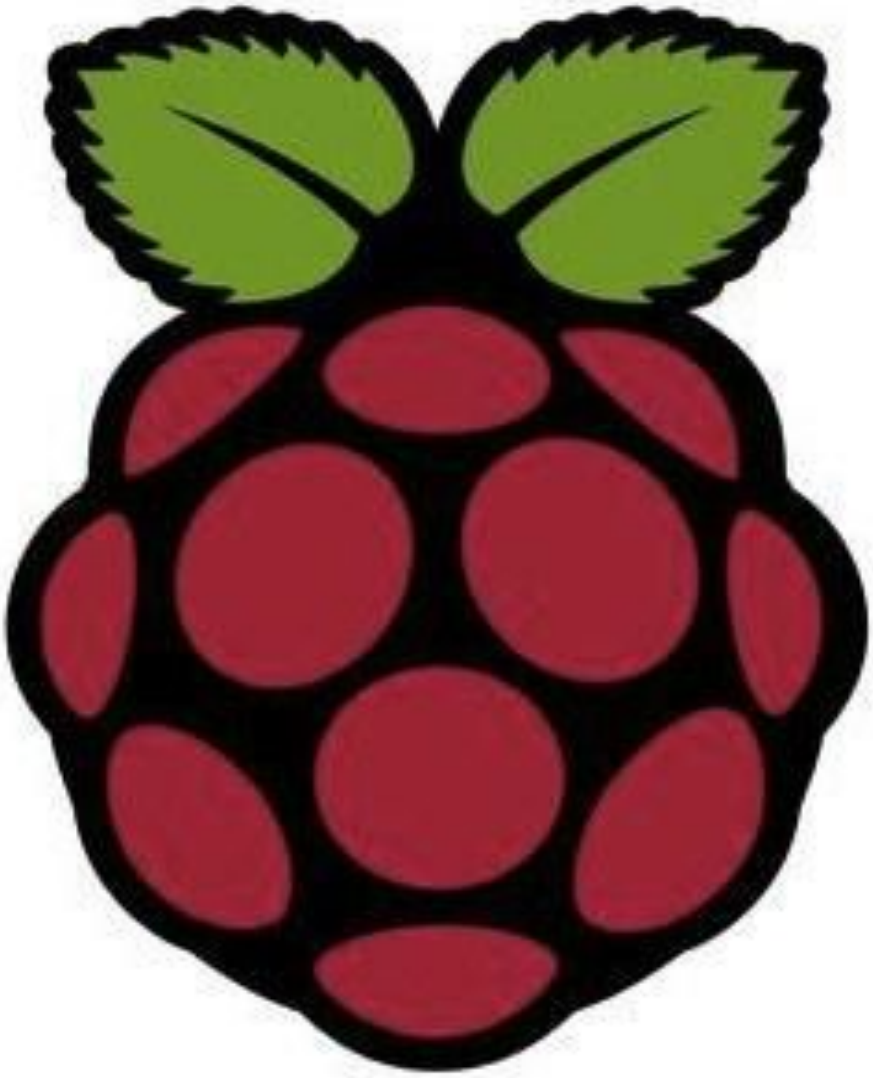


Report

Of CS631-Project 03

by *Yuanyuan Zhang*
Wentao Du
Kaijie Zhou

Table of Contents

Results Summary

Cache Configuration

Block Size: 32B;

Cache Size: 16K;

Associativity: 4-ways

Program Test

	Raspbian			Bare Metal			Qemu on Raspbian		
N	**Program 1**	**Program2**	**Rate**	**Program1**	**Program2**	**Rate**	**Program1**	**Program2**	**Rate**
1	0.12293225	0.911774983	7.416890058	0.471933	0.731286	1.549554704	0.714371	23.337347	32.66838519
10	1.077529926	8.166731057	7.579122271	4.719077	7.313289	1.549728686	7.137623	253.061854	35.45464001
100	10.78427955	81.6724683	7.573289239	47.191866	73.135523	1.549748488	71.31997	2453.907323	34.40701564
1000	107.7195322	816.7246838	7.58195535	471.948079	731.36625	1.549675235	713.208763	25207.8614	35.34429567
10000	1077.195304	8167.246755	7.581955405	4719.029407	7313.763204	1.549844803	7131.148027	232378.9054	32.58646497

Block Size Analysis

Algorithm summary

Create an Array A

First, get sample data - the time (ts) of reading a set of elements which is all cache missed (assumed L1cache size no larger than 256 bytes) .

Then, start from 4 bytes (1 word) - Read same length of elements sequence from A, one by one. Get the time of this process.

Third, check 8 bytes - same with checking 4 bytes, but read every other element.

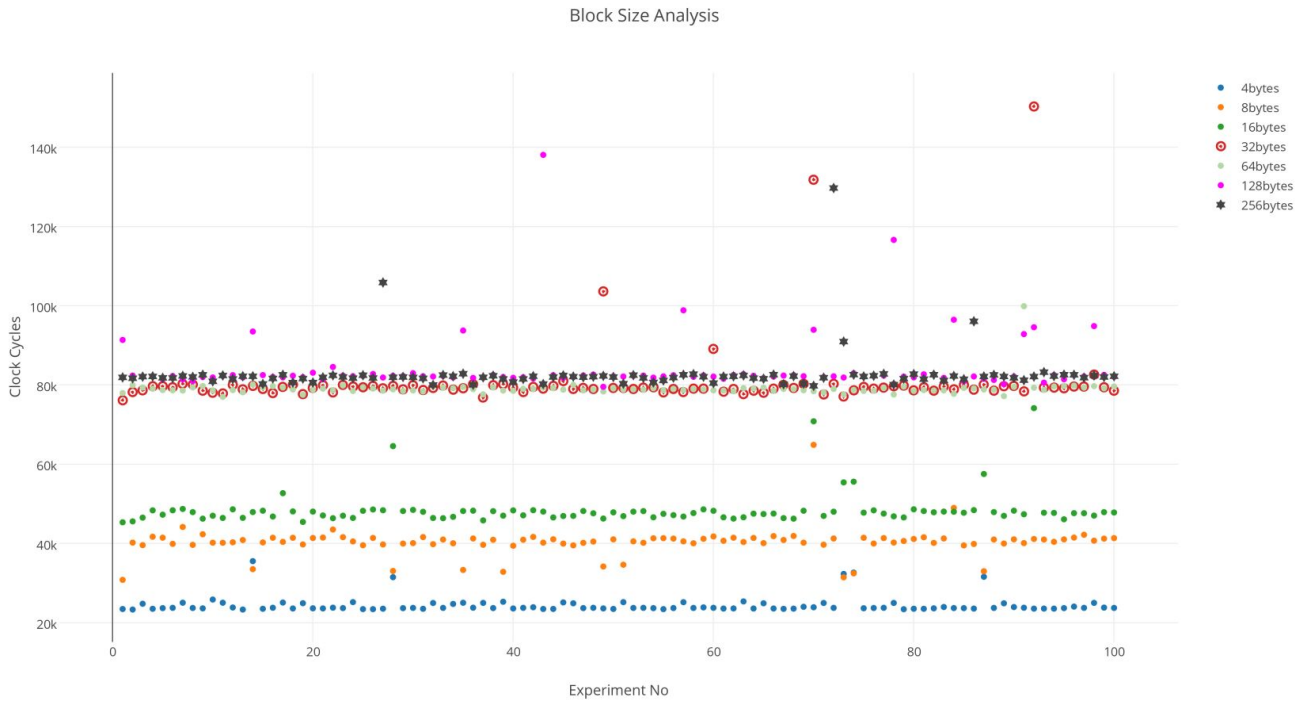
Finally, check 16 bytes (every two words), 32 bytes, 64 bytes, 128 bytes ... compare each running time with ts, the time of real the block size (more than) should be mostly same with ts.

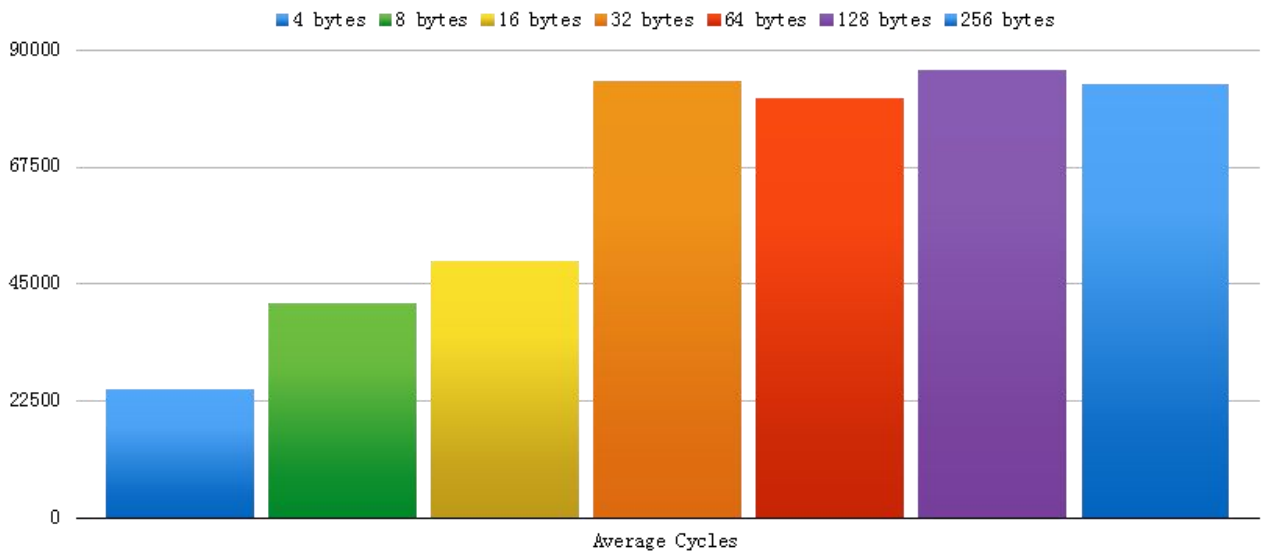
So, the first test case that reach ts should be the “block size” case.

Results

Here is part of our statistic result: average cycles of 100 experiments..

	4 bytes	8 bytes	16 bytes	32 bytes	64 bytes	128 bytes	256 bytes
Average Cycles	24778.34	41254.11	49476.54	83909.52	80715.25	85961.35	83387.14





Conclusion:

It's clear that **the block size is 32 Bytes**, for there is a "big jump" between 16 bytes and 32 bytes.

Cache Size Analysis

Algorithm summary

New an array A, and we already knew that block size is 32 bytes.

First, start from checking if it is 1 block size : write 1 element 10000 times and get the running time.

Then, we check 2 block size : write 2 element from 2 block size 10000 times.

Then check 4 block size: access the first element of each 4 blocks 10000 times.

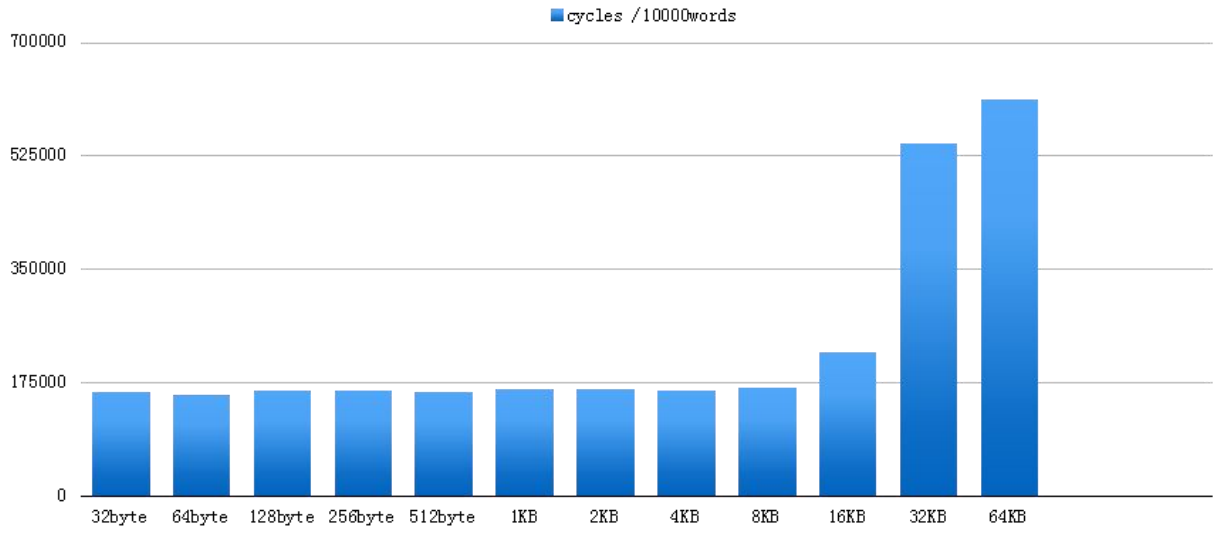
Go on checking 8,16, 32, 64, 128, 256, 512 .....blocks size.

Finally, compute the average time of writing one element 10000 times in each cases.

The cache size can be seen when there is a case much slower than the previous cases.

Results

Assumed cache Size	cycles /10000words
32byte	159520
64byte	155247
128byte	161921
256byte	162921
512byte	159256
1KB	163030
2KB	164998
4KB	162225
8KB	165287
16KB	220746
32KB	543114
64KB	610444



Conclusion

We can see from the chart that **the data cache size is 16KB**.

Degree of Associativity Analysis

Algorithm summary

New an array A, make it aligned and size it x times cache size.

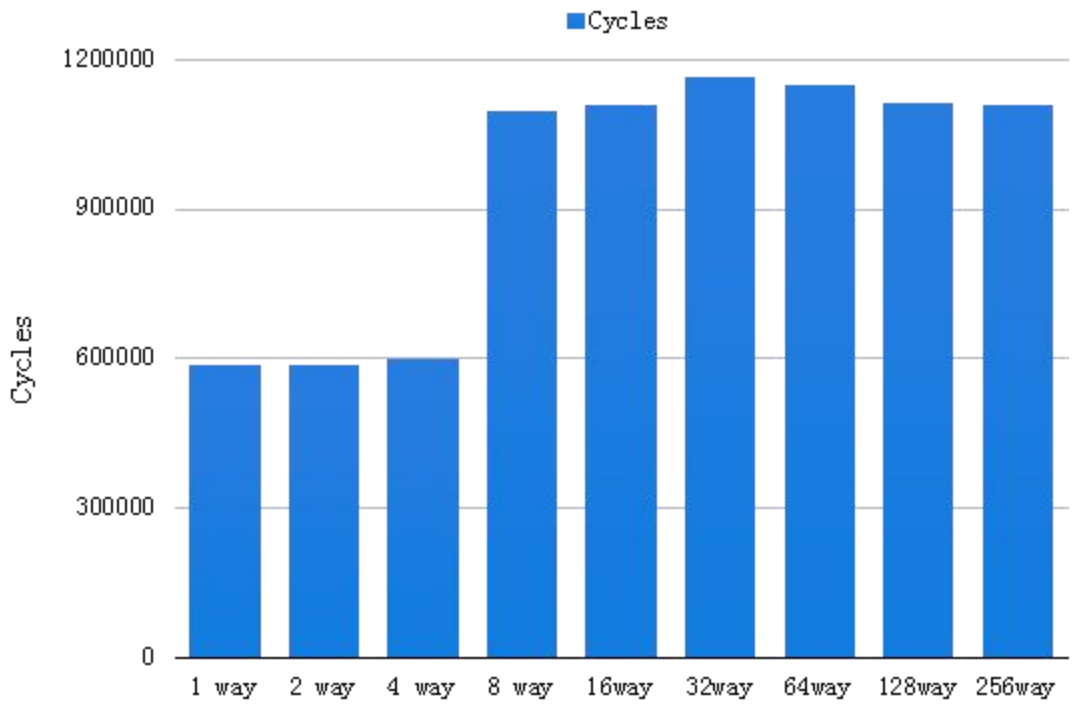
Start experiment from directly mapping, then 2 ways, 3 ways.... in following way.

We access the first number of each way 10000 times, and compute the average time of accessing one number 10000 times for each case.

The degree can be seen when the next case has a sudden increase in this average time.

Results

Assumed Associativities	Cycles
1 way	585386
2 way	584460
4 way	597684
8 way	1096297
16way	1106243
32way	1162291
64way	1148876
128way	1112991
256way	1108761



Conclusion

It's clear in the chart that **the degree of associativity is 4 ways.**

Program 1

Algorithm summary

Loop start writing the last elements of the array sequentially. Because, after initialing the array, the last part of elements are in cache already. So we can write element in reverse order to save more time.

Program 2

Algorithm summary

Loop writing every 4096 elements.

E.g. A[0], A[4096], A[8192]....

A[1], A[4097], A[8193]...

....

Integer is 4 bytes each. So, 4096 integers is 16KB, which is a cache size. We actually access the first element of a cache size to make it miss every time.

	Raspbian			Bare Metal			Qemu on Raspbian		
N	**Program1**	**Program2**	**Rate**	**Program1**	**Program2**	**Rate**	**Program1**	**Program2**	**Rate**
1	0.12293225	0.911774983	7.416890058	0.471933	0.731286	1.549554704	0.714371	23.337347	32.66838519
10	1.077529926	8.166731057	7.579122271	4.719077	7.313289	1.549728686	7.137623	253.061854	35.45464001
100	10.78427955	81.6724683	7.573289239	47.191866	73.135523	1.549748488	71.31997	2453.907323	34.40701564
1000	107.7195322	816.7246838	7.58195535	471.948079	731.36625	1.549675235	713.208763	25207.8614	35.34429567
10000	1077.195304	8167.246755	7.581955405	4719.029407	7313.763204	1.549844803	7131.148027	232378.9054	32.58646497