Implementing and comparing the in-order and out-of-order instruction execution for a two-level on-chip cache optimized for matrix multiplication operation

Cache Masters

Arpitha Nagaraj Hegde 26601584 | Mayank Gupta 48998256 | Rajandeep Singh 41917909 | Venkata Adithya Nimmagadda 51169601

Abstract--In this project, we have implemented two level cache for in-order and out-of-order execution and studied the effects of various parameters such as miss rate, cycles per instruction, etc.

Section 1

1.1 Introduction

Multi-level caching was invented to close the growing gap between the main memory access time and fast clock rates of modern processors. A two-level cache structure allows the primary cache to focus on minimizing hit time to yield a shorter clock cycle or fewer pipeline stages, while allowing the secondary cache to focus on miss rate to reduce the penalty of long memory access times.

Out-of-order execution is built into the high performance multi core processors like Intel-i7. The out-of-order-execution is implemented to reduce the effect of a cache miss on performance when for pipelined computers. If the instructions in the pipeline are stalled as one instruction got a cache miss in fetch cycle and if the later instruction in the pipeline has fetched the operands with a cache hit, these later instructions can be executed first.

1.2 Background

Simplescalar is an open source computer architecture Simulator software, that can compare the performance of different machines by simulating the performance of each according to various parameters such as cache size, block size and associativity to name a few.

1.3 Benchmarks

For this purpose, our benchmark consists of Matrix Multiplication to assess which combination would be best suited. It has wide applications ranging from image processing, convolution neural networks and many other digital signal processing techniques. Along with it being an important part of the application, it is also one of the most computationally intensive part of the process. Optimizations related to these multiplications will help in all the above applications by reducing running times.

Each of the Matrix is 64X64 2-D matrices of integer values of 64 bits(8 bytes).

Matrices play an important role in data sciences as well image processing.

These are the three major types of matrices:

1) Complete Matrix: This is the general matrix we see in form of datapoints, linear equations etc. It has a higher no of non zero elements compared to zero elements and computationally this the most expensive matrix.
2) Diagonal Matrix: A diagonal matrix is defined as one that has all the elements along a diagonal to be zero. In SVD in Machine Learning a Diagonal Matrix is one of the foundational matrices used. Computationally this is easier than the full matrix
3) Sparse Matrix: It is a matrix that has a higher number of non-zero elements. It is used widely in information retrieval algorithms as well as graphics processing. Computationally this is the less expensive.

Due to C not being an object oriented language, we could not find a matrix multiplication

Section 2

2.1 Implementation and Parameters

For this project we use the -sim cache and the -simoutorder for simulating an inorder and out of order execution.

We varied the four fields of the cache configuration for DL1 data level 1 cache and DL2 data level 2 cache:

<name>:<nsets>:<bsize>:<assoc>:<repl>

<*nsets*> Number of sets in the cache
<*bsize*> Block size
<*assoc*> Associativity (power of two)
<*repl*> Replacement policy (l | f | r), where l = LRU, f = FIFO, r = random replacement

Each of the benchmarks that we've chosen have been simulated for various cache configurations.

We varied the cache configuration for DL1 cache while keeping the cache configuration constant for DL2 cache and generated an output file.

Similarly, we generated another output file by varying the cache configuration for DL2 cache while keeping the cache configuration constant for DL1 cache.

From the output files generated, we selected the configuration which provided the least miss rate and used it as a baseline to study the effect of individual parameters on the two levels of cache.

Based on the optimal configuration for DL1 cache and DL2 cache for in-order execution, we implemented a two-level cache for out-of-order execution.

2.2 In-order execution

The instructions are fetched, executed and completed in compiler-generated order in case of in-order execution. A comparative study of the miss rates of level 1 data cache and level 2 data cache has been performed for in-order execution using the **sim-cache** simulator on the matrix multiplication benchmarks.

2.2.1 DL1 cache

We varied the associativity from 1 to 32 by keeping the cache size and block size constant. It could be seen that as the associativity increases, the miss rate starts to decrease. No performance gain was observed for associativity more than 4.

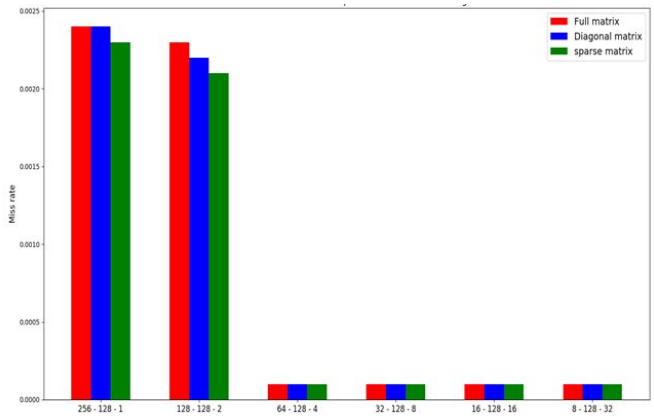

Fig 2.1 Associativity for DL1 cache

We varied the block size from 16-128 bytes for a constant cache size of 32 KB. As the block size increases, the miss rate decreases.

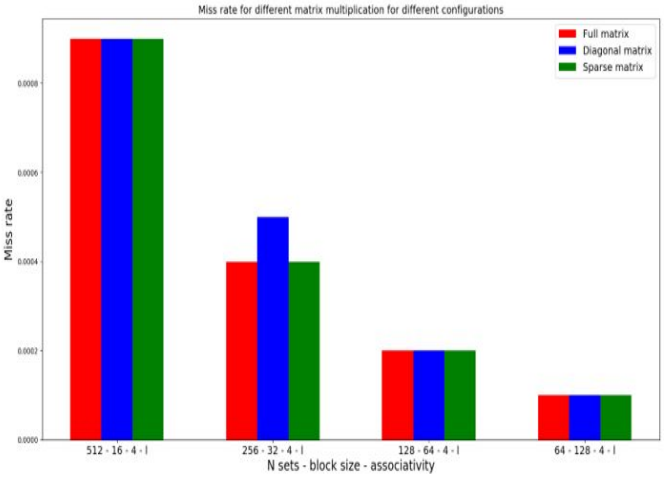

Fig 2.2 Block size for DL1 cache

Cache size was varied by varying the number of sets while keeping the associativity and block size constant.As the cache size increases, miss rate decreases and then remains constant for cache sizes greater than 32 KB.

Number of sets and block size are kept constant; the replacement policies are varied along with associativity. LRU replacement policy was found to be the ideal policy for cache optimization in case of matrix multiplication. Since LRU is based on the principle of temporal locality, it is a good replacement policy for programs such as multiplication.

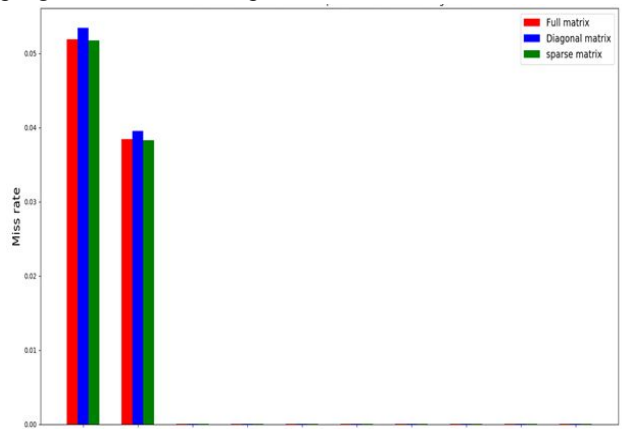

Fig 2.3 Cache size for DL1 cache

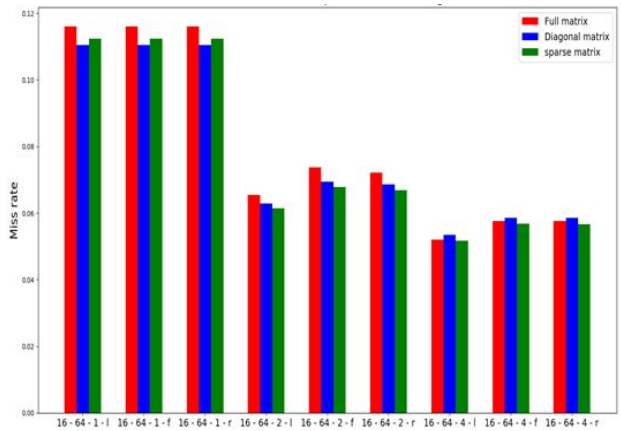

Fig 2.4 Cache Replacement Policies for DL1

2.2.2 DL2 cache:

Cache size is kept constant at 128 KB. From our observation, the miss rate for L2 cache is minimum for n way associative cache. Associativity is kept constant and the block size is varied for a DL2 cache. The miss rate decreases as the block size increases. For a DL2 cache, as the cache size increases, the miss rate decreases.The replacement policies are varied along with associativity. From the results obtained, we could see that the miss rate remained independent of the cache replacement policy for a DL2 cache.

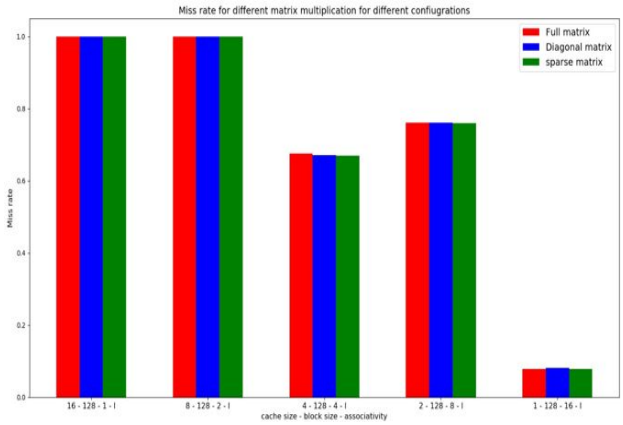

Fig 2.4 Associativity for DL2 cache

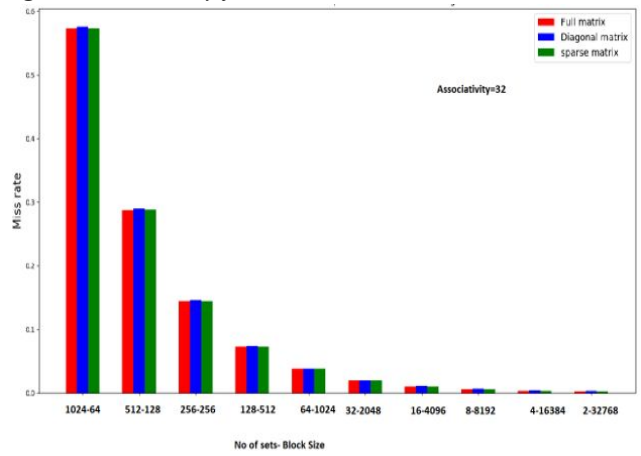

Fig 2.5 Block size for DL2 cache

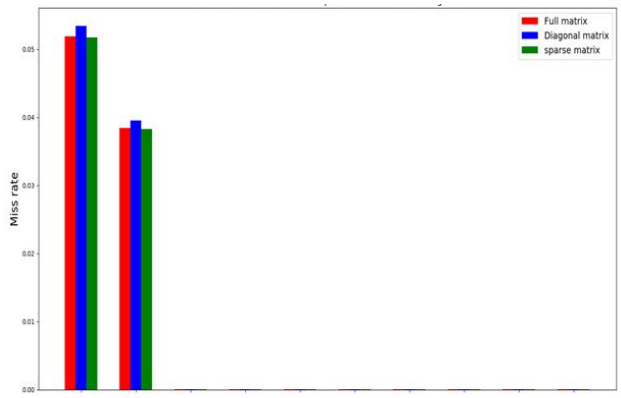

Fig 2.6 Cache size for DL2 cache

2.3 Out of Order execution

In a standard pipeline, the instructions are statically scheduled and executed in program order. In case of any hazard, the pipeline would stall until the instruction that caused the hazard was executed. However, in case of out of order execution, the instructions would be issued in order and executed out-of-order regardless of their order in the program. Since we can reorder instructions, there is a possibility of occurrence of WAR and WAW hazards which was not a concern for an in-order pipeline.

We have used **sim-outorder** simulator which supports out-of-order issue and execution, based on the Register Update Unit. We have modified the following parameters of the cache, the processor core and branch prediction :

<name>:<nsets>:<bsize>:<assoc>:<repl> <ruu> <lsq> <bpred>

<*bpred*> Branch prediction is specified as taken or not taken

<*ruu*> Capacity of the RUU (in instructions). The default for RUU is 16

<*lsq*> Capacity of the load/store queue (in instructions). The default is 8.

After conducting experiments for the cache configuration for both DL1 cache and DL2 cache in case of in-order execution, we obtained the value 64:128:4=32KB for DL1 and 1:1024:128=128KB for DL2 to be the most optimum value (denoted by O).We have confirmed the results by comparing these values with the default configuration in out of order dl1:256:32:1 and dl2:1024:64:4 (denoted by N) in out of order execution. Here, we also include another performance parameter i.e. Clocks per instructions or CPI.

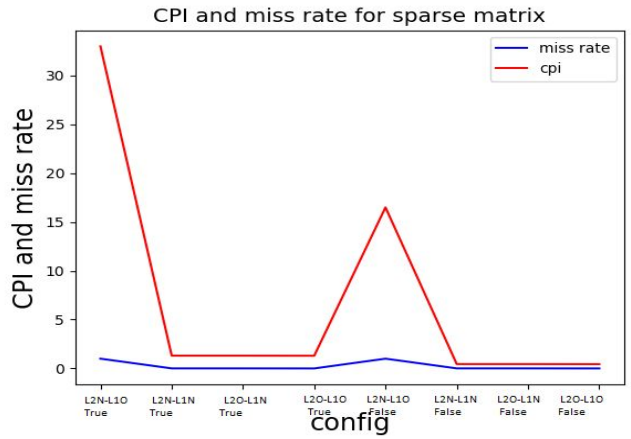

Fig 2.7 Comparison between O and N

2.3.2 Analysing CPI

Now we find out that the least miss rate and lowest CPI were obtained for the case when we used our optimized values in both the dl1 and dl2 caches.

This verifies our findings obtained in the first part of this project.

2.3.3 RUU

After verifying that our findings from in-order execution hold true in out of order execution, we now move to changing the values of the Register Update Unit(RUU).

The RUU in the simplescalar combines the reorder buffer as well as the register station. The diagram shows the process flow for the same.

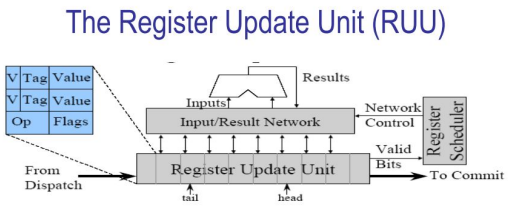

Fig 2.8 Flow of an instruction through the Register Update Unit(RUU)

The default value for the register update unit is 16. We vary it to find that the best value saturates after having 32 units in the RUU.

2.3.4 Wrong Path

This execution has only the flag true or false. The default flag is true.In out of order execution, when there is speculation and mispredictions (i.e. when the instructions are executed following a mispredicted branch before the branch has been resolved).

We can see that for the matrix multiplication benchmark, speculative prediction does not seem to have any effect on the CPI.

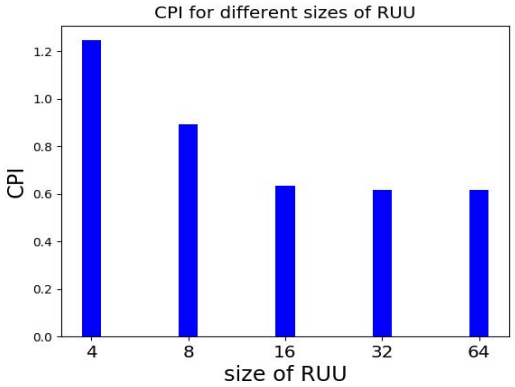

Fig 2.9 CPI for different sizes of RUU

2.3.5 Load/Store Queue(LSQ)

Finally we move to the Load/Store Queue(LSQ). Address translation happens in the RUU but LSQ identifies resolves any memory dependencies.The key feature is that loads happen out of order.

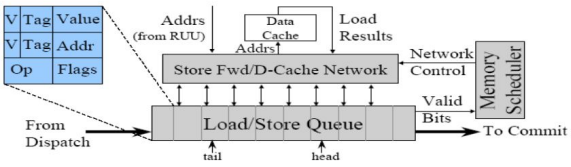

Fig 2.10 Flow through a load/store queue(LSQ)

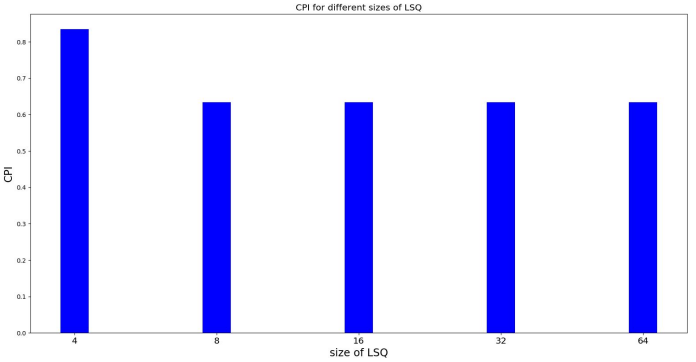

Fig 2.11 CPI for different sizes of LSQ

The default value for the load store unit is set to 8. We find that the optimum value saturates after 16.

Conclusion

Out-of-order execution has less CPI when compared to a processor which has only cache optimizations(i.e., in-order execution) as shown in Fig 2.12 Although, the default configuration of the cache and the processor core parameters has a lower CPI when compared to the rest of the configurations(i.e,), has a much higher miss rate as shown in Fig 2.13

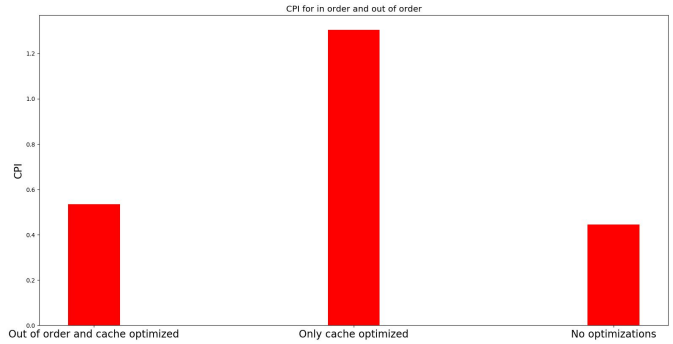

Fig 2.12 CPI for in-order and out of order execution

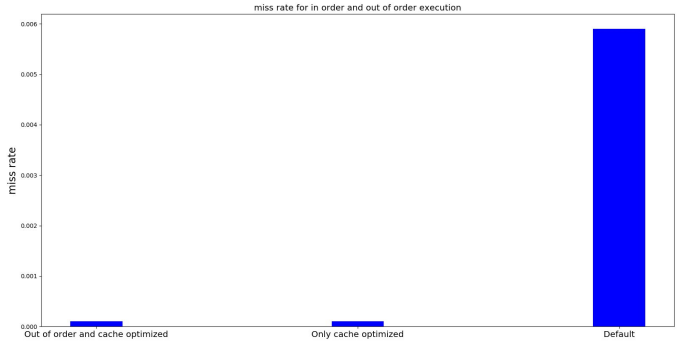

Fig 2.13 Miss rate for in order and out of order execution

The remaining benchmarks test-math and test-lswlr also follow a similar trend when CPI and miss rates for different configurations are compared. It can be inferred from Table 1 that the miss rate remains same for all the three matrix multiplication benchmarks and has different values but stays constant for other two benchmarks. For all the benchmarks, the CPI for an out of order execution remains lesser when compared to an in-order execution.

Benchmark	*Miss rate*		*CPI*	
	Out-of-order and cache optimized	*Cache optimized*	*Out-of-order and cache optimized*	*Cache optimized*
Complete matrix	*0.0001*	*0.0001*	*0.5608*	*1.3011*
Diagonal matrix	*0.0001*	*0.0001*	*0.5643*	*1.3013*
Sparse matrix	*0.0001*	*0.0001*	*0.5353*	*1.3041*
test-fmath	*0.0042*	*0.0042*	*1.0399*	*1.2179*
test-lswlr	*0.0294*	*0.0294*	*2.5325*	*3.0349*

Table 1 Comparison of miss rates and CPI for different benchmarks

Now, having analysed the different parameters in out of order execution as well, we are finally able to piece together our best optimized system which is as follow**s**

dl1:64:128:4:l dl2:1:1024:128:l -ruu:size 32 -wrong path {any} -lsq:size 16

Finally, we are able to conclude the following:

1) For minimizing the miss rate:
    a. 32KB and 128KB are the best for dl1 and dl2 respectively.
    a. Keeping the associativity at 4 and 8 for dl1 and dl2 respectively.
    b. The LRU is the best replacement policy.
2) This gives us the best optimized cache.Further, in out of order execution,the best values to reduce the CPI are obtained by keeping
    a. RUU at 32
    b. LSQ at 16

References

[1]https://cs.nju.edu.cn/swang/CA_16S/simplescalar_tutorial.pdf
[2]https://web.eecs.umich.edu/~taustin/papers/UWTR97-simple.pdf