Yu Liu

Oct. 24, 2019

Problem 1

Using the same trace of homework1, study the effects of different prefetching policies and report your results based on dineroIV simulator, and the provided trace in homework 1. Please use the prefetching capabilities provided in dineroIV (the -tfetch, -pfdist and –pfabort switches). Attempt to find the best prefetching policy and discuss why you feel this would be the best policy for the given workload. Make sure to explain prefetching policy per each switch (20pts).

Answer:

Using the dineroIV with the same data source (trace.din), run with the different switches, I got the data presendted in the Table 1 and Table 2.

In order to simplify the data pattern, only miss number and miss rate are issed, the hit number can be got by total number minus miss number, as well as the hit rate can minus by miss rate.

Table 1 shows the total miss number and rate combined with demanded fetch and prefetch fetch, while Table 2 present the data of prefetch's only.

As a conclusion, from both two tables, I found the best prefetch performance is under configuration of : always prefetch policy, prefetch distance of 1 and 4B, prefetch abort rate in 0% and 20%, miss rate is low to 0.01%. I think this result shows this trace.din file is with a regular tag pattern, the prefetcher with small stride achieves the better performance relatively.

Fetch policy d-demand

Prefetch abort %	0%								20%								50%								100%							
fatch distance	1	4	16	32	64	128	256	4096	1	4	16	32	64	128	256	4096	1	4	16	32	64	128	256	4096	1	4	16	32	64	128	256	4096
Demand fetch+prefetch fetch	832477	832477	832477	832477	832477	832477	832477	832477	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na
Demand Miss+prefetch Miss	318	318	318	318	318	318	318	318	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na
Miss rate	0.04%	0.04%	0.04%	0.04%	0.04%	0.04%	0.04%	0.04%	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na

Fetch policy a- always

Prefetch abort %	0%								20%								50%								100%							
fatch distance	1	4	16	32	64	128	256	4096	1	4	16	32	64	128	256	4096	1	4	16	32	64	128	256	4096	1	4	16	32	64	128	256	4096
Demand fetch+prefetch fetch	1560441	1560441	1560441	1560441	1560441	1560441	1560441	1560441	1415064	1415064	1415064	1415064	1415064	1415064	1415064	1415064	1196782	1196782	1196782	1196782	1196782	1196782	1196782	1196782	832477	832477	832477	832477	832477	832477	832477	832477
Demand Miss+prefetch Miss	332	346	387	405	1403	1406	1406	1406	332	346	387	405	1320	1323	1323	473	332	346	387	405	1100	1103	1103	na	318	318	318	318	318	318	318	318
Miss rate	0.02%	0.02%	0.02%	0.03%	0.09%	0.09%	0.09%	0.09%	0.02%	0.02%	0.03%	0.03%	0.09%	0.09%	0.09%	0.03%	0.03%	0.03%	0.03%	0.03%	0.09%	0.09%	0.09%	na	0.04%	0.04%	0.04%	0.04%	0.04%	0.04%	0.04%	0.04%

Fetch policy m- miss

Prefetch abort %	0%								20%								50%								100%							
fatch distance	1	4	16	32	64	128	256	4096	1	4	16	32	64	128	256	4096	1	4	16	32	64	128	256	4096	1	4	16	32	64	128	256	4096
Demand fetch+prefetch fetch	832525	832529	832538	832638	832666	832666	832666	832666	832520	832523	832527	832531	832561	832561	832561	832561	832508	832508	832512	832516	832523	832523	832523	832523	832477	832477	832477	832477	832477	832477	832477	832477
Demand Miss+prefetch Miss	328	333	345	347	414	414	414	414	327	332	341	349	399	399	399	399	322	326	333	341	366	366	366	366	318	318	318	318	318	318	318	318
Miss rate	0.04%	0.04%	0.04%	0.04%	0.05%	0.05%	0.05%	0.05%	0.04%	0.04%	0.04%	0.04%	0.05%	0.05%	0.05%	0.05%	0.04%	0.04%	0.04%	0.04%	0.04%	0.04%	0.04%	0.04%	0.04%	0.04%	0.04%	0.04%	0.04%	0.04%	0.04%	0.04%

Fetch policy t-tagged

Prefetch abort %	0%								20%								50%								100%							
fatch distance	1	4	16	32	64	128	256	4096	1	4	16	32	64	128	256	4096	1	4	16	32	64	128	256	4096	1	4	16	32	64	128	256	4096
Demand fetch+prefetch fetch	832569	832569	832569	832569	832569	832569	832569	832569	832544	832544	832544	832544	832544	832544	832544	832544	832522	832522	832522	832522	832523	832523	832523	832523	832477	832477	832477	832477	832477	832477	832477	832477
Demand Miss+prefetch Miss	328	335	350	367	414	414	414	414	327	332	342	359	399	399	399	399	325	327	336	346	366	366	366	366	318	318	318	318	318	318	318	318
Miss rate	0.04%	0.04%	0.04%	0.04%	0.05%	0.05%	0.05%	0.05%	0.04%	0.04%	0.04%	0.04%	0.05%	0.05%	0.05%	0.05%	0.04%	0.04%	0.04%	0.04%	0.04%	0.04%	0.04%	0.04%	0.04%	0.04%	0.04%	0.04%	0.04%	0.04%	0.04%	0.04%

Table 1. Miss number and rate with what of demand fetch and prefetch fatch are combined, under the various configuration of prefetch distance, abort percentage, fetch policy (demand, always, miss, and tagged)

Fetch policy

Prefetch abort %	0%								20%								50%								100%							
fatch distance	1	4	16	32	64	128	256	4096	1	4	16	32	64	128	256	4096	1	4	16	32	64	128	256	4096	1	4	16	32	64	128	256	4096
Prefetch fetch	0	0	0	0	0	0	0	0	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na
Prefetch Miss	0	0	0	0	0	0	0	0	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na
Miss rate	-	-	-	-	-	-	-	-	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na

Fetch policy a- always

Prefetch abort %	0%								20%								50%								100%							
fatch distance	1	4	16	32	64	128	256	4096	1	4	16	32	64	128	256	4096	1	4	16	32	64	128	256	4096	1	4	16	32	64	128	256	4096
Prefetch fetch	727964	727964	727964	727964	727964	727964	727964	727964	582587	582587	582587	582587	582587	582587	582587	582587	364305	364305	364305	364305	364305	364305	364305	364305	0	0	0	0	0	0	0	0
Prefetch Miss	82	83	96	95	508	508	508	508	82	83	96	95	481	481	481	126	82	83	96	95	396	396	396	na	0	0	0	0	0	0	0	0
Miss rate	0.01%	0.01%	0.01%	0.01%	0.07%	0.07%	0.07%	0.07%	0.01%	0.01%	0.02%	0.02%	0.06%	0.06%	0.06%	0.02%	0.02%	0.02%	0.03%	0.03%	0.11%	0.11%	0.11%	na	-	-	-	-	-	-	-	-

Fetch policy m- miss

Prefetch abort %	0%								20%								50%								100%							
fatch distance	1	4	16	32	64	128	256	4096	1	4	16	32	64	128	256	4096	1	4	16	32	64	128	256	4096	1	4	16	32	64	128	256	4096
Prefetch fetch	48	52	61	61	89	89	89	89	43	46	50	54	74	74	74	74	31	31	35	39	46	46	46	46	0	0	0	0	0	0	0	0
Prefetch Miss	44	45	48	50	89	89	89	89	40	40	42	46	74	74	74	74	29	29	30	32	46	46	46	46	0	0	0	0	0	0	0	0
Miss rate	47.83%	46.39%	44.04%	45.05%	50.00%	50.00%	50.00%	50.00%	48.19%	46.51%	45.65%	46.00%	50.00%	50.00%	50.00%	50.00%	48.33%	48.33%	46.15%	45.07%	50.00%	50.00%	50.00%	50.00%	-	-	-	-	-	-	-	-

Fetch policy t-tagged

Prefetch abort %	0%								20%								50%								100%							
fatch distance	1	4	16	32	64	128	256	4096	1	4	16	32	64	128	256	4096	1	4	16	32	64	128	256	4096	1	4	16	32	64	128	256	4096
Prefetch fetch	82	82	82	82	82	82	82	82	67	67	67	67	67	67	67	67	45	45	45	45	46	46	46	46	0	0	0	0	0	0	0	0
Prefetch Miss	77	74	69	70	89	89	89	89	63	59	55	56	74	74	74	74	43	40	37	37	46	46	46	46	0	0	0	0	0	0	0	0
Miss rate	48.43%	47.44%	45.70%	46.06%	52.06%	52.06%	52.06%	52.06%	48.46%	46.83%	45.08%	45.53%	52.48%	52.48%	52.48%	52.48%	48.86%	47.06%	45.12%	45.12%	50.00%	50.00%	50.00%	50.00%	-	-	-	-	-	-	-	-

Table 2. Miss number and rate of prefetch fatch, under the various configuration of prefetch distance, abort

percentage, fetch policy (demand, always, miss, and tagged)

Figure 1 shows the screenshot of one of the results running with dineroIV

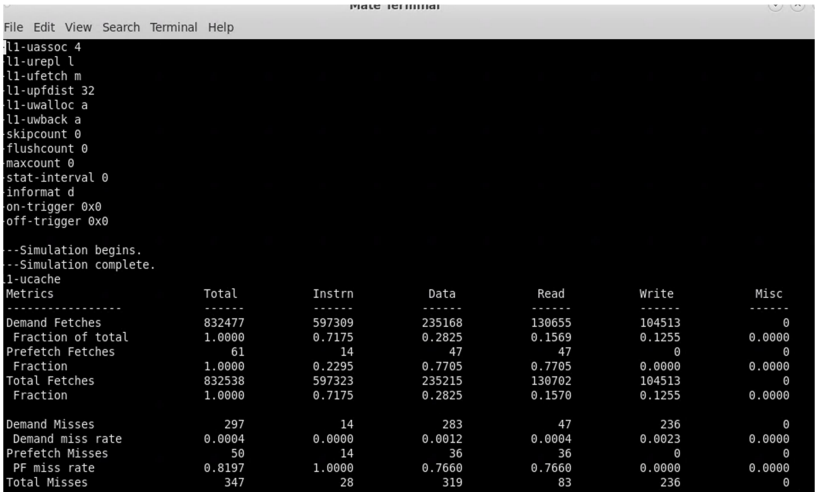


Figure 1. A sample of simulation output by running dineroIV

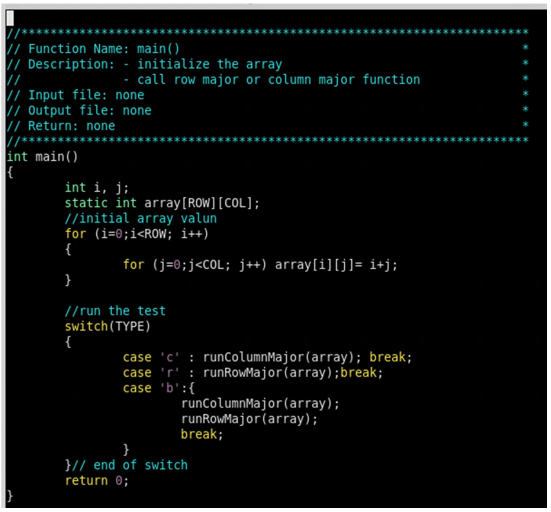
Problem 2

Write a program as such to detect column-major or row-major memory layout for 2-dimentional array in your system. The suggestion is to not use compiler optimization. You can use the example in power point slide. Make sure to run the loop for enough number of iterations. For performance evaluation, use gprof. Also use Pin to analyze the effect of column-major and row-major on the cache performance (cache hit rate). Report your results and analysis (30pts).

Answer:

1) In the program named loop_yu.c , I set up the 10000 x 10000 array, and run both column-major and row-major functions (in loop_yu.c, set TYPE ‘b’, which means both, details please refer README.pdf in file package) . The program screen-shot refers the Figure 2a. I complied with prof and run the program, then got the result as what was presented in Figure 2b, it shows the running time of main(), runColumnMajor and runRowMajor function, which are 0.56s, 1.14s and 0.26s. The running time of main() is of initializing the array.

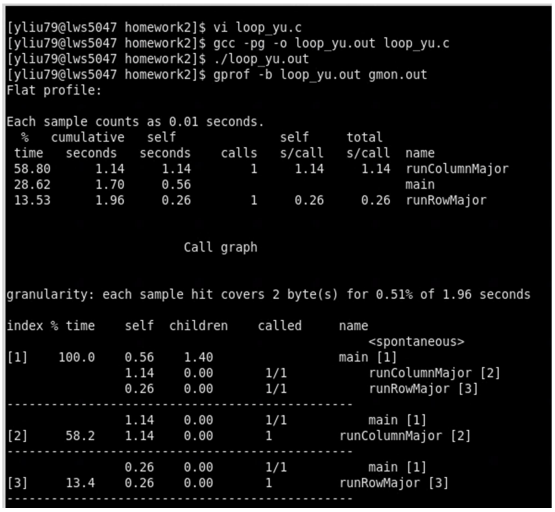
When compare with the running time of runColumnMajor and runRowMajor , it shows obviously that the first one consumes much more time, reach to five times more than the latter. This is because the column-major loops method cause much higher cache miss rate of data access, due to long stride given by the index in inner loop. When the loop changes to row-major method, the stride change to 1, which means the data access is in a sequential order, so the miss rate drops down significantly.



```
//*********************************************************************
// Function Name: main()                                              *
// Description: - initialize the array                                *
//              - call row major or column major function             *
// Input file: none                                                   *
// Output file: none                                                  *
// Return: none                                                       *
//*********************************************************************
int main()
{
        int i, j;
        static int array[ROW][COL];
        //initial array valun
        for (i=0;i<ROW; i++)
        {
                for (j=0;j<COL; j++) array[i][j]= i+j;
        }

        //run the test
        switch(TYPE)
        {
                case 'c' : runColumnMajor(array); break;
                case 'r' : runRowMajor(array);break;
                case 'b':{
                        runColumnMajor(array);
                        runRowMajor(array);
                        break;
                }
        }/// end of switch
        return 0;
}
```

Figure 2a. part of program source code



```
[yliu79@lws5047 homework2]$ vi loop_yu.c
[yliu79@lws5047 homework2]$ gcc -pg -o loop_yu.out loop_yu.c
[yliu79@lws5047 homework2]$ ./loop_yu.out
[yliu79@lws5047 homework2]$ gprof -b loop_yu.out gmon.out
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 58.80      1.14     1.14        1     1.14     1.14  runColumnMajor
 28.62      1.70     0.56                             main
 13.53      1.96     0.26        1     0.26     0.26  runRowMajor


                        Call graph


granularity: each sample hit covers 2 byte(s) for 0.51% of 1.96 seconds

index % time    self  children    called     name
                                                 <spontaneous>
[1]    100.0    0.56    1.40                 main [1]
                1.14    0.00       1/1           runColumnMajor [2]
                0.26    0.00       1/1           runRowMajor [3]
-----------------------------------------------
                1.14    0.00       1/1           main [1]
[2]     58.2    1.14    0.00       1         runColumnMajor [2]
-----------------------------------------------
                0.26    0.00       1/1           main [1]
[3]     13.4    0.26    0.00       1         runRowMajor [3]
-----------------------------------------------
```

Figure 2b. gprof simulation shows row-major loop is more effective

2) I also use Pin to analyze the column-major (in loop_yu2.c, set TYPE 'c') and row-major function (in loop_yu2.c, set TYPE 'r'). The result shows the same conclusion as what is analyzed in gprof: in column major mode, the miss rate is much more higher than what is in the row major, which is 11.12% versus 1.39%. Refer the Table 3, raw output data is in the file of pin_loop.pdf in package of "HW2_prefetcher_yu.zip".

	Colum Major	Row Major
Instruction Hit	2000156973	2000156974
Instruction Miss	1131	1130
DataLoad Hit	799953188	887538637
DataLoad Miss	**100099886**	**12514437**
DataStore Hit	200027751	200027750
DataStore Miss	4521	4522
Data Hit	999980939	1087566387
Data Miss	100104407	12518959
Total Hit	3000137912	3087723361
Total Miss	100105538	12520089
Instruction Total	2000158104	2000158104
Data Total	1100085346	1100085346
Total	3100243450	3100243450
DataLoad miss rate	**11.12%**	**1.39%**
Hit Total	96.77%	99.60%
Miss Total	3.23%	0.40%

Table 3. In Pin simulation, data Load miss rate of Colum-Major loops reaches to 11.12%

Problem 3

Use Pin to create memory trace for Dhrystone and Linpack benchmarks. PinTools provides the instruction of generating memory trace. Use your cache simulator (developed in homework 1) to create the following cache model:

Data cache size: 16KB

Cache block size: 32B

Associativity: 4-way

Then, develop a stride prefetcher module next to the cache, with the following configurations:

Prefetcher buffer size: 500B, 1KB, 2KB, 4KB

Prefetching confidence bits:

- 2 (prefetcher prefetches confidence values equal or bigger than 2)
- 3 (prefetcher prefetches confidence values equal and bigger than 4).

Note: Number of Prefetcher controller entries is equal to Prefetcher buffer size divided by Cache block size.

Use your own desired interval to keep an unused prefetched block in prefetcher buffer.

Evaluate the performance of the cache without prefetcher and with prefetcher with various prefetcher buffer sizes. Can you elaborate and identify the percentage of compulsory misses covered by prefetcher? (50pts)

Answer:

The program source code please check what in the package of "HW2_prefetcher_yu.zip", which includes source code, data source and README file.

After running the pin code, there are totally got 5,044,119 data items in Dhrystone.out and 4,672,663 data items in Linpack,out. Figure 3 shows the result of the prefetch simulation, which are miss numbers and miss rate of the two benchmarks. We can draw some conclusion as below:

Firstly, when we compare the different result with different configuration of the prefetcher, it shows:

- A prefetch with higher buffer size improves the prefetch performance(hit number and rate drop down) , but the improve trend will be flat when buffer size reach 1K size.
- Because Dhrystone focus on integer benchmark, while Linpack focus on float benchmark, the miss number of Linpack is higher, due to more out-of-order sequence.
- It also shows that, comparatively, confidence 2 is more suitable for Dhrystone, which get less miss number and rate. The reason is the data are more regular, which says the history data pattern is more regular. While confidence 4 is more fit to Linpack because of relatively out-of-order history data pattern.

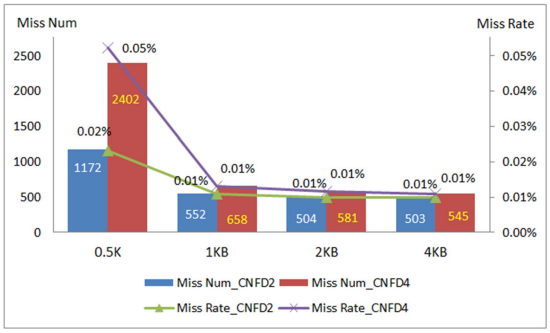


Figure 3a. Prefetch miss number and rate of Dhrystone

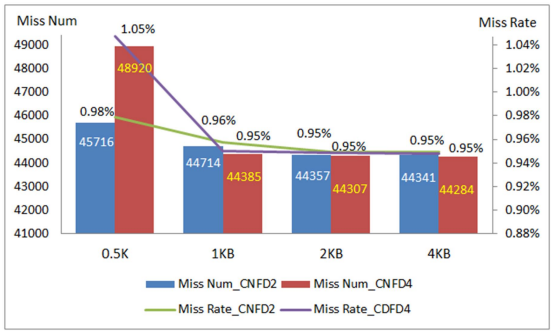
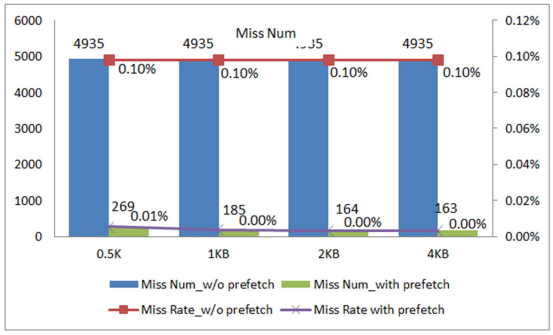


Figure 3b. Prefetch miss number and rate of Linpack

Secondly, comparing the miss number and rate from cache with prefetcher and without prefetcher, it indicates

obviously that the cache total hit performance improves a lot. For Dhrystone benchmark , the miss rate drops down from 0.10% to 0.01%; For Linpack benchmark, the miss rate drops from 0.34% to 0.24~0.27%. The data shows in Figure 4a and Figure 4b.



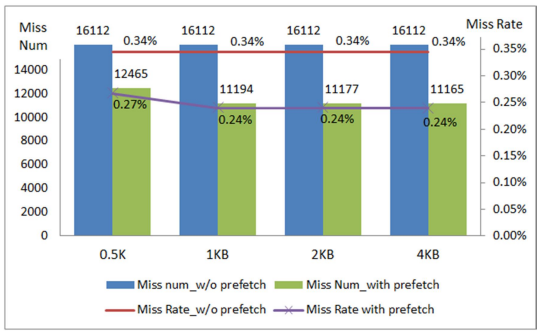


Figure 4a. Cache miss number and rate comparison of with prefecher and withour prefecher in Dhrystone

Figure 4b. Cache miss number and rate comparison of with prefecher and withour prefecher in Linpack