CS683 - Advanced Computer Architecture Assignment 1

Rohan Rajesh Kalbag, 20D170033

September 7, 2022

1 Approach

The code provided was modified slightly to make it supported by the computer I am currently using. The `GNU GCC Compiler` was used to compile the code on Windows 11.

1.1 Modified Code

`int _tmain(int argc, _TCHAR* argv[])` was changed to
`int main(int argc, char* argv[])` and also `include "stdafx.h"` was removed.

```
#include <stdio.h>
#include <time.h>
#define ARRAY_MIN (1024) /* 1/4 smallest cache */
#define ARRAY_MAX (4096*4096) /* 1/4 largest cache */
int x[ARRAY_MAX]; /* array going to stride through */

double get_seconds() { /* routine to read time in seconds */
  __time64_t ltime;
  _time64( &ltime );
  return (double) ltime;
}
int label(int i) {/* generate text labels */
  if (i<1e3) printf("%1dB,",i);
  else if (i<1e6) printf("%1dK,",i/1024);
```

```
  else if (i<1e9) printf("%1dM,",i/1048576);
  else printf("%1dG,",i/1073741824);
  return 0;
}
int main(int argc, char* argv[]) {
int register nextstep, i, index, stride;
int csize;
double steps, tsteps;
double loadtime, lastsec, sec0, sec1, sec; /* timing variables
    ↪  */

/* Initialize output */
printf("␣,");
for (stride=1; stride <= ARRAY_MAX/2; stride=stride*2)
  label(stride*sizeof(int));
printf("\n");

/* Main loop for each configuration */
for (csize=ARRAY_MIN; csize <= ARRAY_MAX; csize=csize*2) {
  label(csize*sizeof(int)); /* print cache size this loop */
  for (stride=1; stride <= csize/2; stride=stride*2) {

    /* Lay out path of memory references in array */
    for (index=0; index < csize; index=index+stride)
      x[index] = index + stride; /* pointer to next */
    x[index-stride] = 0; /* loop back to beginning */

    /* Wait for timer to roll over */
    lastsec = get_seconds();
    do sec0 = get_seconds(); while (sec0 == lastsec);

    /* Walk through path in array for twenty seconds */
    /* This gives 5% accuracy with second resolution */
    steps = 0.0; /* number of steps taken */
    nextstep = 0; /* start at beginning of path */
    sec0 = get_seconds(); /* start timer */
    do { /* repeat until collect 20 seconds */
      for (i=stride;i!=0;i=i-1) { /* keep samples same */
```

```
      nextstep = 0;
      do nextstep = x[nextstep]; /* dependency */
      while (nextstep != 0);
    }
    steps = steps + 1.0; /* count loop iterations */
    sec1 = get_seconds(); /* end timer */
  } while ((sec1 - sec0) < 20.0); /* collect 20 seconds */
  sec = sec1 - sec0;

  /* Repeat empty loop to loop subtract overhead */
  tsteps = 0.0; /* used to match no. while iterations */
  sec0 = get_seconds(); /* start timer */
  do { /* repeat until same no. iterations as above */
    for (i=stride;i!=0;i=i-1) { /* keep samples same */
      index = 0;
      do index = index + stride;
      while (index < csize);
    }
    tsteps = tsteps + 1.0;
    sec1 = get_seconds(); /* - overhead */
  } while (tsteps<steps); /* until = no. iterations */
  sec = sec - (sec1 - sec0);
  loadtime = (sec*1e9)/(steps*csize);
  /* write out results in .csv format for Excel */
  printf("%4.1f,", (loadtime<0.1) ? 0.1 : loadtime);
  }; /* end of inner for loop */
  printf("\n");
 }; /* end of outer for loop */
 return 0;
}
```

1.2 Compilation & Execution Steps

A new terminal window was opened and the following was done

```
gcc assg1.c
.\a.exe
```

1.3 Obtained Program Output

```
    ,4B,8B,16B,32B,64B,128B,256B,512B,1K,2K,4K,8K,16K,32K,64K,128K,256K,512K,1M,2M,4M,8M,16M,32M,
4K, 1.5, 1.5, 1.5, 1.5, 1.0, 0.7, 0.4, 0.2, 0.1, 0.8,
8K, 2.0, 2.0, 2.2, 2.2, 1.8, 1.3, 0.8, 0.5, 0.2, 0.1, 0.6,
16K, 1.5, 1.5, 1.5, 1.5, 1.6, 1.6, 1.1, 0.8, 0.5, 0.3, 0.1, 0.6,
32K, 1.6, 1.6, 1.7, 1.6, 1.8, 1.6, 1.6, 1.1, 0.8, 0.5, 0.3, 0.1, 0.6,
64K, 1.6, 1.7, 1.7, 1.6, 3.2, 3.2, 3.4, 3.3, 2.2, 1.6, 1.0, 0.2, 0.1, 0.6,
128K, 1.5, 1.5, 1.6, 1.6, 3.2, 3.2, 3.2, 3.2, 3.3, 2.2, 1.5, 1.0, 0.2, 0.1, 0.6,
256K, 1.5, 1.5, 1.6, 1.6, 3.2, 3.4, 3.5, 3.5, 3.6, 3.8, 2.9, 1.6, 1.0, 0.3, 0.1, 0.6,
512K, 1.5, 1.6, 1.6, 1.7, 3.4, 4.1, 6.8, 7.6, 9.1,11.1,12.2, 8.7, 6.2, 4.0, 0.6, 0.1, 0.6,
1M, 1.6, 1.6, 1.6, 1.7, 3.6, 4.1, 6.8, 7.6, 9.0,11.2,12.4,12.8, 8.8, 6.1, 3.9, 0.6, 0.1, 0.6,
2M, 1.5, 1.6, 1.6, 1.7, 3.4, 4.1, 6.7, 7.6, 9.2,11.1,12.4,12.3,12.4, 8.6, 6.3, 4.0, 0.7, 0.1, 0.6,
4M, 1.5, 1.6, 1.6, 1.7, 3.5, 4.1, 6.8, 7.6, 9.1,11.3,12.3,12.2,12.0,12.0, 8.8, 6.3, 4.0, 0.7, 0.1, 0.6,
8M, 1.6, 1.6, 1.7, 1.9, 3.8, 5.4, 8.1, 9.2,12.9,16.9,17.3,14.3,14.9,16.7,23.0, 8.7, 6.4, 4.0, 0.7, 0.1, 0.6,
16M, 1.6, 1.7, 1.8, 2.5, 5.2,14.3,22.5,22.1,27.6,37.0,58.9,53.6,52.4,54.2,70.1,21.7, 8.8, 6.4, 4.0, 0.7, 0.1, 0.6,
32M, 1.6, 1.7, 1.9, 2.6, 5.4,16.6,29.1,30.2,40.4,61.1,66.2,76.9,72.2,79.5,82.2,79.5,24.3, 8.7, 6.3, 4.0, 0.7, 0.1, 0.6,
64M, 1.6, 1.7, 1.9, 2.6, 5.4,16.8,30.6,33.1,44.2,70.8,78.2,85.1,79.5,85.1,83.4,85.1,79.5,22.5, 8.7, 6.3, 4.1, 0.7, 0.1,
    ↪ 0.6,
```

1.4 Obtained Tabular Columns

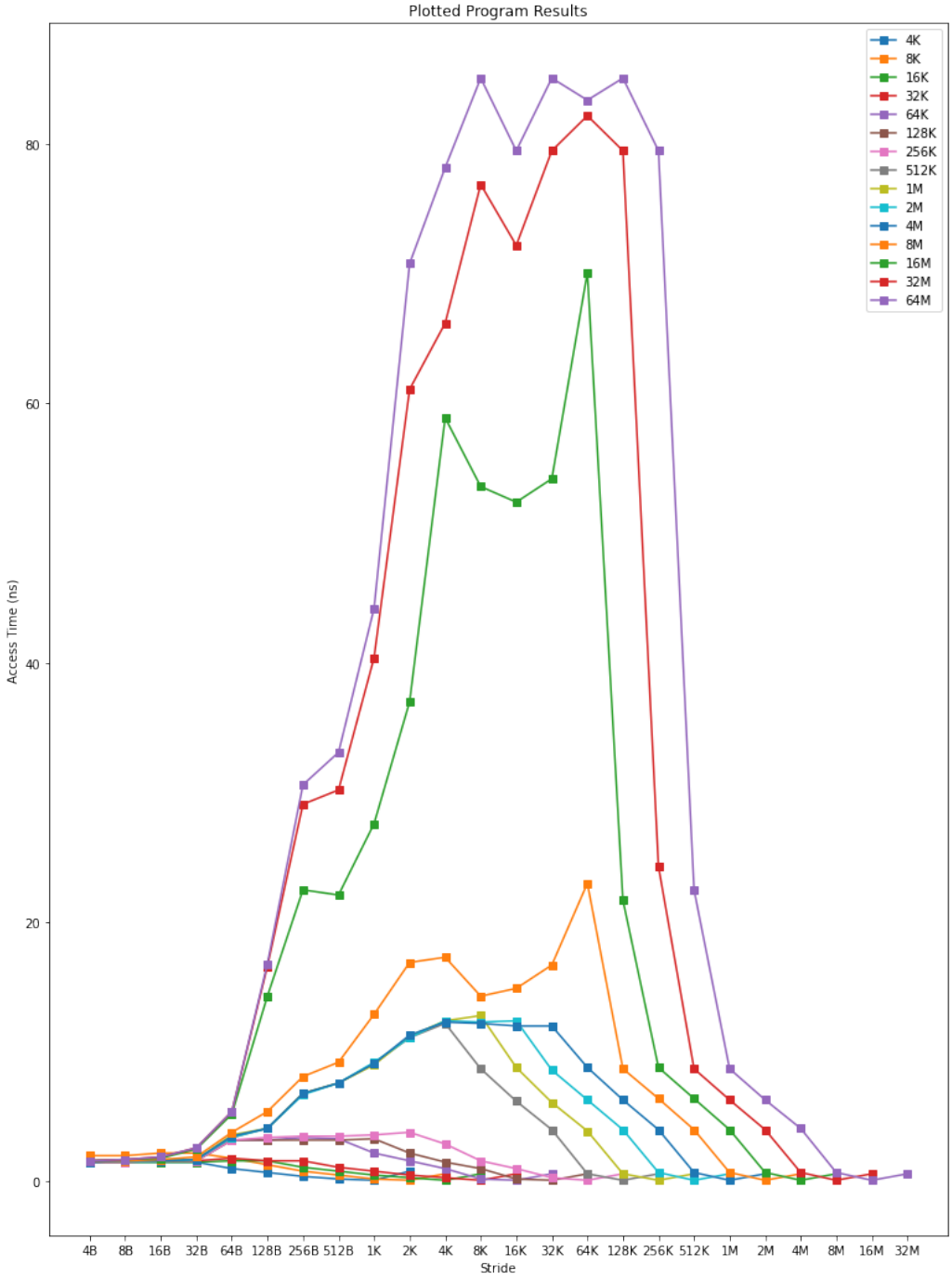
Values	4B	8B	16B	32B	64B	128B	256B	512B	1K	2K	4K	8K
4K	1.5	1.5	1.5	1.5	1	0.7	0.4	0.2	0.1	0.8		
8K	2	2	2.2	2.2	1.8	1.3	0.8	0.5	0.2	0.1	0.6	
16K	1.5	1.5	1.5	1.5	1.6	1.6	1.1	0.8	0.5	0.3	0.1	0.6
32K	1.6	1.6	1.7	1.6	1.8	1.6	1.6	1.1	0.8	0.5	0.3	0.1
64K	1.6	1.7	1.7	1.6	3.2	3.2	3.4	3.3	2.2	1.6	1	0.2
128K	1.5	1.5	1.6	1.6	3.2	3.2	3.2	3.2	3.3	2.2	1.5	1
256K	1.5	1.5	1.6	1.6	3.2	3.4	3.5	3.5	3.6	3.8	2.9	1.6
512K	1.5	1.6	1.6	1.7	3.4	4.1	6.8	7.6	9.1	11.1	12.2	8.7
1M	1.6	1.6	1.6	1.7	3.6	4.1	6.8	7.6	9	11.2	12.4	12.8
2M	1.5	1.6	1.6	1.7	3.4	4.1	6.7	7.6	9.2	11.1	12.4	12.3
4M	1.5	1.6	1.6	1.7	3.5	4.1	6.8	7.6	9.1	11.3	12.3	12.2
8M	1.6	1.6	1.7	1.9	3.8	5.4	8.1	9.2	12.9	16.9	17.3	14.3
16M	1.6	1.7	1.8	2.5	5.2	14.3	22.5	22.1	27.6	37	58.9	53.6
32M	1.6	1.7	1.9	2.6	5.4	16.6	29.1	30.2	40.4	61.1	66.2	76.9
64M	1.6	1.7	1.9	2.6	5.4	16.8	30.6	33.1	44.2	70.8	78.2	85.1

Values	16K	32K	64K	128K	256K	512K	1M	2M	4M	8M	16M	32M
4K												
8K												
16K												
32K	0.6											
64K	0.1	0.6										
128K	0.2	0.1	0.6									
256K	1	0.3	0.1	0.6								
512K	6.2	4	0.6	0.1	0.6							
1M	8.8	6.1	3.9	0.6	0.1	0.6						
2M	12.4	8.6	6.3	4	0.7	0.1	0.6					
4M	12	12	8.8	6.3	4	0.7	0.1	0.6				
8M	14.9	16.7	23	8.7	6.4	4	0.7	0.1	0.6			
16M	52.4	54.2	70.1	21.7	8.8	6.4	4	0.7	0.1	0.6		
32M	72.2	79.5	82.2	79.5	24.3	8.7	6.3	4	0.7	0.1	0.6	
64M	79.5	85.1	83.4	85.1	79.5	22.5	8.7	6.3	4.1	0.7	0.1	0.6

1.5 Analysis and Interpretation using Python

The following code was used to plot the graph given in the subsequent page to analyse the data better

```
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
x = pd.read_csv("output.csv")
plt.rcParams['figure.figsize'] = [13, 12]
for i in range(len(x)):
    val = x.iloc[i]
    plt.plot(list(val.keys())[1:], list(val.values)[1:], label=
        ↪ list(val.values)[0], marker='s')
plt.title("Interpretation␣of␣Program␣Output")
plt.xlabel("Stride")
plt.ylabel("Read␣(ns)")
plt.legend()
plt.show()
```



Plotted Program Results

1.6 Answer to Questions

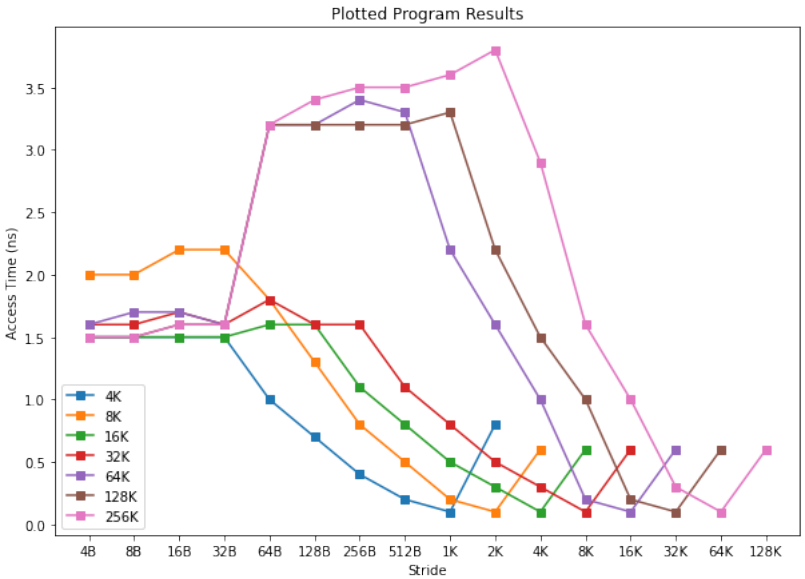
1.6.1 What is the overall size and block size of the second level cache?

We see that the plots are divided into three possible groups. The **red** plot corresponding to 32K denotes overall size corresponding to L1 cache. The **pink** plot corresponding to 256K denotes overall size corresponding to L2 cache.

Thus the **size per core** of **L2 cache** is given by **256 kB**. Similarly by observing for others we get the size of L1 and L3 cache to be 32 kB and 8 MB respectively

To find the block size we find the **array stride** corresponding to the **nearly constant access times** (region where plot becomes nearly flat) because of persistent cache hits, for the pink plot which happens at at stride of $\approx$ 128 bytes for access time of 3.5 ns

Thus the **block size** of **L2 cache** is given by **128 bytes**.

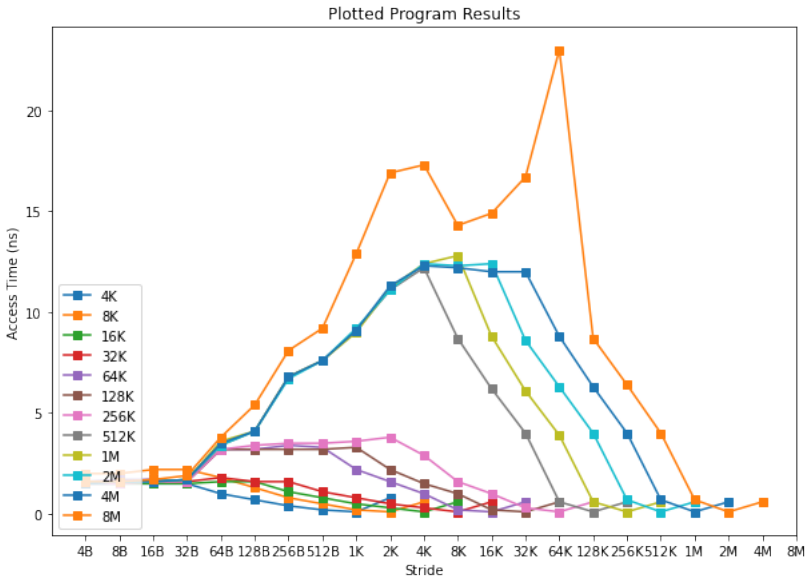


Magnified Plot for L1 and L2

1.6.2 What is the miss penalty of the second level cache?

We know that the miss penalty for L2 cache is given by the time difference of L3 cache access time and L2 cache access time.

We know from previous question that the L2 cache access time is $\approx$ **3.5 ns**



Magnified Plot for L1 and L2 and L3

We see from this graph that the L3 access time corresponds to a stride of around 8 kB (orange graph) which is an access time of $\approx$ **17 ns**

Thus we see that **miss penalty for L2 cache** is 17 ns - 3.5 ns = **13.5 ns**

1.6.3 What is the associativity of the second level cache?

Since L2 cache has size **256 kB** and block size of **128 bytes** and main memory has **16 bit** addresses.

We need $log_2(128) = 7$ bits to address a single cache element.

No of cache lines = 256kB/128B = 2000.

Let us assume the Set Associativity (S.A) to be **8**. The S.A is given by **(no of cache lines/no of sets)** = $2000/256 = 7.8 \approx 8$. No of sets is 256 as we have 256 sets of 1kB (128 bytes $\times$ 8(assumed S.A)). Hence Verified

Thus the **L2 cache** is **8 way set associative**

2 References

- Computer Architecture Book - Hennessey and Patterson
- `https://stackoverflow.com/questions/21299488/cache-size-estimation-on-your-system`
- `http://www.vlsiip.com/cache/cache_0003.html`
- `https://stackoverflow.com/questions/24163566/set-associative-cache-calculate-size-of-tag`
- `https://stackoverflow.com/questions/39457343/find-associativity-of-this-cache#:~:text=With%20two%20bytes%2C%20we%20can,is%20thus%208%2Dway%20associative.`

3 Submission

The submission `submission.zip` contains the benchmark code `assg1.c`, the executable `a.exe`, the data `output.csv`, the plotting script `analysis.ipynb`, `output.txt` contains the raw data obtained from the terminal.