Gopalakrishnan Swetha (A0070601X)
Elias Boutaleb (A0128072L)

CS4223 Assignment 2

1 INTRODUCTION

Goal: Simulate uniprocessor and multiprocessor caches with MSI and MESI cache coherence protocols. Find the optimal cache configuration for FFT and WEATHER benchmarks.

Simulation Environment:

1. Java Programming Language, Version 1.8
2. Eclipse IDE

Assumptions made:

1. A cache waiting for a flush gets the data only after 10 cycles
2. Each cache can only have 1 pending instruction in the Bus Request Queue
3. Cache hit includes cases where the cache block was found but a bus transaction still needs to be generated
4. Caches carry out snooping on bus transactions by other caches
5. Fetch instructions can still be carried out though processor is blocked for read/write operations

2 IMPLEMENTATION

1. Design

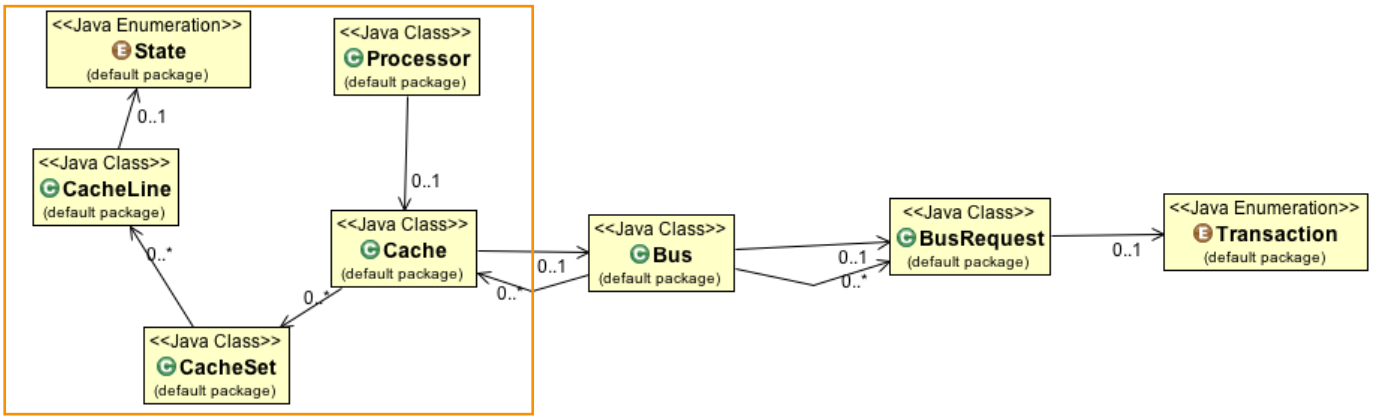
Classes:

The following table shows the structure for caches, processors and the bus implemented.

Entity	Variables	Remarks
Processor	• Int processor_id • Cache proc_cache • Int cycle_count • Int[] pending_instruction	• Each processor reads its own trace file • Each processor keeps track of its cycle count • It is responsible for asking the cache to execute the instructions and the cache replies if it is able to process instructions at the moment
Cache	• int cache_id • int cache_size • int block_size • int associativity • int countCacheMiss • int countCacheHit • Bus bus (access to bus) • CacheSet[] cache_sets (each	• Each cache has sets • It keeps track of number hits and misses • Each cache has access to the bus • Each cache also has a function to carry out snooping and this is called by the bus • On a cache miss, the cache enqueues a BusRequest into the BusRequest Queue

	cache has sets) • bool pending_bus_request (to accommodate assumption 2) • bool uniproc_flag (is this a uniprocessor)	
Cache block	• int address • State state • Int tag • Int LRU_age (to carry out LRU policy) • String protocol	• Each of the elements in the cache_sets array in each cache will contain block(s). • State is an enum value that can be MODIFIED, EXCLUSIVE, SHARED, INVALID • Valid and dirty bits are not needed here because they can be known from the state of the block
Bus	• ArrayList<Cache> caches • State state • Queue<BusRequest> requests • BusRequest current_request • Int bus_traffic	• The bus has access to all the caches • It also has a queue to implement the first come first serve arbitration policy • The bus keeps track of how many bytes of traffic it has. The amount of traffic is incremented by the amount of a block size
Bus Request	• Int cache_id (cache that made request) • Transaction transaction • Int address • Int cycles_left	• There can be different types of transaction including BusRd, BusRdX and BusWB • The number of cycles left is decremented in each clock cycle so the cache can easily check if its instruction has been completed.

The class diagram below shows the interaction between classes. As we can see, the orange box shows that the simulation follows the structure of a cache. A processor has one cache, which has zero or more sets, which has zero or more blocks, which have one state each.



Calculation:

The processor reads a line from the trace file and extracts the label and address. If it is a PrRd or PrWr instruction, it passes this instruction to the cache. The cache extracts the following information from the address:

1. No. offset bits = lg (block_size) / lg(2)
2. No. index bits = lg (no. blocks) / lg (2)
3. No. tag bits = 32 – no. offset bits – no. index bits
4. For each address, the tag appears first, followed by the index followed by the offset. We then extract these 3 information from the address since we know the number of bits

The index bits are used to know which set to access. A cache hit occurs when one of the blocks in the set matches the tag of the address and is valid. If the address is not in the cache, it is a cache miss. If the cache block is in invalid state, it is also a cache miss.

Logic:

The pseudo code for the while loop is as follows:

while(true){

for each processor {

if the processor is not blocked, read a new line and send to cache if it is a read or write

if the processor is blocked, don't read a new line and send the pending instruction waiting to be processed to the cache

cache checks if it is a cache hit or miss

if miss {

if there is already a pending instruction in the bus queue, block processor.

If there is not a pending instruction in the bus queue, enqueue request into bus.

Run LRU policy

}

}

Bus processes request in queue

Caches snoop on bus transaction and change their state is needed. If the instruction being processed currently is a cache's instruction, check if the instruction is complete and unblock the processor

If the trace file is complete, break out of loop

Print cache content and statistics to log file

}

2. Results

The input values are:

- *"protocol"* is either MSI or MESI
- "*input_file*" is the input benchmark name (e.g., WEATHER)
- *"no processors":* number of processors
- "*cache_size*": cache size in bytes
- *"associativity":* associativity of the cache
- "*block_size*": block size in bytes

The output values are:

- Data cache miss rate
- Amount of Data traffic in bytes on the bus
- Execution cycles per processor

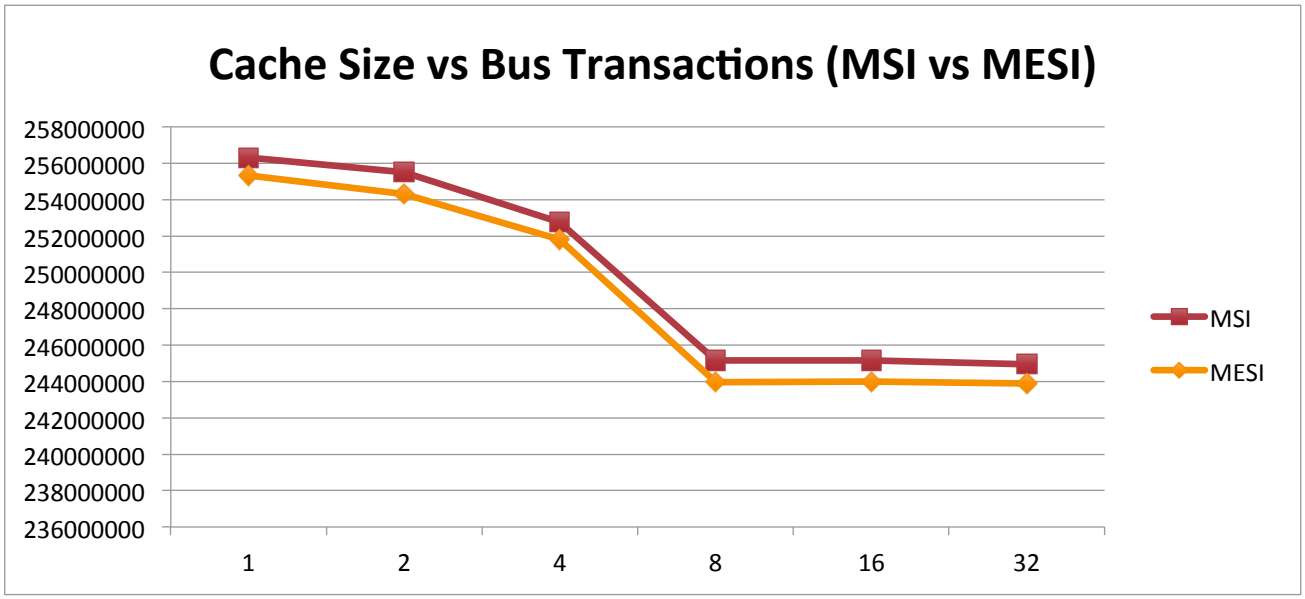
The variable inputs are:

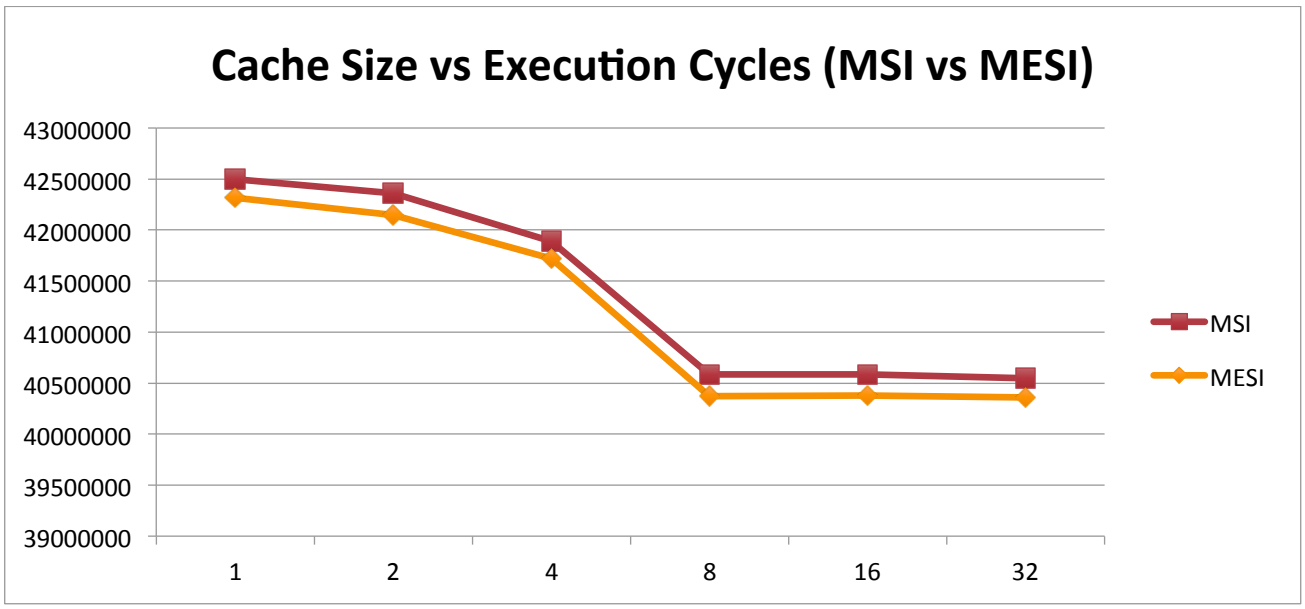
- Cache size (vary between 1KB and 32KB)
- Associativity (vary between 1, 2, 4)
- Number of processors (vary between 1, 2, 4, and 8)
- Block size (Vary between 8 byte and 128 bytes)

2.2.1 Basic Task

In the basic task, we are asked to compare MSI and MESI protocols. The improvement of MESI over MSI is the fact that in MESI, a cache in the E state does not need to generate any BusRdX transaction to move to the M state. However, in the MSI protocol, a cache will be in the S state and will always generate a BusRdX when it wants to move to the M state. Therefore, the improvement of MESI over MSI depends on how many times this transition from E to M states happens.

The following test was run using the default values of 4 processors, 16-bit word size, 64-byte block size, and direct-mapped cache per processor. The cache size is varied and for each cache size, the bus transactions (bytes) and number of execution cycles is measured.





We can see from the graphs that MESI is performing slightly better than MSI, in terms of bus transactions and execution cycles.

2.2.2 Advanced Task

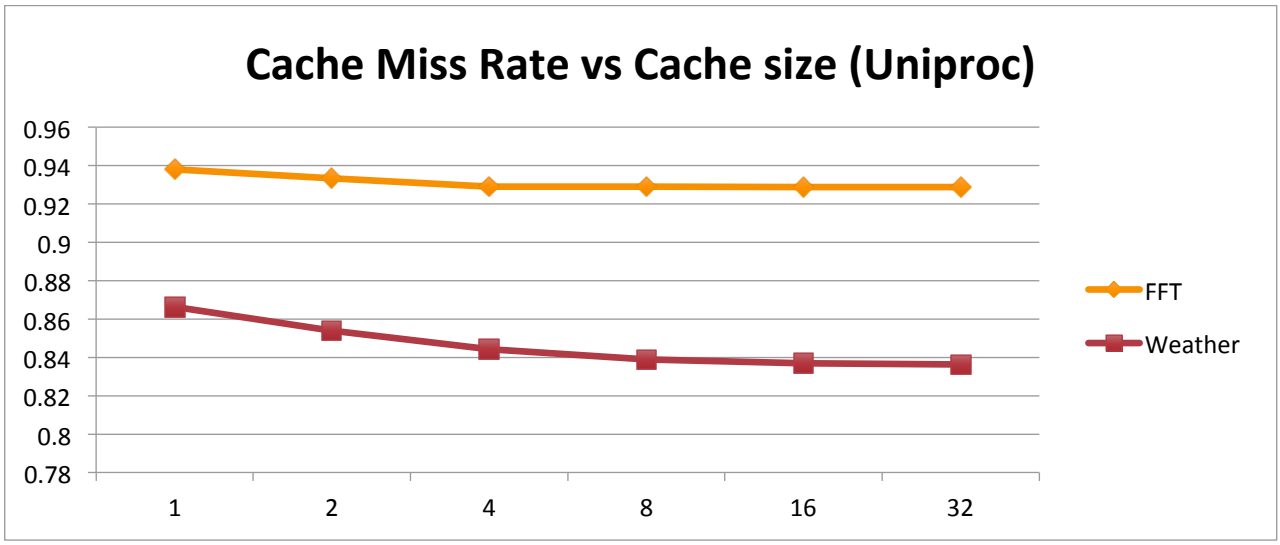
In the advanced task, we are asked to analyze the impact of varying cache size, block size, associativity and number of processors. By analyzing the impact, we come up the optimal configuration for each benchmark (FFT and WEATHER). We also implement an optimization to improve MESI, which will be explained later in the report.

2.2.2.1 Uniprocessor

The 2 given benchmarks, FFT and WEATHER are used to analyze the impact of the cache size, block size and associativity on performance. Performance refers to the cache miss rate. In the following graphs, we vary only one of the parameters and for that configuration, see how the WEATHER benchmark performs against the FFT benchmark.

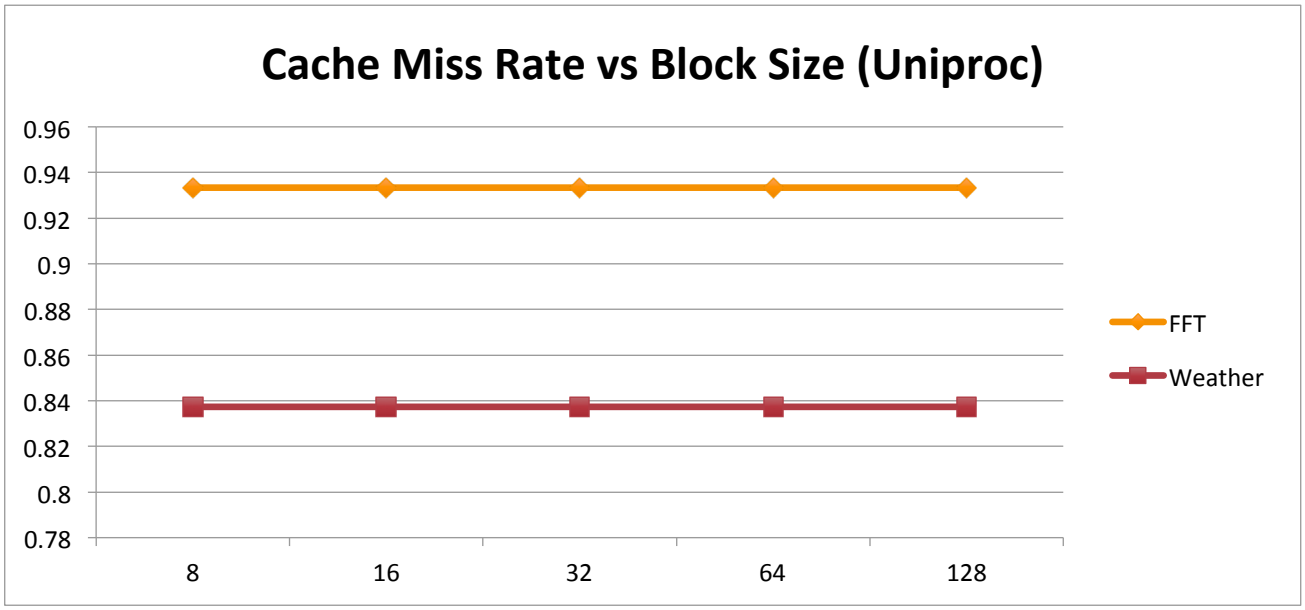
First, we vary the cache size from 1KB to 32KB. The other parameters are kept constant as follows:

- Block size = 16 bytes
- Associativity = 2



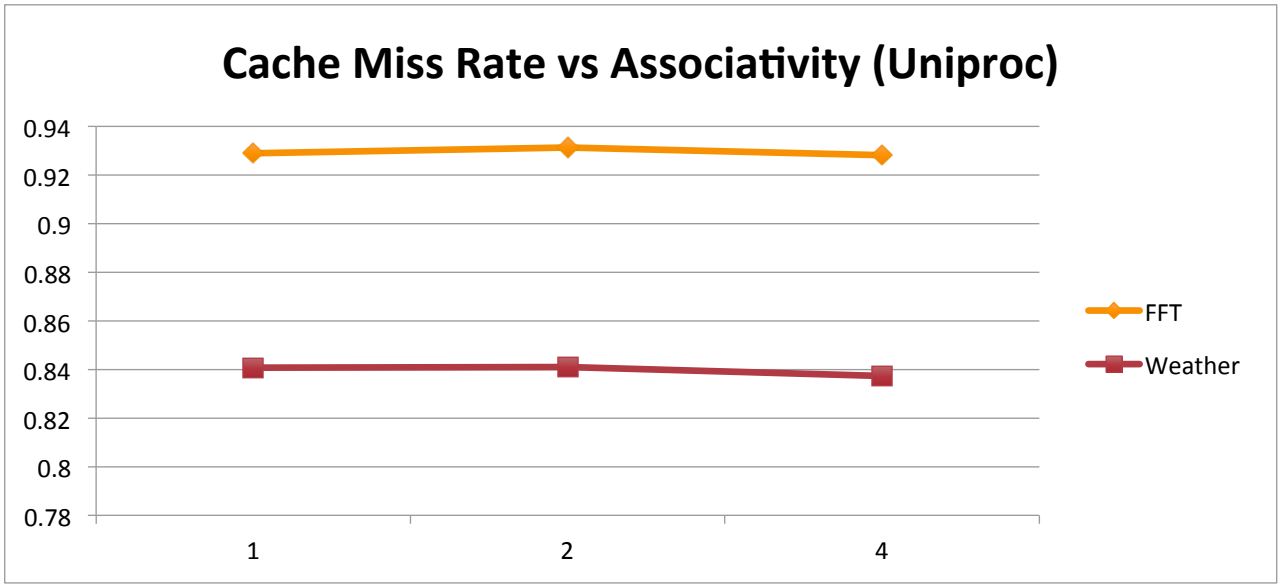
Next, we keep the cache size constant and vary the block size between 8 and 128 bytes. The other parameters that are constant are:

- Cache size = 4KB
- Associativity = 4



Next, we vary the associativity between 1 and 4. The other parameters are kept constant as follows:

- Cache size = 4KB
- Block size = 64 bytes



Based on the results, we can see that in general the performance of the WEATHER benchmark is better than FFT. The impact of cache size is that when cache size increases, the cache miss rate decreases. When block size increases, there is no impact on the cache miss rate. This is because though the block size increases, the performance is still limited by the cache size and therefore, the cache miss rate remained the same for all the tests. Increasing the associativity resulted in an overall decrease in the cache miss rate. This is because each set would have more blocks and multiple instructions with the same address can be put in the set without eviction and subsequent cache misses.

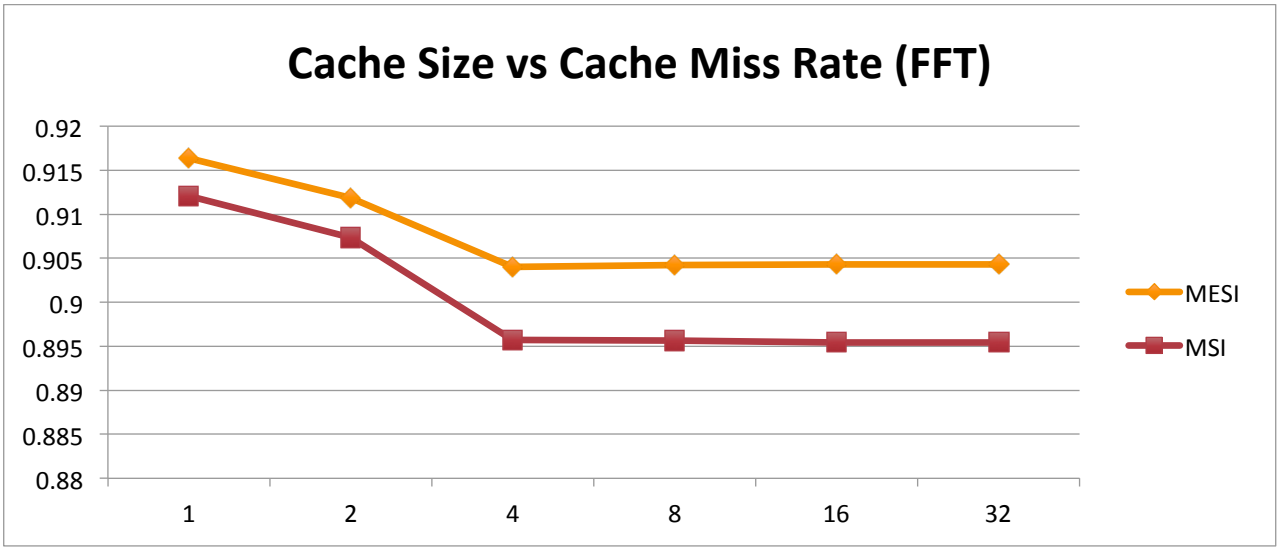
2.2.2.2 Multiprocessor

For the multiprocessor, we find out the optimal configuration for WEATHER and FFT. We first run tests for FFT and then WEATHER. For each benchmark, we vary cache size, block size or associativity and compare MSI and MESI. Afterwards, we test a configuration with varying number of processors. The performance is based on the cache miss rate.

FFT:

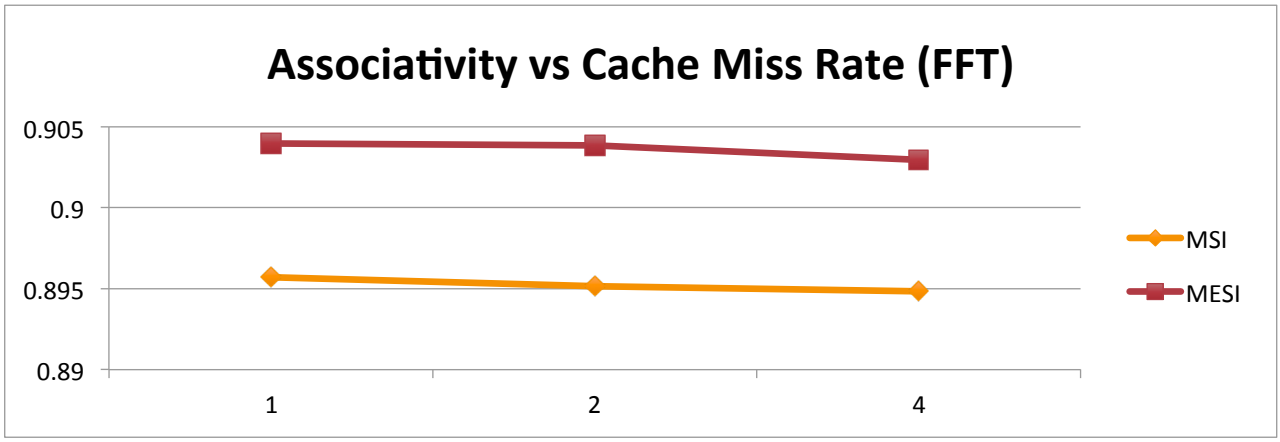
First, we vary the cache size from 1KB to 32KB. The other parameters were kept constant as follows:

- Block size = 16 bytes
- Associativity = 2
- No. Processors = 2



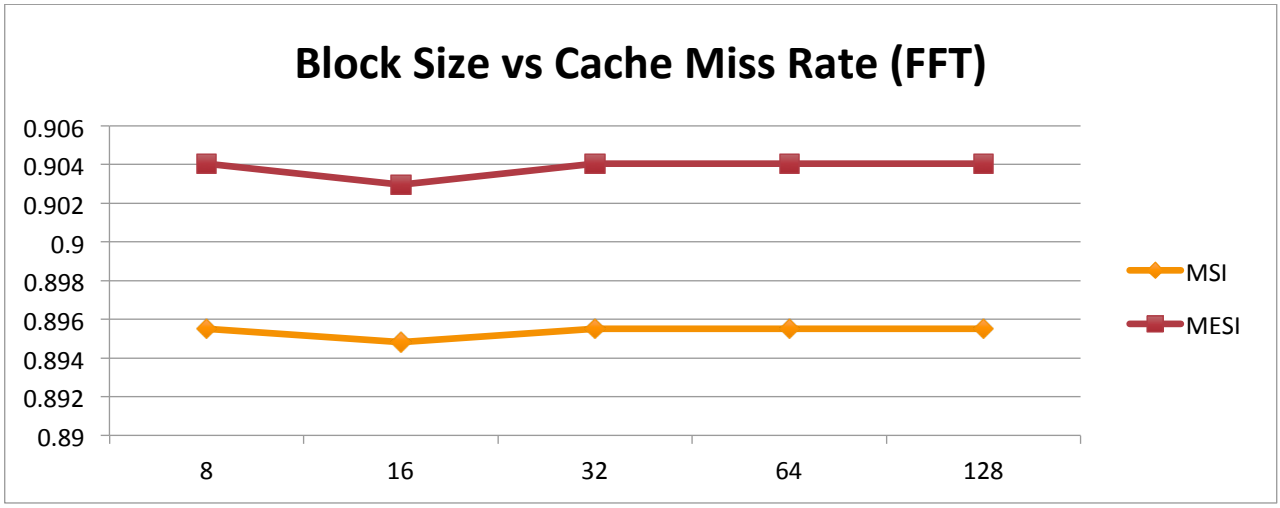
The lowest miss rate seems to be when the cache size is 4KB with little improvement after 4KB. Hence, for the next test, we vary the associativity with cache size 32KB and see if the cache miss rate can be lowered even more. The parameters kept constant for the next test were:

- Cache size = 32 Kb
- Block size = 16 bytes
- No. Processors = 2



The lowest miss rate seems to be when the associativity is 4. Hence, for the next test, we vary the block size with these parameters:

- Cache size = 32 Kb
- Associativity = 4
- No. Processors = 2



The block size does not seem to have any effect on the cache miss rate. The small drop seen is by 0.001 only. Hence, we conclude that a size of 8 bytes is sufficient.

Hence, the optimal cache configuration for FFT is as follows:

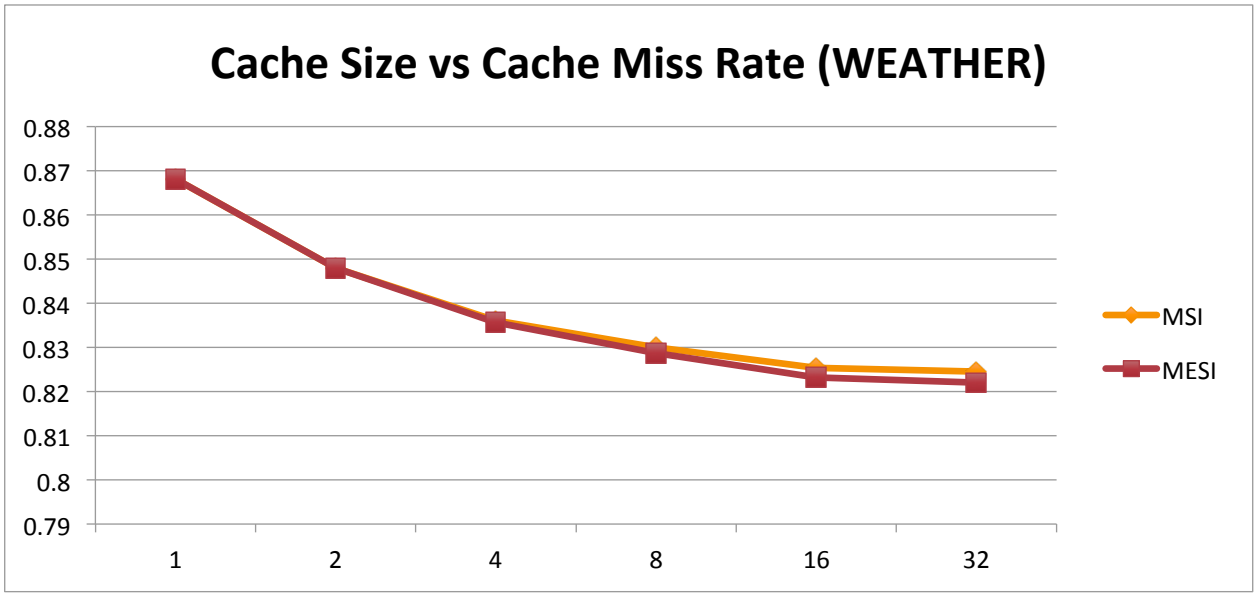
- Cache size = 32 Kb
- Block size = 8 bytes
- Associativity = 4

WEATHER:

First, we vary the cache size from 1KB to 32KB. The other parameters are kept constant as follows:

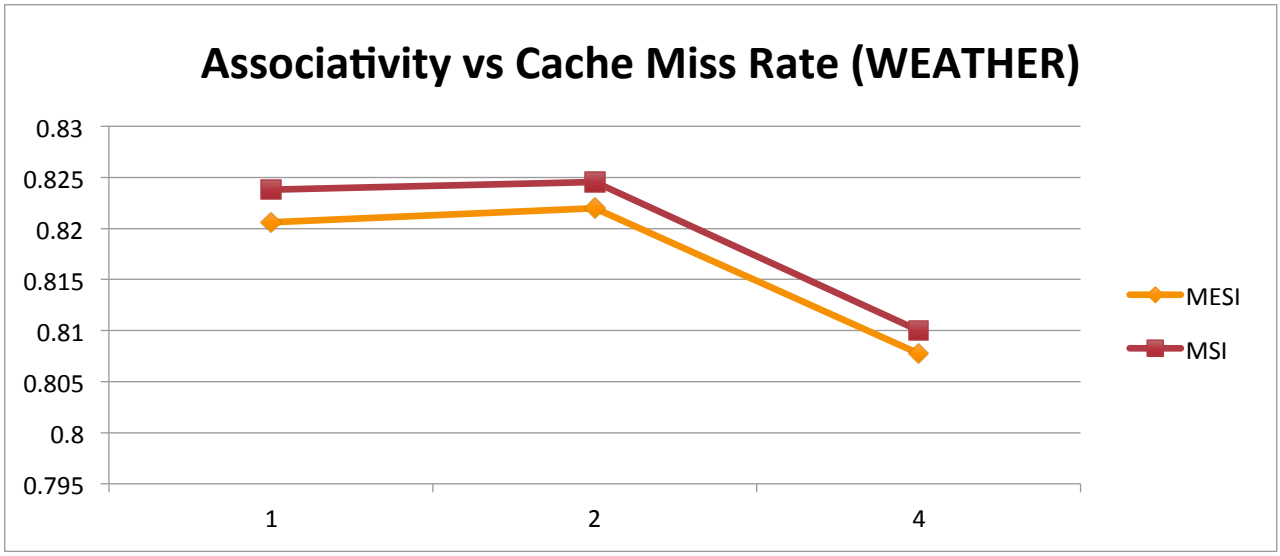
- Block size = 16 bytes

- Associativity = 2
- No. Processors = 2



The lowest miss rate seems to be when the cache size is 32Kb. Hence, for the next test, we vary the associativity and with these parameters:

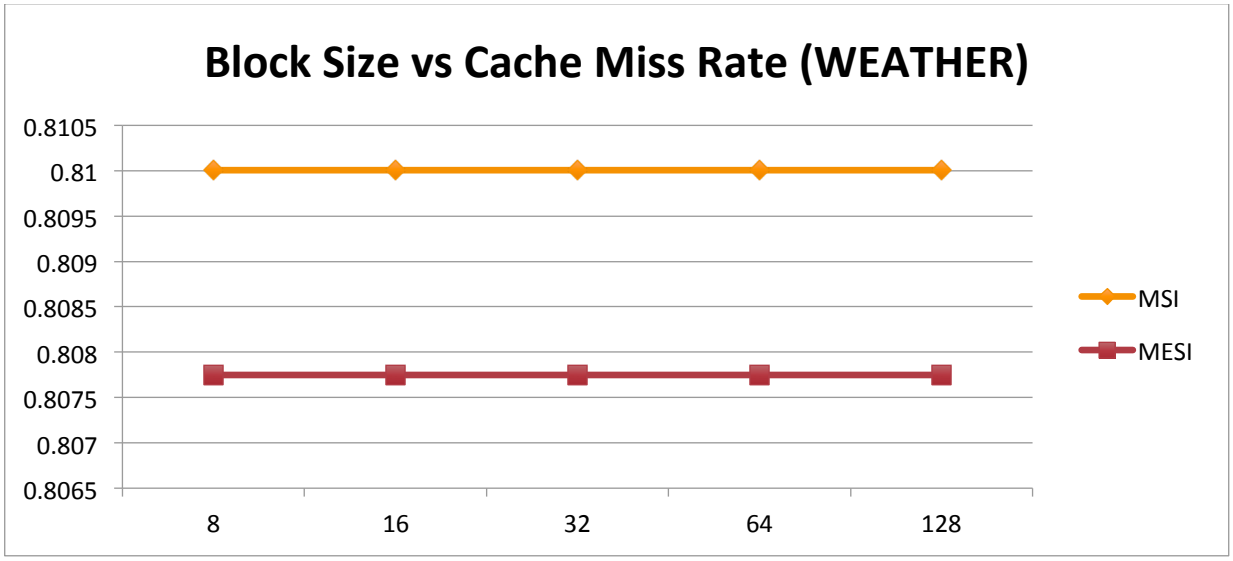
- Cache size = 32 Kb
- Block size = 16 bytes
- No. Processors = 2



The lowest miss rate seems to be when the associativity is 4. Hence, for the next test, we vary the block size with these parameters:

- Cache size = 32 Kb
- Associativity = 4

- No. Processors = 2

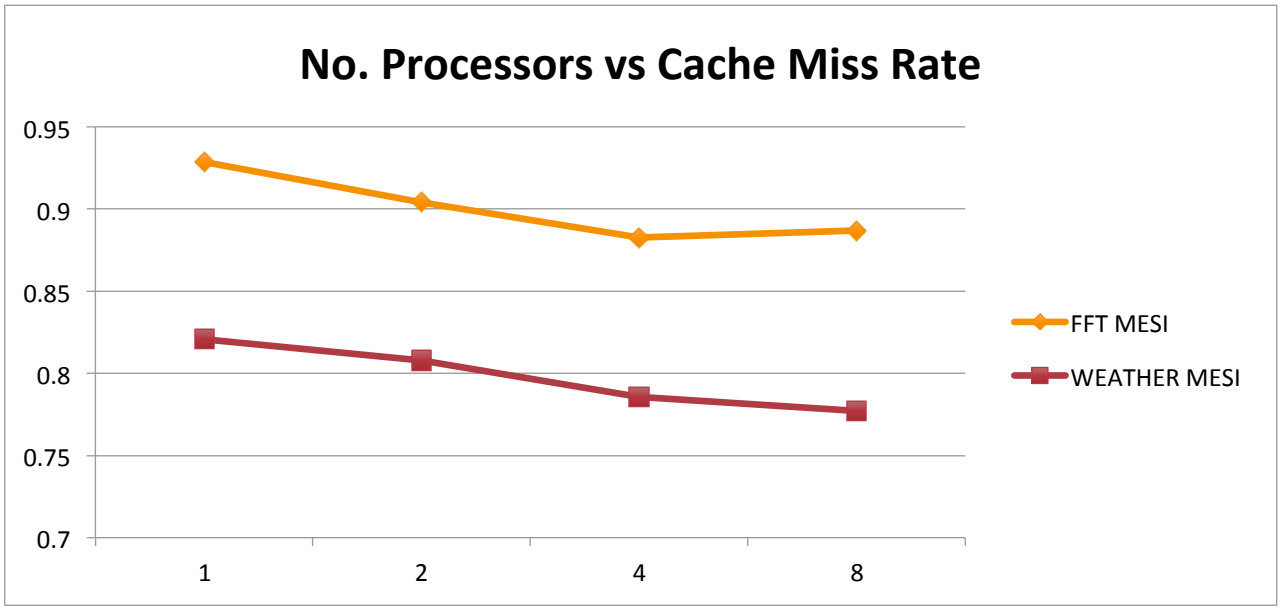


The block size does not seem to have any effect on the cache miss rate. Hence, for a lower cost implementation of the cache, we can simply take the lowest value of 8 bytes.

Hence, the optimal cache configuration for FFT is as follows:

- Cache size = 32 Kb
- Block size = 8 bytes
- Associativity = 4

We can see that the optimal configuration for both benchmarks is the same. Hence, we use this configuration to run tests for FFT and WEATHER for the MESI protocol to see how the number of execution cycles changes for different number of processors.

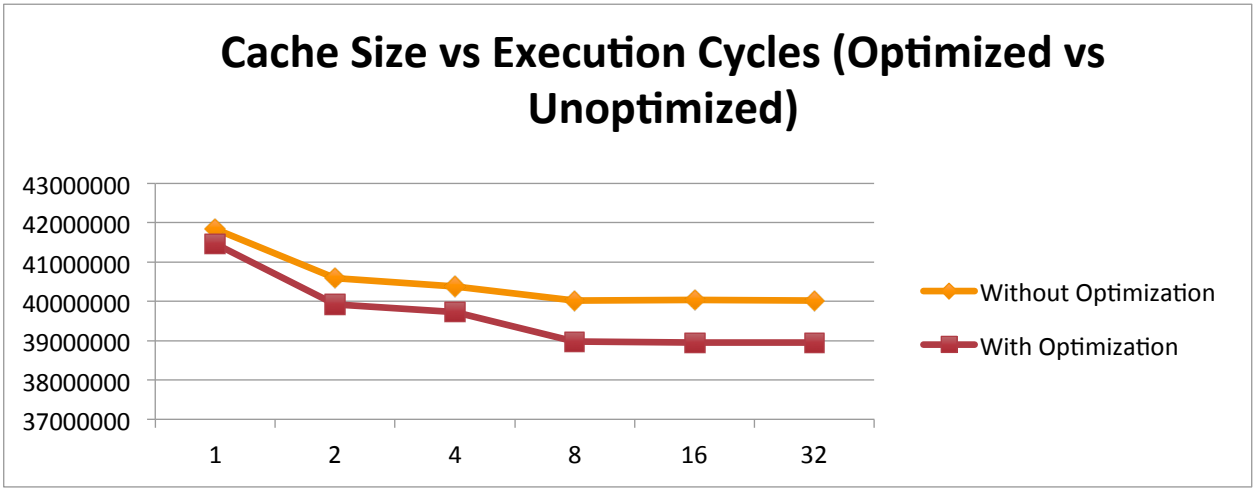


We can see that as the number of processors increases, the trend is that the cache miss rate goes down. However, the difference is not much. This is because the processors are still reading the same number of instructions and at the same time coherence needs to be maintained by even more processors.

2.2.2.3 Optimization

The optimization used to improve the MESI protocol was to add a new transaction called BusUpadte. Currently with the MESI invalidation protocol, when a block is in a Shared state and gets a PrWr, it will generate a BusRdX and go to Modified while invalidating the other caches' blocks. A BusRdX takes 10 cycles to process according to the assumptions of this project as the bus cannot differentiate between a BusRdX from an Invalid or Shared state. But with BusUpdate, the cache in the Shared state will now send out a BusUpdate to the other caches to let them know that it is going to write. The caches that receive this will invalidate themselves. According to the assumptions of this project, sending some data from one cache to another only takes 1 cycle. Therefore, the bus is now no longer blocked for 10 cycles but only 1 cycle. Hence, we expect to see a lower number of execution cycles for the optimized code. We ran the test for the optimized code for FFT with the following parameters:

- MESI
- Cache size = 32 KB
- Block size = 8 bytes
- Associativity = 4
- No. processors = 2
- Benchmark = FFT



As expected, the number of execution cycles decreased with the optimized code by adding BusUpdate.

PROBLEMS FACED

One of the biggest problems faced was that the cache miss rate was very high. How the code was debugged was:

1. Check that address calculations are correct.
2. Check that the place where countCacheMiss++ being done is correct
3. Print states of cache blocks at different points in time to ensure that they are consistent with how the protocol works. Eg. If there is a block in Shared state, ensure that there is another cache with the same block. If a block is in Modified, ensure that others are invalidated.
4. For MESI, ensure that blocks are going from E to M correctly.

From the checks, they were all correct. However, one observation that was made was that multiple processors get a PrWr on a particular address multiple times continuously. Therefore, the caches end up invalidating each other and end up with a cache miss every other time the same PrWr instruction is processed.

Another challenge faced was in debugging the code while not compromising the execution time. Debugging was tedious in that several log messages were required to analyze the situation, which made the execution time very slow. Another challenge was in the refactoring of the program design after making adding some assumptions and this led to some bugs.