ENEE 446 - Project 2 Report

By Calvin Leung

Contents

Implementation

Helper Functions:

I believe that it would be more beneficial for me to first explain the helper functions that I have implemented to provide functionalities for the two main functions: perform_access_load and perform_access_store.

remove_tail(int i):

This function takes a core ID as an input and remove the tail of the cache line from that core.

get_node(int i, int tag):

This function takes a core ID and tag ID as input.

This function returns the cache line that has the same tag ID from that core, returns NULL if that cache line does not exist in that core.

no_other_copies(int sender, int tag):

This function takes a sender (a core ID) and tag ID as input.

The sender refers to the core that is looking for the information. Since during a read miss, the core has to see if other cores hold a block that has the same tag.

This function returns FALSE if any of the other caches hold a block with the specified tag and is in state “shared”, “exclusive” or “modified”; return TRUE if otherwise.

bus_transaction(int transaction_type, int sender, int tag):

This function takes a transaction type, sender (a core ID) and tag ID as input.

The sender refers to the core that has broadcasted a message on the bus.

This function dispatches one of the 3 bus functions: bus_read_miss, bus_write_miss and bus_write_hit, depending on the transaction type.

Notice that the called function will affect all the core except the one that broadcasted the bus transaction.

bus_read_miss(int i, int tag):

This function takes a core ID and tag ID as input.

This function goes through every cache line that is in the specified core and does the following if the cache line has the same tag ID specified:

1. If the cache line is “exclusive”, it will be switched to state “shared”.
2. If the cache line is “modified”, it will write back and be switched to state “shared”.

bus_write_miss(int i, int tag):

This function takes a core ID and tag ID as input.

This function goes through every cache line that is in the specified core and does the following if the cache line has the same tag ID specified:

1. If the cache line is “modified”, it will be switched to state “invalid”.
   I assume that a write miss will simply send the modified data to the core that broadcasted the write miss, so the need for a write back is not needed
2. If the cache line is “exclusive, shared or invalid”, it will be switched to state “invalid”.

bus_write_hit(int i, int tag):

This function takes a core ID and tag ID as input.

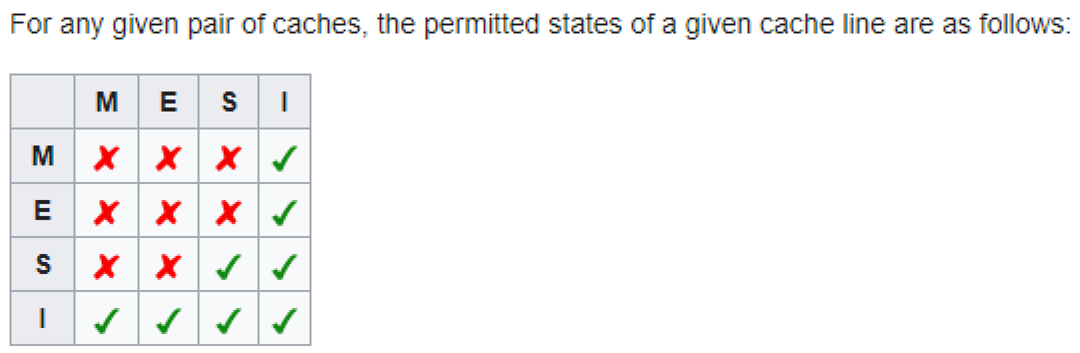
This function goes through every cache line that is in the specified core and does the following if the cache line has the same tag ID specified:

1. If the cache line is “shared”, it will be switched to state “invalid”.
   Since in a MESI cache, a write hit on the bus occurs only the all the cache that have the block tag is in state “shared”, so this only affects shared blocks.

print_consistent(int i, int tag): *function for debug use*

This function takes a tag ID as input.

This function goes through every cache in the system and look for cache lines that has the tag ID. The function has counter for shared, exclusive and modified. For every cache line that has the tag ID, it would increment that counter for its respective state.



For any given pair of caches, the permitted states of a given cache line are as follows:

	M	E	S	I
M	✗	✗	✗	✓
E	✗	✗	✗	✓
S	✗	✗	✓	✓
I	✓	✓	✓	✓

The function would print an error if the MESI cache system is violated.

The function is call at the end of perform_access_load and perform_access_store during debugging.

**All my simulation passed this test, which mean my simulation does not violate this MESI coherence rule.

perform_access_load:

The function would first increment the access count and then call the get_node() function to see if any cache line holds the specified tag. If get_node() returns a NULL, the cache line does not exist in the cache, otherwise get_node()returns a pointer to the cache line.

when the cache block does not exist:

In this case, a **read miss** occurs, so a misses and fetches increments.

1. The function would check whether the cache is full. If it is full, it will remove the tail of the cache line. Furthermore, if the removed cache line is in the modified state, copies back will be incremented. The cache_contents count will be decremented.
2. The function will call the no_other_copies() to check whether other caches hold the block. This will determine the if the inserted block would be in state “shared” or state “exclusive”.
3. The function will insert the new block into the linked list with the updated information. The cache_content count is also incremented.
4. The function will call the function bus_transaction() with transaction type bus_read_miss. The broadcast count is also incremented.

when the cache block exists in the cache and is in state “Invalid”:

In this case, a **read miss** occurs, so a misses and fetches increments.

1. The function does not need to check whether the cache is full. Since we only need to replace the old cache line.
2. The function will call no_other_copies() to check whether other caches hold the block. This will determine the if the inserted block would be in state “shared” or state “exclusive”.
3. The function will delete the old block and insert the new block into the linked list with the updated information.
4. The function will call bus_transaction() with transaction type bus_read_miss. The broadcast count is also incremented.

when the cache block exists in the cache and is in state “Shared”, “Exclusive” or “Modified”:

In this case, a **read hit** occurs.

1. Then the function would delete the old block and insert the new block into the linked list with the old information. We are doing this to update the LRU structure.
2. Then the function would NOT call the function bus_transaction function since it is a read hit and no bus communication is needed for this state.

perform_access_store:

The function would first increment the access count and then call the get_node() function to see if any cache line holds the specified tag. If get_node() returns a NULL, the cache line does not exist in the cache, otherwise get_node() returns a pointer to the cache line.

A new cache line with the specified tag and with state "modified" is created and prepared to be inserted later.

when the cache block does not exist:

In this case, a **write miss** occurs, so a misses and fetches increments.

1. The function would check whether the cache is full. If it is full, it will remove the tail of the cache line. Furthermore, if the removed cache line is in the modified state, copies back will be incremented. The cache_contents count will be decremented.
2. The function will insert the new block into the linked list with the updated information (tag and state modified). The cache_content count is also incremented.
3. The function will call the function bus_transaction() with transaction type bus_write_miss. The broadcast count is also incremented.

when the cache block exists in the cache and is in state "Invalid":

In this case, a **write miss** occurs, so a misses and fetches increments.

1. The function will delete the old block and insert the new block into the linked list with the updated information.
2. The function will call bus_transaction() with transaction type bus_write_miss. The broadcast count is also incremented.

when the cache block exists in the cache and is in state "Shared":

In this case, a **write hit** occurs.

1. The function will delete the old block and insert the new block into the linked list with the updated information.
2. The function will call bus_transaction() with transaction type bus_write_hit. The broadcast count is also incremented.

when the cache block exists in the cache and is in state "Shared", "Exclusive" or "Modified":

In this case, a **write hit** occurs.

1. Then the function would delete the old block and insert the new block into the linked list with the old information. We are doing this to update the LRU structure.
2. Then the function would NOT call the function bus_transaction function since it is a write hit and no bus communication is needed for this state.

Result and Benchmark

Green blocks	represent a <1% error
Yellow blocks	represent a <10% error
Red blocks	represents a >10% error

spice-1c_8192_16

validation	
access	1000001
misses	25966
demand fetch	103864
broadcasts	25966
copies back	8472

my results	
access	1000001
misses	25966
demand fetch	103864
broadcasts	25966
copies back	8472

Percent Error	
access	0
misses	0
demand fetch	0
broadcasts	0
copies back	0

The benchmark for 1 core is very promising, with all the results matching the validation exactly. This reveals that my program can simulate a single core cache very accurately.

fft-2c_8192_16

validation		
access	3465000	3464832
misses	481255	294136
demand fetch	3101564	
broadcasts	1089493	
copies back	2468364	

my results		
access	3465000	3464832
misses	498663	311273
demand fetch	3239744	
broadcasts	1124036	
copies back	2469380	

Percent Error		
access	0	0
misses	3.617209172	5.826216
demand fetch	4.455171649	
broadcasts	3.170557314	
copies back	0.041160866	

The benchmark for 2 cores is relatively accurate, with the results having a maximum error of 6 percent. Statistics with a higher error comes from slightly overcounting misses, demand fetches and broadcasts. The copies back statistics is also very close to the validation.

fft-4c_8192_16

validation				
access	1734444	1734230	1734209	1734225
misses	294809	316358	273634	271411
demand fetch	4624848			
broadcasts	1736479			
copies back	3390312			

my results				
access	1734444	1734230	1734209	1734225
misses	309761	332927	290425	288028
demand fetch	4884564			
broadcasts	1801211			
copies back	3396044			

Percent Error				
access	0	0	0	0
misses	5.071758	5.237421	6.136299	6.122449
demand fetch	5.615666			
broadcasts	3.727773			
copies back	0.16907			

The benchmark for 4 cores is relatively accurate, with the results having a maximum error of 6 percent. Like the 2 cores simulation, the statistic with a higher error comes from slightly overcounting misses, demand fetches and broadcasts. The copies back statistics is also very close to the validation.

fft-8c_8192_16

validation								
access	869616	869343	869385	869504	869336	869326	869335	869300
misses	285047	287972	298687	278884	270172	264075	308474	299313
demand fetch	9170496							
broadcasts	3536652							
copies back	7180320							

my results								
access	869616	869343	869385	869504	869336	869326	869335	869300
misses	324157	322744	314027	334488	320444	319702	335742	312350
demand fetch	10334616							
broadcasts	3782687							
copies back	8337932							

Percent Error								
access	0	0	0	0	0	0	0	0
misses	13.72054433	12.07478505	5.135811066	19.93804	18.60741	21.06485	8.839643	4.355641
demand fetch	12.69418797							
broadcasts	6.95672065							
copies back	16.12201128							

The benchmark for the 8 cores is the least promising, with statistics having an error as much as 20%

fft-2c_16384_16

validation		
access	3465000	3464832
misses	260142	260671
demand fetch	2083252	
broadcasts	834958	
copies back	1715944	

my results		
access	3465000	3464832
misses	260940	260676
demand fetch	2086464	
broadcasts	835752	
copies back	1719060	

Percent Error		
access	0	0
misses	0.306755541	0.001918127
demand fetch	0.154182019	
broadcasts	0.095094604	
copies back	0.181591008	

The benchmark for 2 cores with a larger cache size shows very accurate results. The largest percent error is a .31 percent. With a larger cache size, there will be less replacements since the cache can store more blocks. This conclusion could explain why simulations above have a less accurate result. One reason could be that my replacement policy is not as well simulated as it could be, causing overcounting.

fft-4c_16384_16

validation				
access	1734444	1734230	1734209	1734225
misses	271982	269773	255096	254218
demand fetch	4204276			
broadcasts	1655796			
copies back	3110452			

my results				
access	1734444	1734230	1734209	1734225
misses	272135	269803	256103	254355
demand fetch	4209584			
broadcasts	1656269			
copies back	3117996			

Percent Error				
access	0	0	0	0
misses	0.056254	0.01112	0.394753	0.053891
demand fetch	0.126252			
broadcasts	0.028566			
copies back	0.242537			

The benchmark for 4 cores with a larger cache size also shows very accurate results. Like the above simulation, the largest percent error is a .39 percent.

fft-2c_16384_32

validation		
access	3465000	3464832
misses	241220	175773
demand fetch	3335944	
broadcasts	618627	
copies back	2569064	

my results		
access	3465000	3464832
misses	241620	175773
demand fetch	3339144	
broadcasts	619021	
copies back	2572128	

Percent Error		
access	0	0
misses	0.165823729	0
demand fetch	0.095924872	
broadcasts	0.063689428	
copies back	0.119265227	

The benchmark for 2 cores with a larger cache size and a larger block size also shows very accurate results, with the largest percent error of .17 percent.

fft-4c_16384_32

validation				
access	1734444	1734230	1734209	1734225
misses	189122	170033	183025	170742
demand fetch	5703376			
broadcasts	1068118			
copies back	3639248			

my results				
access	1734444	1734230	1734209	1734225
misses	189228	170051	183530	170834
demand fetch	5709144			
broadcasts	1068340			
copies back	3646952			

Percent Error				
access	0	0	0	0
misses	0.056048	0.010586	0.275919	0.053882
demand fetch	0.101133			
broadcasts	0.020784			
copies back	0.211692			

The benchmark for 4 cores with a larger cache size and a larger block size also shows very accurate results, with the largest percent error of .28 percent.

Observed Issues

Higher percent error with smaller cache size.

As we can see from the results, the caches with a smaller cache size have a less accurate result. I believe that this is due to the replacement policy that I currently implemented is not extensive enough.

Higher percent error with 8 cores.

As we can see from the results, the simulation with 8 cores has the least promising result. I think this is a combination of problems, such as the previously mentioned replacement policy and the interaction between cores that could be compounding the error.

How I could improve the simulation

To improve the simulation, I will have to fix the problems mentioned above, especially the replacement policy.

Currently, my replacement policy is to remove the tail of cache and increment copies back if the removed cache line is modified. My implementation assumes that the removed block would be gone and the next read or write to that tag would be treated like a read miss or write miss to a non-existing block, which would then increment read miss. This implementation basically skipped a step from the MESI cache graph that is introduced in the textbook. So, if there is any intricate detail about the replacement policy that I didn't know that would save the system from having a read/write miss, that would be the reason my simulation slightly overcounts misses hence the demand fetch would also affect since demand fetches is basically the total of misses times words-per-block.

The error in number of broadcast could have similar reasoning. If there are more block that are treated as invalid or not in cache, there will subsequently be more bus transaction since most bus transactions originates from an invalid block or block not in cache.