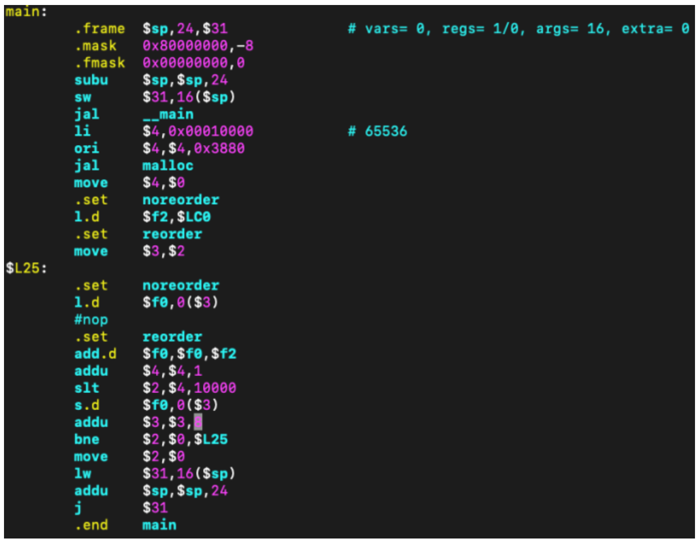
ECE 552 Lab 4 Report – By Yuhe Chen and Rudy Jin

Q1. Microbenchmark for NLP

Please see mbq1.c and q1.cfg for detailed coding. In our microbenchmark, we decided to create an array of 10000 type double – arr[10000]; the flag chosen is -O2. We adjusted our block size in cache config to be 8 bytes, as one double is 8 bytes, index++ is basically referring to the next block. We access the array in a for loop with index each increases by 1. We expect there will be 0 misses in the array accessing, as we are always prefetching next line; in this case, accessing arr in this iteration is already pre-fetched in the last cycle (because we defined our block size to be the same as double type). The results are satisfying. The total D_{L1} access is 43848, hit is 43760. This seems reasonable after checking assembly code, given that there are also other elements being accessed in our program besides the array causing some misses. We enhanced our belief by checking prefetch hit, which is defined as the number of prefetched item is already in the cache. This value is 0, meaning that every time we prefetch, the data is not currently in the cache. This should be expected because the successful microbenchmark execution with correct next-line pre-fetcher should only fetch once in a previous cycle in the for loop. Therefore, our implementation of next-line prefetcher is proved.

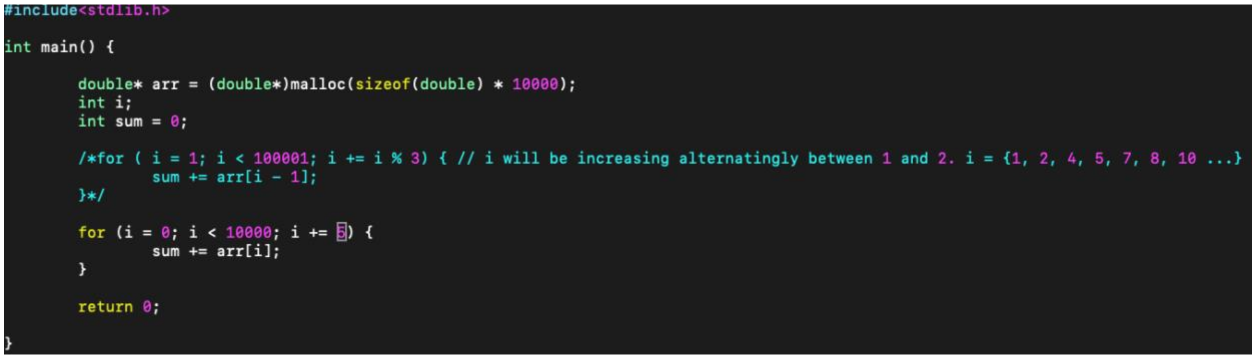


```
main:
        .frame  $sp,24,$31              # vars= 0, regs= 1/0, args= 16, extra= 0
        .mask   0x80000000,-8
        .fmask  0x00000000,0
        subu    $sp,$sp,24
        sw      $31,16($sp)
        jal     __main
        li      $4,0x00010000           # 65536
        ori     $4,$4,0x3880
        jal     malloc
        move    $4,$0
        .set    noreorder
        l.d     $f2,$LC0
        .set    reorder
        move    $3,$2
$L25:
        .set    noreorder
        l.d     $f0,0($3)
        #nop
        .set    reorder
        add.d   $f0,$f0,$f2
        addu    $4,$4,1
        slt     $2,$4,10000
        s.d     $f0,0($3)
        addu    $3,$3,8
        bne     $2,$0,$L25
        move    $2,$0
        lw      $31,16($sp)
        addu    $sp,$sp,24
        j       $31
        .end    main
```

Figure 1. Assembly Code for Q1 Microbenchmark

Q2. Microbenchmark for Stride

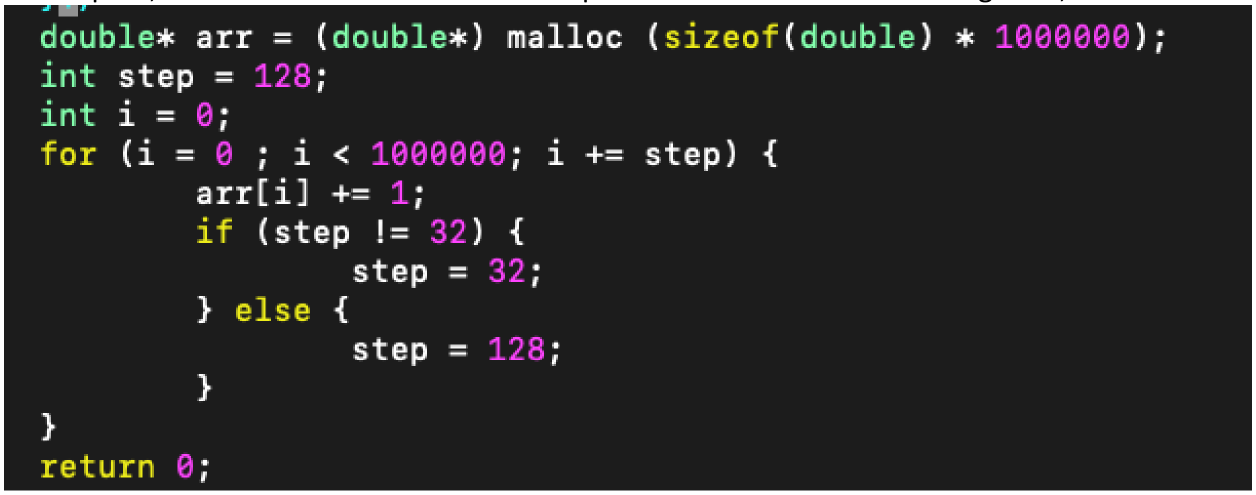
Please see mbq2.c and q2.cfg for detailed coding. We defined block size to be 8 bytes, and an array of double of size 10000. The parameters are chosen like Q1. In Figure 2, Case I, we will be expecting near perfect prefetch (not to consider several misses in the beginning to develop stride pattern). This is because the loop presents a block pattern of increasing by 5 * sizeof(double) each time, which is a constant stride in memory access. The total D_{L1} access is 3847, hit is 3739, which is reasonable. The prefetch hit also gives 0, which is expected, reason explained in details in Q1.



```
#include<stdlib.h>

int main() {

        double* arr = (double*)malloc(sizeof(double) * 10000);
        int i;
        int sum = 0;

        /*for ( i = 1; i < 100001; i += i % 3) { // i will be increasing alternatingly between 1 and 2. i = {1, 2, 4, 5, 7, 8, 10 ...}
                sum += arr[i - 1];
        }*/

        for (i = 0; i < 10000; i += 5) {
                sum += arr[i];
        }

        return 0;

}
```

Figure 2. Q2 Microbenchmark Case I

In compare, it is harder to find counter examples to miss from cache. See Figure 3, Case II.



```
double* arr = (double*) malloc (sizeof(double) * 1000000);
int step = 128;
int i = 0;
for (i = 0 ; i < 1000000; i += step) {
        arr[i] += 1;
        if (step != 32) {
                step = 32;
        } else {
                step = 128;
        }
}
return 0;
```

Figure 2. Q2 Microbenchmark Case II

In this case, we expect arr[i] will miss 100000 / 80 = 12500 times as the average step it takes is ((32 + 128) * 0.5) = 80. The simulation result gives 63133 accesses with 50491 hits, which is expected as 12500 + 50491 is close to 63133. Thus, stride correctness is proved.

Q3. Average Memory Access Computation

$T_{access} = T_{L1\text{-}hit} + P_{L1\text{-}miss} \times (T_{L2\text{-}hit} + P_{L2\text{-}miss} \times T_{mem})$

Given that $T_{L1\text{-}hit} = 1$ *sec* ; $T_{L2\text{-}hit} = 10$ *sec* ; $T_{mem} = 100$ *sec*

Config	L1 Miss Rate	L2 Miss Rate	Average Access Time
No Prefetecher	4.16%	11.4%	1 + 0.0416 * (10 + 0.114 * 100) = 1.89 sec
Next Line	4.19%	8.38%	1 + 0.0419 * (10 + 0.0838 * 100) = 1.77 sec
Stride	3.85%	5.78%	1 + 0.0385 * (10 + 0.0578 * 100) = 1.61 sec

Q4. Graph – Stride Entry Number vs L1-Miss Rate

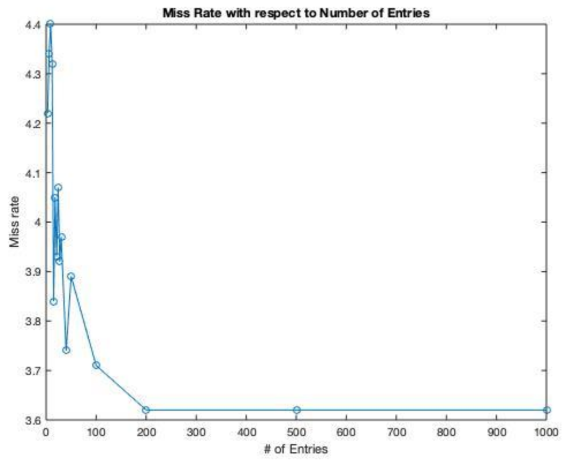


Figure 4. Graph of Entries vs Miss Rate

The graph is generated using MatLab. At first, there exists **slight fluctuation** when the # of entries are small. This is because **conflicts/replacements of entries** are likely to happen, and the miss rate may vary due to some specific address pattern in cache testcase config. When the # of entries are large, the miss rate is **saturated to 3.62** because the RPT table is large enough to find most of the potential stride pattern. At this point, the most influential factor to miss rate is the address pattern regarding **striding frequency** itself in the testcase.

Q5. More Statistics for Conclusion?

If one statistic can be added, it should be average accessing time. The miss rate is important, but the average access time is what a design should be aiming to optimize. With access time, we can test out and find the optimal parameters for the design, regarding different test cases, which can sufficiently reduce the memory bottleneck and improve performance.

Q6. Open-Ended Microbenchmark

We will talk about our modification on stride solution, as it gives lower miss rate than machine learning model. To test if it would miss, we can use the microbenchmark in Figure 2, as the modification-on-stride solution would still not be able to capture inconsistent change in address. Our result is the same in Q2, which proves its vulnerability. However, we still need to prove that it only pre-fetches in stable stage. Therefore, Figure 6 represents our microbenchmark algorithm. As you can see, the repetition happens every third cycle, and that is the only time prefetch will be activated, but it will miss because stride is different. The expected misses is (100000 / (128 + 64)*0.5) = 1041, which is close to simulation miss of 1081. Proved.

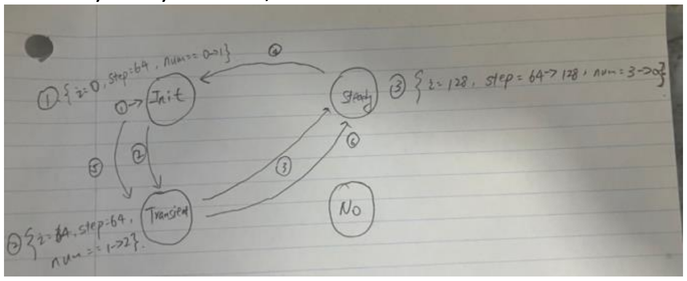
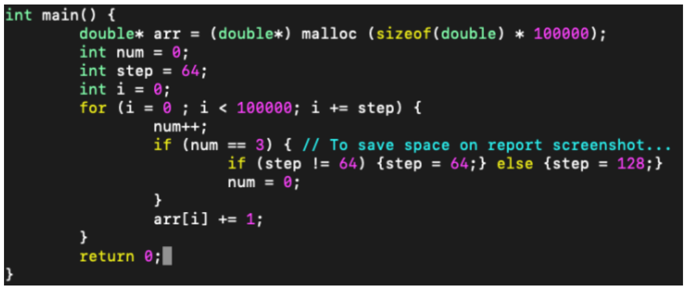

① {i=0, step=64, num== 0->1}
④
Init
Steady
③ {i=128, step= 64->128, num= 3->0}
⑤
②
③
⑥
Transient
No
② {i=64, step=64, num== 1->2}

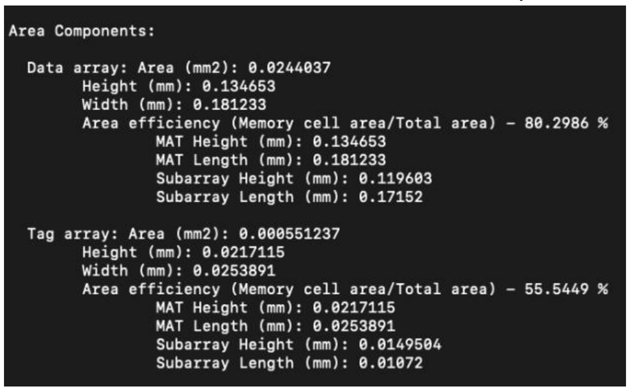
Figure 5. FSM Update Logic



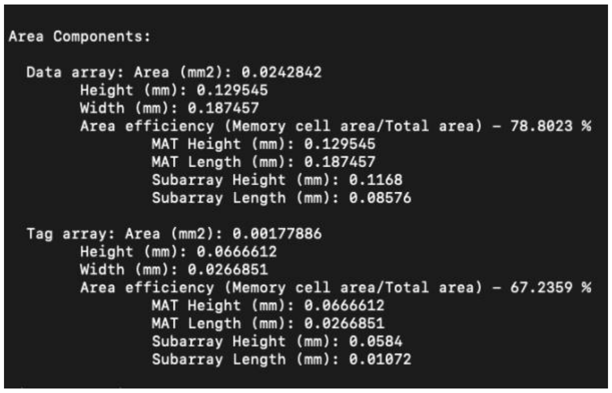
```
int main() {
        double* arr = (double*) malloc (sizeof(double) * 100000);
        int num = 0;
        int step = 64;
        int i = 0;
        for (i = 0 ; i < 100000; i += step) {
                num++;
                if (num == 3) { // To save space on report screenshot...
                        if (step != 64) {step = 64;} else {step = 128;}
                        num = 0;
                }
                arr[i] += 1;
        }
        return 0;
}
```

Figure 6. Open-Ended Microbenchmark

Q7. Additional Question on Open Implementation
We have developed two solutions. The first solution is a modification on stride pre-fetcher. In addition to increasing the number of entries to reduce possibility of different PCs indexing into the same storage, we restrict the prefetching unless it is in stable stage. In this case, we eliminate possible mispredictions, only fetching when confident, therefore reduces the possibility of evicting useful data. In this method, our average miss rate among three benchmarks is 1.963. Our second solution consists of a perceptron model, where it is a convoluted process of one cycle delay. This is because we do not have the next-data information, so we can only update the L weight & bias at iteration L+1. If the model prediction gives 1, we use next-line fetcher, vise versa. The backward propagation process happens when the predictor tells to do next line fetch but finds out it is incorrect in the next cycle. Using this method, we reach an average miss rate of 2.093. Thus, the first solution performs better. We used CACTI to calculate its hardware requirements and efficiency. The left picture is the original Cache requirements for Open-Ended part. The second picture is the cache implemented in our method. Basically, it takes about the same hardware resource as original Cache requirements. Despite in the simulation we ignore the hardware resource, our methodology creates a large overhead, close to 100% of L1 cache requirements.



```
Area Components:

  Data array: Area (mm2): 0.0244037
        Height (mm): 0.134653
        Width (mm): 0.181233
        Area efficiency (Memory cell area/Total area) - 80.2986 %
                MAT Height (mm): 0.134653
                MAT Length (mm): 0.181233
                Subarray Height (mm): 0.119603
                Subarray Length (mm): 0.17152

  Tag array: Area (mm2): 0.000551237
        Height (mm): 0.0217115
        Width (mm): 0.0253891
        Area efficiency (Memory cell area/Total area) - 55.5449 %
                MAT Height (mm): 0.0217115
                MAT Length (mm): 0.0253891
                Subarray Height (mm): 0.0149504
                Subarray Length (mm): 0.01072
```



```
Area Components:

  Data array: Area (mm2): 0.0242842
        Height (mm): 0.129545
        Width (mm): 0.187457
        Area efficiency (Memory cell area/Total area) - 78.8023 %
                MAT Height (mm): 0.129545
                MAT Length (mm): 0.187457
                Subarray Height (mm): 0.1168
                Subarray Length (mm): 0.08576

  Tag array: Area (mm2): 0.00177886
        Height (mm): 0.0666612
        Width (mm): 0.0266851
        Area efficiency (Memory cell area/Total area) - 67.2359 %
                MAT Height (mm): 0.0666612
                MAT Length (mm): 0.0266851
                Subarray Height (mm): 0.0584
                Subarray Length (mm): 0.01072
```

Disclaimer:

We did the coding together, supervised each other, and seperated the report work equally.