sizes are the same. This method is called loop fusion. If loops are separated the program should walk through the cache again so it will contain double of the miss rate that the first loop had so it improves temporal locality. If loop fusion is possible, it should give better performance if loops are combined.

Performance counter stats for original matrix multiplication		
cache-misses	756,825,190	(37.52%)
L1-dcache-loads	50,522,294,529	(50.02%)

L1-dcache-load-misses	13,303,449,979	# 26.33% of all L1-dcache hits (50.00%)
L1-dcache-stores	9,790,232,805	(49.99%)
L1-dcache-store-misses	<not supported>	
LLC-loads	4,685,708,591	(49.99%)
LLC-load-misses	252,929,580	# 5.21% of all LL-cache hits (49.99%)
LLC-stores	3,395,743	(25.01%)
LLC-store-misses	1,710,287	(25.02%)

Table 2

The change from the original matrix was initializing two matrices together. The overall result did not improve much yet the run time decrease about one second compared to the original matrix program.

[Loop Interchange]

Another way to decrease the miss rate is by interchanging loops (loop interchange). This method is used because repeated referenced to variables will continuously result in hit in cache until it reaches the size of the byte. According to the research, it is found that C arrays are allocated in row-major order. Meaning each row is in contiguous in memory locations. For example, when a matrix is accessed through columns in one row, a[0][i], the cache accesses are reviewed in successive manner. Therefore, having cold miss in an empty block, it will have 1(miss)/32(reference) bit if the block size is 32-bit. Whereas stepping through rows in one column, a[i][0], will result in having 100% miss rate. After loop interchange, the time it takes to run the code decreased significantly table [3].

Performance counter stats for loop interchange matrix multiplication		
cache-misses	641,625,277	(37.42%)
L1-dcache-loads	50,705,065,567	(49.78%)
L1-dcache-load-misses	518,783,768	# 1.02% of all L1-dcache hits (49.84%)
L1-dcache-stores	9,797,076,278	(49.87%)
L1-dcache-store-misses	<not supported>	
LLC-loads	4,774,360	(49.99%)
LLC-load-misses	429,775	# 9.00% of all LL-cache hits (50.23%)
LLC-stores	2,894,571	(25.02%)
LLC-store-misses	1,596,948	(24.96%)

Table 3

Comparing the result with table 1 and table 3, there is a significant drop in L1-dcache-load-misses. The original graph has 22.65% of all L1-dcache hits. After loop interchange, it decreased down to 1.02%. This result is achieved by allowing cache hits through columns in one row. As a result, time to run the code decreased from 35 seconds to 6 seconds which is a huge improvement.

[Loop Blocking]

As other strategies, loop blocking eliminates cache misses. It divides iteration space into smaller parts to make sure that the used data is inside the cache until it reaches the capacity. Since the cache size of the CPU is limited, blocking size is carefully chose. The sub-blocks within the matrix is set to 8. The result is shown in table 4. The time taken to run the code is 8.85 seconds.

Time of running program by blocking size is shown in figure 1. As shown in fig 1, the least time it takes to run the program is the block size of 8.

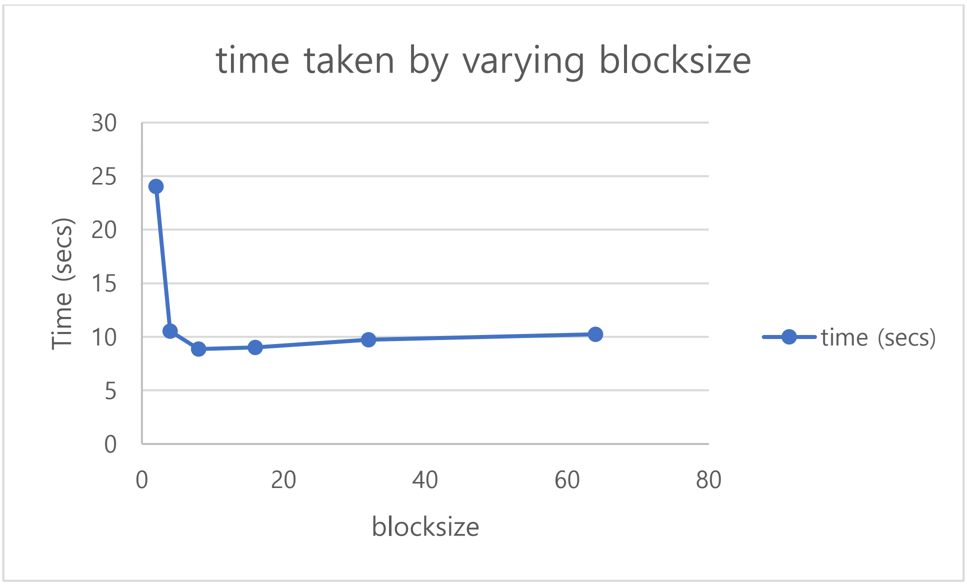

Figure 1

Performance counter stats for loop interchange matrix multiplication		
cache-misses	223,737,752	(37.50%)
L1-dcache-loads	41,421,627,105	(49.99%)
L1-dcache-load-misses	752,443,800	# 1.82% of all L1-dcache hits (50.00%)
L1-dcache-stores	9,797,076,278	(50.00%)
L1-dcache-store-misses	<not supported>	
LLC-loads	233,905,812	(50.01%)
LLC-load-misses	1,768,588	# 9.00% of all LL-cache hits (50.02%)

LLC-stores	126,057,431	(24.99%)
LLC-store-misses	733,601	(25.00%)

Table 4

As a result, the most optimized code for this specific matrix multiplication is by loop interchanging. If loop fusion and interchanging are used together, the performance does not improve but it gives longer time to run the code, around 7 seconds. It takes 6 seconds only with loop interchange.

Part2:

Machines that are going to be tested for the data are H252. CPU is Intel® Core™ i7-6700 @ 3.40GHz - 4 cores and H248 machine CPU: Intel® Core™ i7-7700 @ 3.60GHz - 4 cores both in iLab.

To determine the cache size, associativity and block size we utilized the cache size prediction, and stride. As cache size and stride increases, it will allow to assume the cache size when the elapsed time jumps because of the capacity misses. To access the data, there is an array that is allocated enough space for testing. Then a dummy value is stored in spaces where the loop touches the array. The assumption of the L1 cache size starts from 4KB and stops at 512KB. This is because conventional size of L1 cache is between 16KB to 64KB, L2 cache is from 128KB to 256KB. When the time measurement is done for once, the time was different from when the measurement is repeated several times. It can be assumed that the stride prefetch is executed since the stride is increased by the same amount during the experiment. To make the experiment accurate, the cache size iteration is repeated 10 times and the result showed that the time decreases from the first elapsed time and find the average point.

The first experiment is done by using H252 machine. Table 5 demonstrates a elapse time for certain size of array. This implies that array size is determined by the sudden jump of elapse time while the array is touched. This is because the capacity misses increased the time it takes to go through the array and therefore it increases time when the cache runs out of space to review the data and ask for new data from lower level caches.

Array size (KB)	Elapse time (nanoseconds)
4	4723
8	9828
16	19626
32	39232
64	82218
128	152706
256	320183
512	575634
1024	967599
2048	690092
4096	1389366
8192	2956644

Table 5

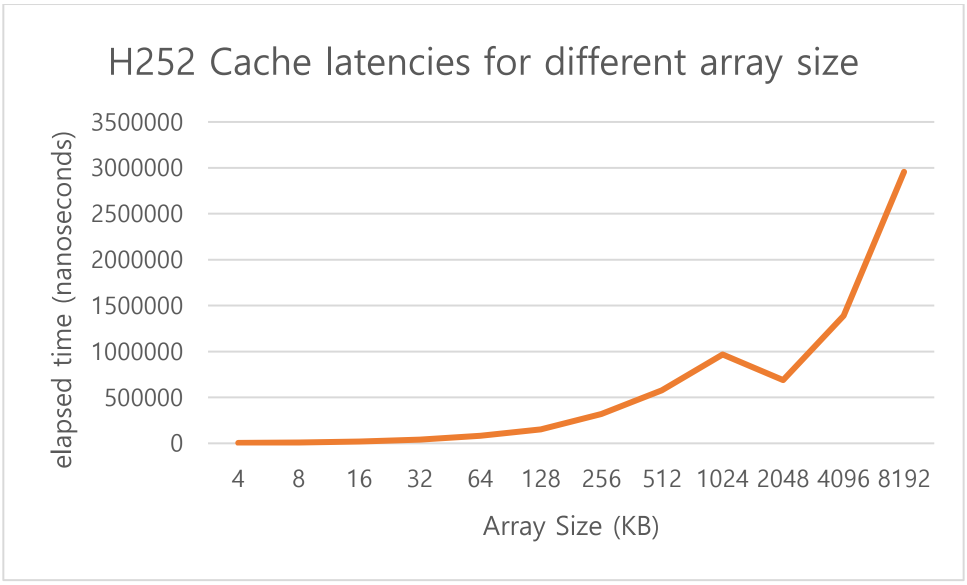

Figure 2

According to the table 5 and figure 2, the first bump in the elapse time is between 32KB and 64KB. While time increases linearly, it starts to take longer and switched to show exponential graph when the array size exceeds 32KB.

To find associativity, we progressively increase the stride size by doubling it. The size of index bit is directly related to associativity because index bit is number of lines divided by associativity. To find associativity, it is then lines divided by index. However, there is no way that we can find line numbers and index bits. Alternative way is to use the high stride size and indicate the jumps occurred at certain spots. To measure this, the array size is fixed to the size of L1 cache as found in previous graphs, (32KB). Then the sections in an array are divided into 20 spots and measure the time difference between the spots in each stride sizes.

Algorithm for detecting cache size and associativity is

```
For array size from 4KB to 8MB by 2*
    For stride s from 4byte to array size/2 by 2*
        Time the loop
        Review an array with dummy value
```

4KB cache size

Stride	4	8	16	32	64	128	256	512	1024	2048
time	101364	8341	6832	5196	3003	1465	667	201	138	138

8KB cache size

Stride	4	8	16	32	64	128	256	512	1024	2048
time	15132	7708	6353	4595	2375	1238	705	201	151	123

16KB cache size

Stride	4	8	16	32	64	128	256	512	1024	2048
time	15155	7684	6235	4603	2401	1236	756	205	154	126

32KB cache size

Stride	4	8	16	32	64	128	256	512	1024	2048
time	15157	7696	6179	4571	2352	1266	683	211	153	126

64KB cache size

Stride	4	8	16	32	64	128	256	512	1024	2048
time	15170	7713	6313	4551	2372	1247	740	221	155	127

According to the table above, the array is divided into 10 spots at a certain size of the cache. As commonly shows, they all add sharp drop of elapse time between 1st and 2nd spot of the stride which are 4 and 8 strides. Because the time jumps are significant in between 8 and 4 strides, we can assume that the associativity is 4.

The experiment is repeated in different machine H248. As clearly seen from the table, there is a jump between 32MB and 64MB of array sizes. This indicates that the L1 cache size is determined by the iterations of the touching one value in a fixed array. However, when the data is shown in a graph, it is difficult to see where the bump is to find the size of the cache.

Array size (KB)	Elapse time (nanoseconds)
4	498
8	631
16	1150
32	2251
64	5326
128	11373
256	34909
512	78699
1024	427649
2048	836687
4096	2355303

Table 6

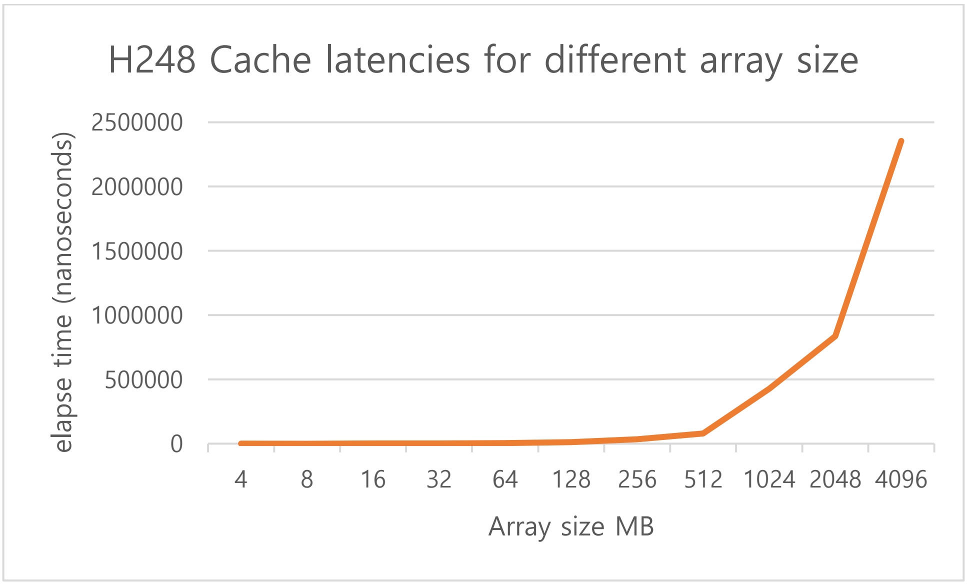

Figure 3

Again, for H248 machine, stride is iterated from 4byte to 2048 byte in a fixed array size to see the behavior. The spot in between 4 and 8 strides is bumped again. This is a similar behavior to the H252 machine. According to the 'lscpu' command in Linux machine, they both had the same size of the L1, L2 cache yet it does not tell the associativity and block size. Since the table below looks very similar except the elapse time decreases, we can assume that this machine has also 4-way associativity.

4KB cache size

Stride	4	8	16	32	64	128	256	512	1024	2048
time	37982	2779	2137	1486	927	508	231	62	47	50

8KB cache size

Stride	4	8	16	32	64	128	256	512	1024	2048
time	4069	2196	2030	1479	908	441	225	61	47	35

16KB cache size

Stride	4	8	16	32	64	128	256	512	1024	2048
time	4082	2306	2030	1498	879	511	238	68	48	35

32KB cache size

Stride	4	8	16	32	64	128	256	512	1024	2048
time	4069	2216	2030	1515	941	515	222	58	48	35

64KB cache size

Stride	4	8	16	32	64	128	256	512	1024	2048
time	4061	2238	2059	1500	912	469	208	60	46	34

Lastly, the way to determine lock size of the entity is using higher stride and lower stride. Higher stride is lesser than or equal of the page size and lower stride would reduce the number of spaces to face jump. However, the higher stride that is greater than the page size and the lower stride would increase the number of spots to access. The method to determine cache and TLB characteristics are shown to use the switch behavior of increase or decrease in spots by the High and low strides [1].

The project to find the block size effectively is yet not found. The previous research in Ohio State University demonstrates that varying higher and lower strides is capable of displaying the behavior of binary bits and find the pattern of the higher and lower strides move with the center of the block size.

Bibliography

[1]Chandran, Varadharajan. *Robust Method to Determine Cache and TLB Characteristics*. The Ohio State University, 2011.