HW2 Writeup

Meihong Ge mg476

1. PB1

   The cache is addressable by 9 bits (since 512=2^9), with a cache blocks size of 64B (64 = 2^6, so 6 bit offset), and 2-way set associative, we have a total of 2^{9-6-1}=4 sets, giving us the cache layout as shown below:

	Way 1	Way 2
Set 1	1432F	52C22
Set 2	F125C	CDE4A
Set 3	92DA3	
Set 4	ABCF2	

   To simulate the process of cache load, we translate the hex address into binary and indicate the hit/miss one reference at a time (considering we need 6 bits for cache block offset, 2 bits for cache index, the 12 bits left are used as tag). The final state of the cache is shown in the table above:

   ABCDE: 10101011110011011110. Tag: ABC, Index: 11, Offset: 011110 (Miss)
   14327: 00010100001100100111. Tag: 143, Index: 00, Offset: 100111 (Miss)
   DF148: 11011111000101001000. Tag: DF1, Index: 01, Offset: 001000 (Miss)
   8F220: 10001111001000100000. Tag: 8F2, Index: 00, Offset: 100000 (Miss)
   CDE4A: 11001101111001001010. Tag: CDE, Index: 01, Offset: 001010 (Miss)
   1432F: 00010100001100101111. Tag: 143, Index: 00, Offset: 101111 (Hit)
   52C22: 01010010110000100010. Tag: 52C, Index: 00, Offset: 100010 (Miss)
   ABCF2: 10101011110011110010. Tag: ABC, Index: 11, Offset: 110010 (Hit)
   92DA3: 10010010110110100011. Tag: 92D, Index: 10, Offset: 100011 (Miss)
   F125C: 11110001001001011100. Tag: F12, Index: 01, Offset: 011100 (Miss)

2. PB2

   a) $\mathrm{AAT} = 3\% \times (30\% \times 300 + 15) + 1 = 4.15\ \mathrm{CPU\ Cycle}$

   b) $AAT = 10\% \times (5\% \times 300 + 15) + 1 = 4\ \mathrm{CPU\ Cycle}$

3. PB3

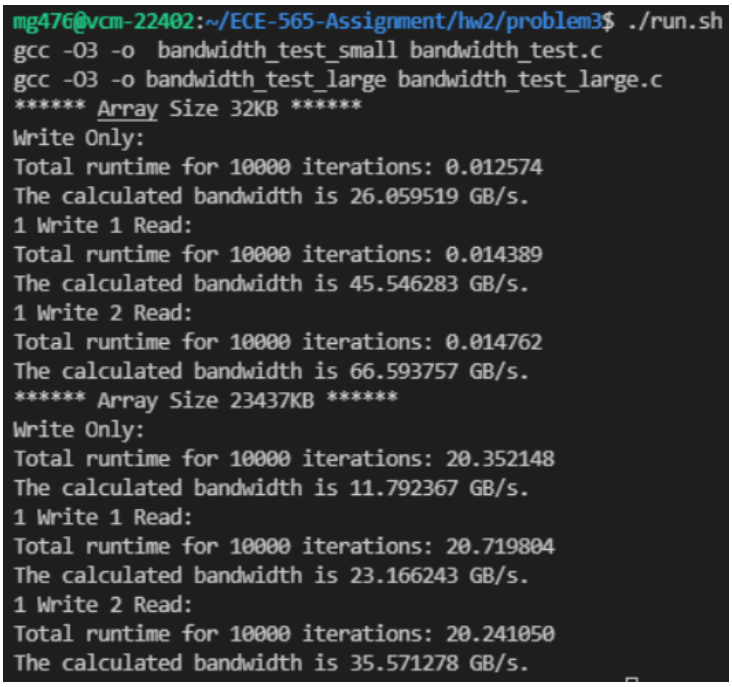
   The size of the L1 data cache is found in the folder /sys/devices/system/cpu/cpu0/cache/index0/, which is 32KB. I used int as the array data type, which is 4B in size, hence the 2^14 = 8192 array size. The program uses an independent access pattern so the access can be optimized to run in parallel and hence fully reflect the cache bandwidth.

   The program is simple, I encapsulated the core code in a function, where the array is allocated space and access based on 3 access patterns. The program run for a certain number of iterations to reach a more stable result.

   As for the large array, considering the L3 cache on my VM is 22528KB in size, which, for array of doubles, would be 2883584 in array size. I round it up to 3000000.

   The program can simply be run by typing ./run.sh, this will run both the small array size code and the

large array size code. Time it took and the measured bandwidth will be displayed for each access pattern and array size, as is shown in the print screen and table:



	Write Only	1:1 Write to Read Ratio	1:2 Write to Read Ratio
Small Array	*26.059*	*45.546*	*66.593*
Large Array	*11.792*	*23.166*	*35.571*

As can be observed from the result, the bandwidth increases as the write to read ratio decrease. We can conclude from the result that, read operation is much faster than write, most likely because it takes more machine level instruction to operate.

When the size of the array is too large for the cache to hold (L2), there is more frequent cache misses which lead to a much smaller overall bandwidth. The result also backs this speculation, considering the bandwidth measurement has diminished by a factor of approximately 2.

4. PB4

a) Performance and Analysis with 3 different ordering:

Ordering	IJK	JKI	IKJ
Time (sec)	23.731	28.880	9.623

The preliminary result seems to match the performance conception discussed in class.

For ordering IJK, A has good spatial locality, B has bad locality and finally C has good temporal locality. With a speculated miss rate of 1.125 per iteration.

For ordering JKI, A has bad locality, B has good temporal locality, C has bad locality, giving a miss rate of 2 per iteration.

For ordering IKJ, A has good temporal locality, B has good spatial locality, C has spatial locality, which result in a rough miss rate of 0.25.

The result match the relative position of the performance, although the ratio is a bit off.

b) Performance and Analysis of Loop Tiling:

Ordering	IJK	JKI	IKJ	IJK (Tiling)
Time (sec)	23.731	28.880	9.623	11.516

The size of my VM's L2 cache is 1024KB, with each entry being 8B, the total space for block of A, B and C will be N^2+1024*2*N <= 2^17, N=64 is a bit higher than the threshold, so I went with 32.

As can be seen from the result, IJK with Tiling is much faster compared with normal IJK and JKI ordering. But are still slightly slower than IKJ ordering.

The result make sense since it provides good spatial locality for A, moderate spatial locality for B and good temporal locality for C, which is an improvement over normal IJK ordering. However, the moderate spatial locality is not as optimized as the spatial locality of B observed in IKJ ordering, which yield a result a bit slower.