Multicore Processors: Architecture and Programming

Project Report
Comparing Cache Coherence Protocols

Author - Adit Kotwal
Net Id - ask9100

Abstract

In a shared-memory architecture, it is important to keep each copy of data up to date in different processors that access it. There should not be any discrepancies between cores that access the same data when it is brought in from the main memory into a local cache of a requesting processor. This is where cache coherence comes in and the protocols that are used to ensure data is uniform and up-to-date throughout the system.
By making use of a generator and simulator program, the project authenticates the advantages of the MESI and MOSI protocol over MSI and how the MOESI protocol encompasses the advantages of both (MESI and MOSI).

Introduction

Without cache coherence, each core would be reading an outdated version of a data value that may be modified by the local cache of another processor, thereby defeating the purpose of having multiple caches to speed up programs in a multicore environment. Thus came the need for developing coherence protocols that ensure consistent access while minimizing the latencies involved.
This project aims to explore different cache coherence protocols in terms of their bandwidth requirements and latency. Four protocols have been analyzed and implemented, namely:

1. MSI
2. MESI
3. MOSI
4. MOESI

The analysis section surveys the advantages of adding additional states in a protocol and also answers the question of whether they are really necessary. This can give us an insight into what protocol is best suited for a given application based on a set of instructions. The MESI and MOSI protocols provide undeniable advantages over the MSI protocol in most cases but upon further observation, it can be concluded that MSI can outperform both these protocols depending on the instruction set.

To implement this project, a generator program is used to produce a set of instructions that will be passed to the simulator program that will execute these instructions based on the number of cores and the protocol that is specified.

Literature Review

In [1], a basic understanding of consistency and coherence has been provided along with implementation details from real-world applications. It also discusses how caching can reduce memory access latency by storing the data that is most frequently accessed in a temporal or spatial manner. In [2], a trenchant analysis has been made for the protocols: MSI, MESI, MOSI, MOESI, token-based, and hammer-based protocols. Even though this paper talks about the advantages of the directory-based protocol over snooping protocol, it does not explore in-depth how the number of directory transfers can vary depending on the protocol. The paper makes use of a simulator to demonstrate the working whereas this project makes use of simulator program capable of generating a trace for a given protocol and the set of instructions.

[4] describes the use of an interconnected network to transfer data from the directory to each processor based on the incoming request. This paper talks about issues such as directory or cache deadlock but does not provide implementation details to simulate its working. Similarly, [5] describes the architecture and structure required to build a directory-based multiprocessor system.

The mentioned references do not provide a reproducible ability to generate a trace and verify the working of multiple cache coherence protocols. This project aims to do just that and provide the feature of understanding the state transitions at each step. A summary is also provided at the end to observe various parameters based on the protocol in action for a given set of instructions.

Proposed Idea

For this project, a set of instructions have been generated based on the number of cores in a system. The generator program is capable of producing instructions in the specified format:
Time : Thread: Memory address: Access Type: #Bytes
The thread number defines the core that is requesting access starting from the given memory location (32-bit addressing). The access type defines whether it is a read or a write. # Bytes parameter defines the number of bytes to be read from or written from the memory location. To assist the simulator program, 4 files have been generated based on the number of cores (2, 4, 8, 16).

Once these instructions have been generated, they are used in the simulator program. This program simulates the working of the 4 protocols in a directory-based manner. Before simulation begins, the program accepts the user to specify the following parameters: the number of processors, total cache size (32KB or 64KB), the type of mapping (Direct or Set-Associative with associativity value), and the write method (write-back and write-through). The cache block size is fixed at 64 bytes and makes use of write invalidate.
Based on these values, the tag bits, the cache set bits, and offset bits are calculated.

For each protocol, a summary is printed stating the number of read/write misses, read/write hits, invalidations, writebacks, and directory transfers. These values are analyzed to determine how each protocol scales with an increase in the number of cores.

Experimental Setup

The project has been entirely written and executed on Python 3.9.5 and can be run on any machine running Python 3.6 and above. Approximately, 100000 instructions have been generated using a python script to aid the simulation. Instructions have been added to display the superiority of a protocol over another. These instructions including the time, thread, and access parameters are saved in a CSV file that is later read by the simulator program.
The simulator program can also run on any machine running Python 3.6 and above. Based on the parameters specified by a user, this program simulates the four protocols in action. Upon completion, different variables that are used to understand the functioning of each protocol are displayed based on which an analysis section is created that talks about the results in further detail.

Results and Analysis

Understanding the advantage of set-associative mapping over direct mapping

For the following tables (1-4), a cache size of 32 KB and write back mechanism is used. These tables indicate the number of read/write hits (RH or WH) based on the number of cores and the type of caching used for a given protocol.

Cache Type	2 cores	4 cores	8 cores	16 cores
Direct	RH - 643 WH - 596	RH - 545 WH - 585	RH - 585 WH - 557	RH - 461 WH - 487
2 way	RH - 915 WH - 942	RH - 932 WH - 982	RH - 933 WH - 902	RH - 774 WH - 815
4 way	RH - 946 WH - 928	RH - 974 WH - 978	RH - 932 WH - 896	RH - 763 WH - 843
8 way	RH - 933 WH - 934	RH - 998 WH - 968	RH - 940 WH - 915	RH - 759 WH - 855

Table 1. Read/Write Hits for MSI Protocol

Cache Type	2 cores	4 cores	8 cores	16 cores
Direct	RH - 610 WH - 568	RH - 528 WH - 587	RH - 555 WH - 562	RH - 511 WH - 495
2 way	RH - 921 WH - 951	RH - 947 WH - 992	RH - 962 WH - 936	RH - 827 WH - 875
4 way	RH - 950 WH - 933	RH - 984 WH - 990	RH - 954 WH - 926	RH - 810 WH - 890
8 way	RH - 938 WH - 940	RH - 1009 WH - 980	RH - 968 WH - 935	RH - 809 WH - 906

Table 2. Read/Write Hits for MESI Protocol

Cache Type	2 cores	4 cores	8 cores	16 cores
Direct	RH - 594 WH - 551	RH - 468 WH - 502	RH - 471 WH - 458	RH - 355 WH - 386
2 way	RH - 779 WH - 844	RH - 812 WH - 844	RH - 839 WH - 831	RH - 770 WH - 839
4 way	RH - 855 WH - 886	RH - 932 WH - 910	RH - 861 WH - 837	RH - 748 WH - 845
8 way	RH - 935 WH - 930	RH - 1004 WH - 964	RH - 946 WH - 908	RH - 767 WH - 866

Table 3. Read/Write Hits for MOSI Protocol

Cache Type	2 cores	4 cores	8 cores	16 cores
Direct	RH - 641 WH - 593	RH - 541 WH - 587	RH - 586 WH - 556	RH - 464 WH - 492
2 way	RH - 779 WH - 845	RH - 812 WH - 847	RH - 844 WH - 835	RH - 785 WH - 858
4 way	RH - 859 WH - 888	RH - 933 WH - 915	RH - 876 WH - 847	RH - 772 WH - 868
8 way	RH - 941 WH - 936	RH - 1013 WH - 975	RH - 967 WH - 935	RH - 813 WH - 919

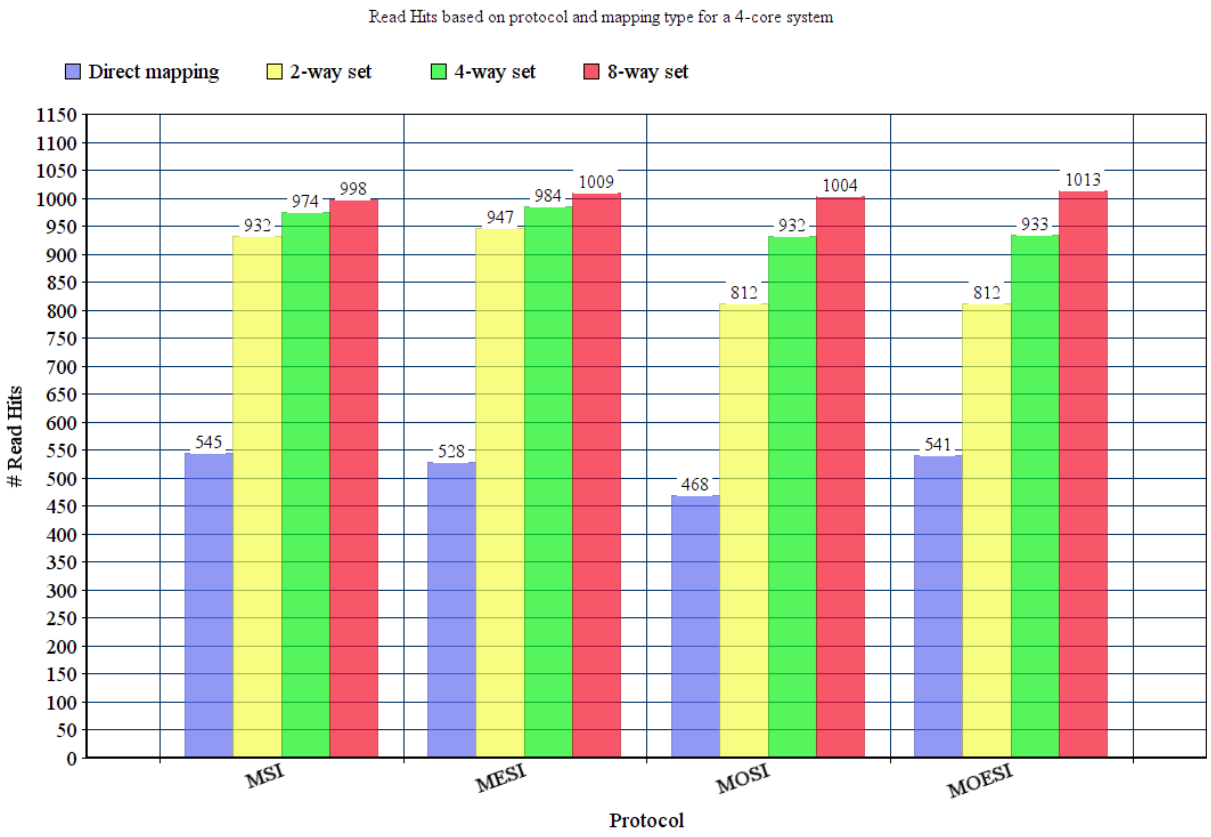
Table 4. Read/Write Hits for MOESI Protocol

Based on these tables, we can observe the following:

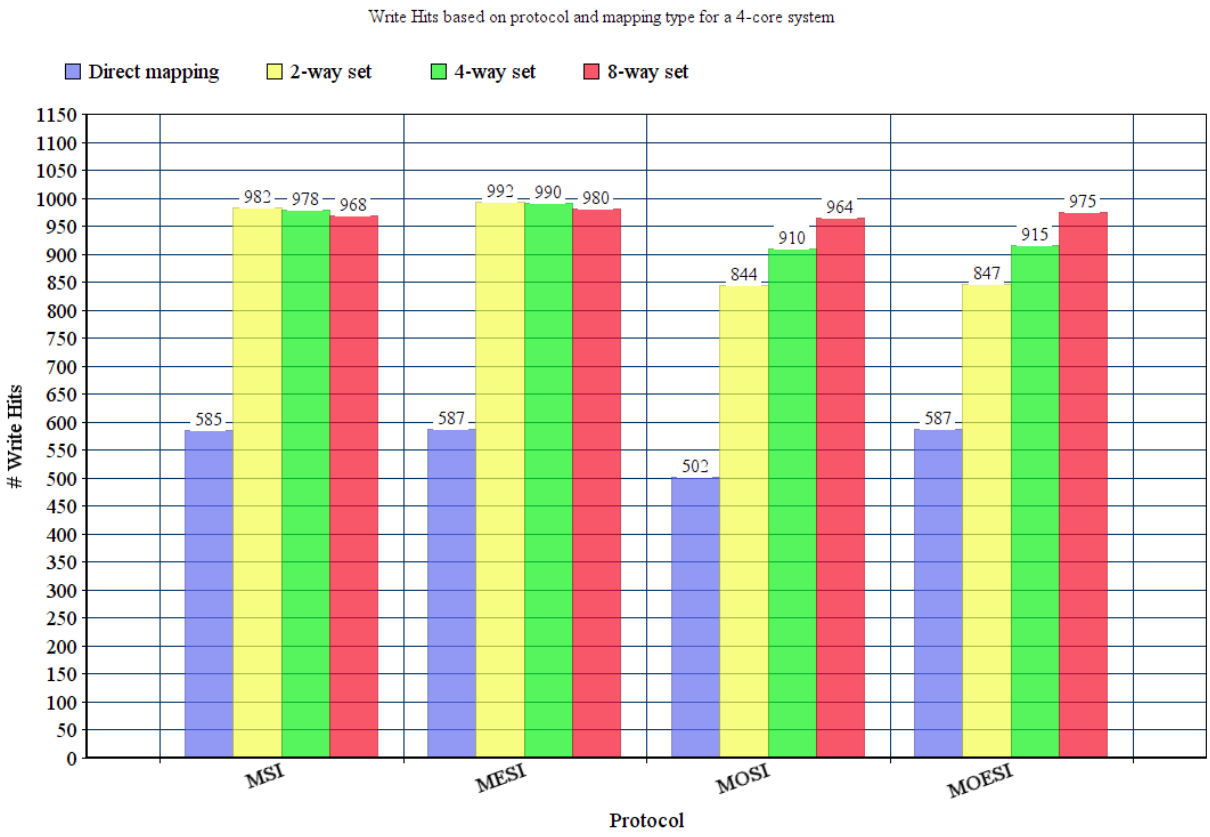
1. There is a rise in the number of read/write hits when switching from direct mapping to set-associative mapping. This is because of the following reasons:
    a. Since there is just one cache block in direct mapping that a memory block maps to, the rate of its eviction will increase as there will be other blocks that need to be placed on that cache line.
    b. With set-associative cache, there are multiple cache lines within a set that a block of memory can be placed into. Hence these blocks do not need to be constantly evicted thereby increasing the number of hits.
2. There is a rise in the number of read/write hits with an increase in associativity as well. This is because of the following reasons:

a. An increase in associativity translates to an increase in the number of cache lines per set. Thus, a block of memory has more choices to be placed in the set.
b. Thus the rate of eviction will be reduced further thereby increasing the number of hits.
c. Graph (1) and (2) displays how the number of read-hits increases with a switch from direct to associative along with an increase in associativity for a 4-core system.

3. With an increase in the number of cores, the difference between the number of read/write hits between direct and associative caching is also greater.



Graph 1. Read hits based on mapping and protocol for a 4-core system



Graph 2. Write hits based on mapping and protocol for a 4-core system

Understanding the advantage of MESI protocol over MSI to reduce invalidations

The extra state in the MESI protocol has been added to reduce the number of invalidations and directory transfers as compared to the MSI protocol. When a block is in an exclusive state, additional directory transfers are unnecessary as there are no other copies of that block.
Similarly, the MOSI protocol serves to have an edge over the MSI protocol by reducing the number of writebacks to memory unless it is utterly necessary. The existence of the owned state delays writebacks to main memory.

The following tables (5-8) describe the number of invalidations (INV), writebacks (WB), and directory transfers (DT) for a given pair of mapping type and the number of cores in the system.

Cache Type	2 cores	4 cores	8 cores	16 cores
Direct	INV - 66736 WB - 40056 DT - 131600	INV - 79357 WB - 36216 DT - 158310	INV - 87979 WB - 34804 DT - 174885	INV - 93073 WB - 34190 DT - 186152
2 way	INV - 1342 WB - 900 DT - 2215	INV - 3398 WB - 2938 DT - 6221	INV - 6585 WB - 6085 DT - 12677	INV - 11362 WB - 10878 DT - 22226
4 way	INV - 1360 WB - 912 DT - 2224	INV - 3420 WB - 2884 DT - 6211	INV - 6612 WB - 6107 DT - 12688	INV - 11389 WB - 10957 DT - 22428
8 way	INV - 1374 WB - 907 DT - 2226	INV - 3434 WB - 2927 DT - 6237	INV - 6620 WB - 6120 DT - 12699	INV - 11650 WB - 11042 DT - 22706

Table 5. Invalidations, writebacks, and directory transfers for MSI Protocol

Cache Type	2 cores	4 cores	8 cores	16 cores
Direct	INV - 67754 WB - 40464 DT - 133878	INV - 80764 WB - 36617 DT - 161455	INV - 89629 WB - 35116 DT - 178825	INV - 94889 WB - 34429 DT - 189999
2 way	INV - 886 WB - 892 DT - 1758	INV - 2964 WB - 2949 DT - 5820	INV - 6423 WB - 6070 DT - 12686	INV - 11615 WB - 10825 DT - 23031
4 way	INV - 918 WB - 908 DT - 1782	INV - 2955 WB - 2889 DT - 5781	INV - 6454 WB - 6067 DT - 12654	INV - 11625 WB - 10961 DT - 23105
8 way	INV - 918 WB - 903 DT - 1770	INV - 2989 WB - 2922 DT - 5805	INV - 6430 WB - 6095 DT - 12631	INV - 11785 WB - 11020 DT - 23200

Table 6. Invalidations, writebacks, and directory transfers for MESI Protocol

Cache Type	2 cores	4 cores	8 cores	16 cores
Direct	INV - 67760 WB - 20452 DT - 133753	INV - 80803 WB - 18376 DT - 161392	INV - 89700 WB - 17776 DT - 178765	INV - 95039 WB - 17133 DT - 190387
2 way	INV - 1260 WB - 412 DT - 2040	INV - 2983 WB - 1233 DT - 5368	INV - 6176 WB - 2755 DT - 11733	INV - 10739 WB - 4701 DT - 21003
4 way	INV - 1342 WB - 462 DT - 2108	INV - 3243 WB - 1362 DT - 5850	INV - 6303 WB - 2712 DT - 12181	INV - 11144 WB - 4933 DT - 21831
8 way	INV - 1394 WB - 460 DT - 2250	INV - 3475 WB - 1469 DT - 6308	INV - 6830 WB - 2980 DT - 13049	INV - 12172 WB - 5171 DT - 23540

Table 7. Invalidations, writebacks, and directory transfers for MOSI Protocol

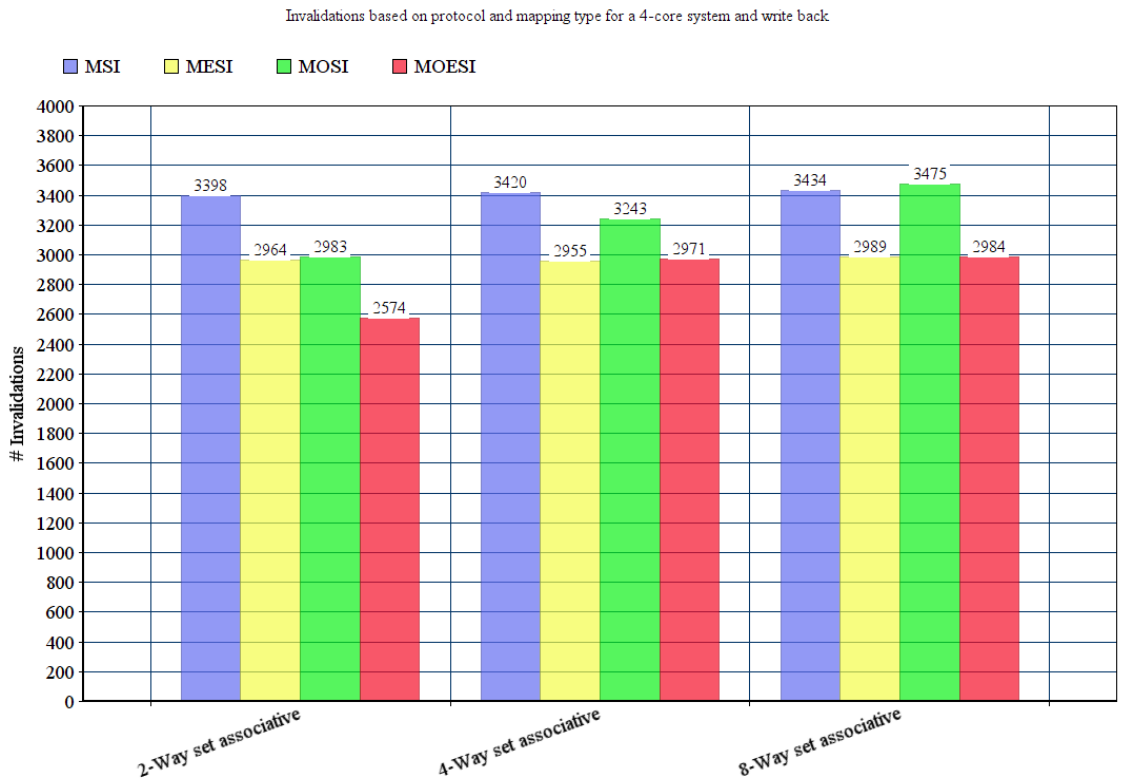
Cache Type	2 cores	4 cores	8 cores	16 cores
Direct	INV - 67758 WB - 25489 DT - 133684	INV - 80795 WB - 25264 DT - 16162	INV - 89671 WB - 25325 DT - 178512	INV - 95004 WB - 25482 DT - 190049
2 way	INV - 826 WB - 413 DT - 1609	INV - 2574 WB - 1257 DT - 4965	INV - 5818 WB - 2862 DT - 11412	INV - 10458 WB - 5033 DT - 20851
4 way	INV - 913 WB - 460 DT - 1680	INV - 2791 WB - 1372 DT - 5403	INV - 5951 WB - 2809 DT - 11884	INV - 10904 WB - 5212 DT - 21767
8 way	INV - 925 WB - 457 DT - 1781	INV - 2984 WB - 1474 DT - 5818	INV - 6441 WB - 3061 DT - 12664	INV - 11744 WB - 5474 DT - 23112

Table 8. Invalidations, writebacks, and directory transfers for MOESI Protocol

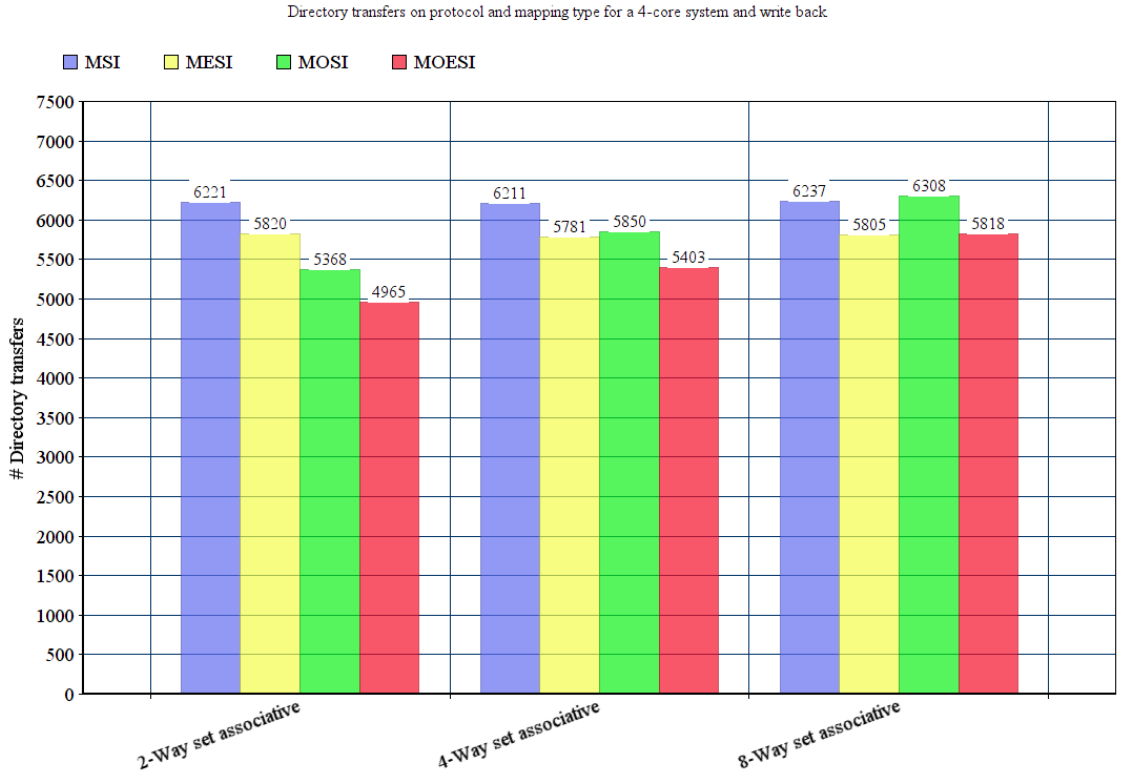
Based on these tables, we can observe the following:

1. The number of invalidations and directory transfers are greatly reduced with a switch from direct to set-associative mapping. This is because:

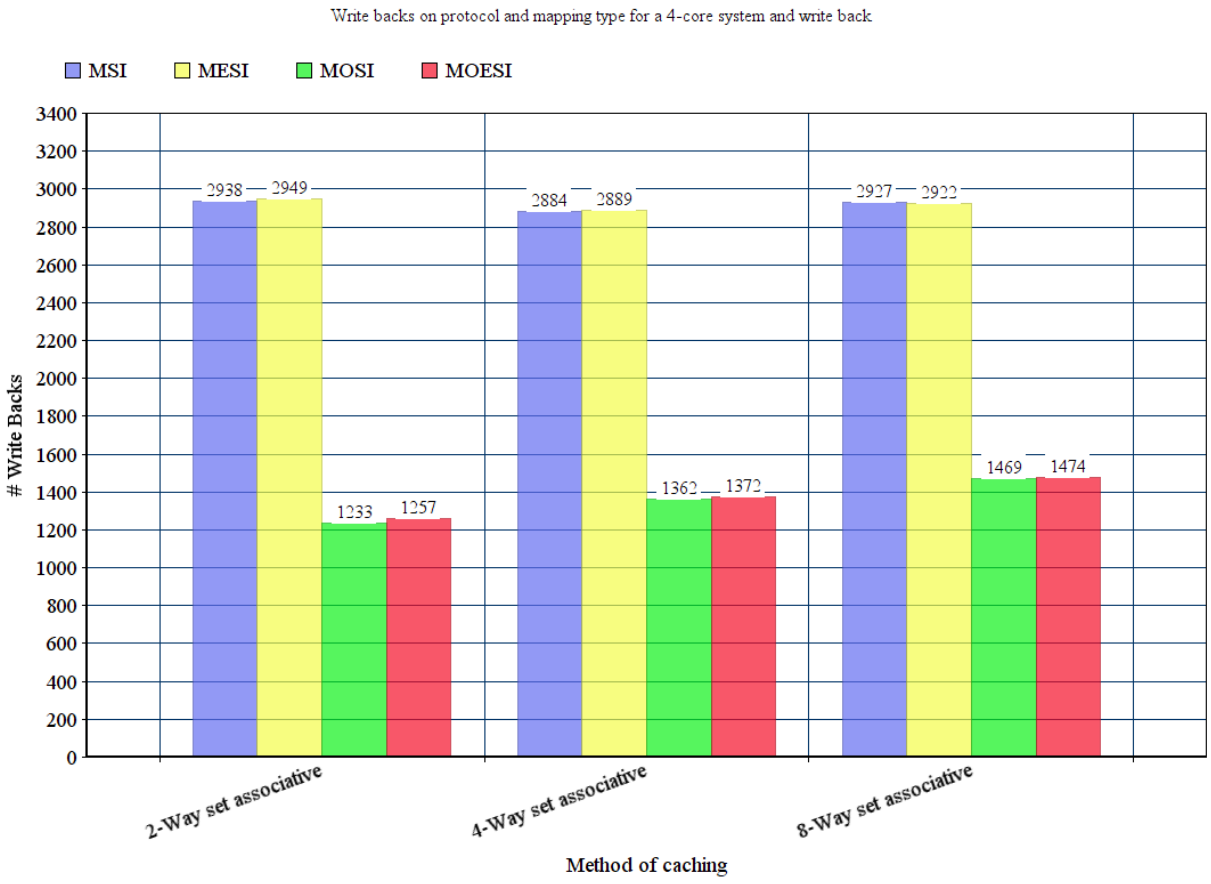
    a. Since there is just one cache block in direct-mapping that a memory block maps to, the rate of its eviction will increase as there will be other blocks that need to be placed on that cache line.
    b. The new cache blocks that are being brought in will constantly have to be invalidated with directory transfers.
2. The number of invalidations along with directory transfers are also reduced significantly when switching from the MSI to the MESI protocol.
    a. This is because when a block is in an exclusive state, there will be no additional directory transfers as there are no other copies of that block.
    b. However, the number of invalidations in MOSI is less than MSI but not lesser than MESI and MOESI.
    c. Consider a 4 core system with 2 way set associative mapping. The number of invalidations and directory transfers is significantly lesser in MESI and MOESI as compared to MSI protocol. Graph (3) and (4) displays how the invalidations and directory transfers are reduced from MSI to MESI and MOESI.
3. The number of writebacks is significantly reduced when switching from MSI to MOSI protocol.
    a. This is because the presence of the owned state delays unnecessary writebacks to main memory in case of a write operation.
    b. Consider a 4 core system with 4 way set associative mapping. The number of writebacks here is significantly lesser in MOSI and MOESI as compared to MSI protocol. Graph (5) displays how the writebacks are significantly lower in MOSI and MOESI in k-way associative mapping.



Graph 3. Invalidations based on the protocol and increasing set associativity



Graph 4. Directory transfers based on the protocol and increasing set associativity



Graph 5. Write-backs based on the protocol and increasing set associativity

Understanding the effect of write-back method and write-through method on the number of writebacks

The primary difference between write-back and write-through is that in the former, the lower level (in this case memory) is only written back to when the block is being evicted from the cache block. Whereas in case of write-through, the lower level is constantly written back to when the data in that block is updated. Thus the bandwidth requirement for implementing a write-through cache is much higher than implementing a write-back cache.

Consider the following tables (9-12) indicating the number of writebacks for a given pair of number of cores and the type of mapping based on the specified protocol. These values are based on the write-through method.

Cache Type	2 cores	4 cores	8 cores	16 cores
Direct	WB - 40652	WB - 36801	WB - 35361	WB - 34677
2 way	WB - 1842	WB - 3920	WB - 6987	WB - 11693
4 way	WB - 1840	WB - 3862	WB - 7003	WB - 11800
8 way	WB - 1841	WB - 3895	WB - 7035	WB - 11897

Table 9. Writebacks for MSI protocol based on write-through method

Cache Type	2 cores	4 cores	8 cores	16 cores
Direct	WB - 41032	WB - 37204	WB - 35678	WB - 34924
2 way	WB - 1843	WB - 3941	WB - 7006	WB - 11700
4 way	WB - 1841	WB - 3879	WB - 6993	WB - 11851
8 way	WB - 1843	WB - 3902	WB - 7030	WB - 11926

Table 10. Writebacks for MESI protocol based on write-through method

Cache Type	2 cores	4 cores	8 cores	16 cores
Direct	WB - 21003	WB - 18878	WB - 18234	WB - 17519
2 way	WB - 1256	WB - 2077	WB - 3586	WB - 5540
4 way	WB - 1348	WB - 2272	WB - 3549	WB - 5778
8 way	WB - 1390	WB - 2433	WB - 3888	WB - 6037

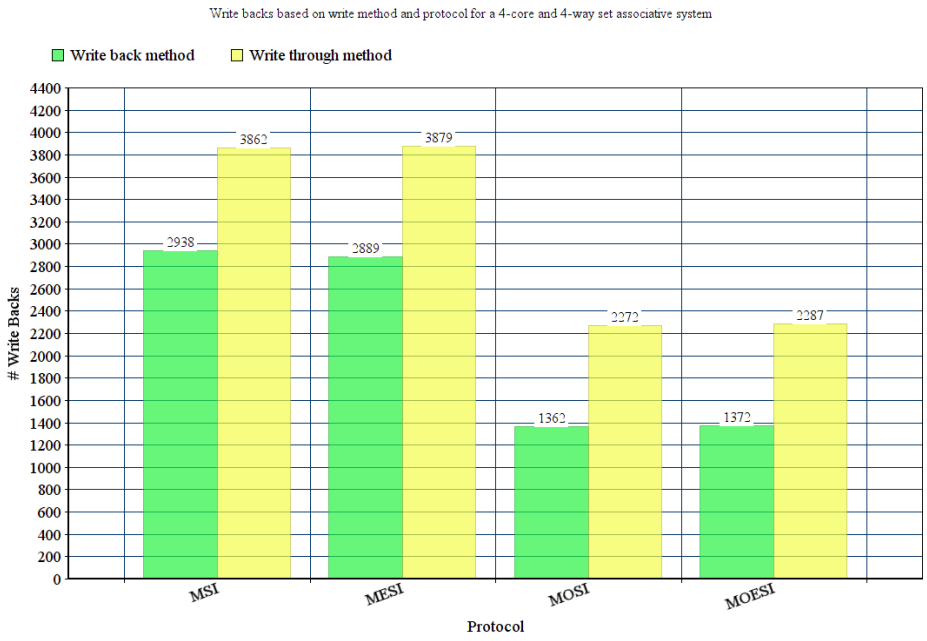
Table 11. Writebacks for MOSI protocol based on write-through method

Cache Type	2 cores	4 cores	8 cores	16 cores
Direct	WB - 26081	WB - 25851	WB - 25881	WB - 25970
2 way	WB - 1258	WB - 2104	WB - 3697	WB - 5891
4 way	WB - 1348	WB - 2287	WB - 3656	WB - 6080
8 way	WB - 1393	WB - 2449	WB - 3996	WB - 6393

Table 12. Writebacks for MOESI protocol based on write-through method

Based on these tables, we can observe the following:

1. When compared to the number of writebacks to memory with the write-back method (Tables 5-8), the number of writebacks to memory with write-through are much higher.
    a. This is because with write through the memory is accessed every time there is an update to the block that is held by a cache. As opposed to writeback method when there is a memory access only when a block is being removed from the cache block.
    b. Graph (6) displays how the writebacks are more in write-through for a given 4-core and 4-way set associative system regardless of the protocol being used.



Graph 6. Comparison of write-back and write-through for each protocol

Conclusions

Based on the observations made in the analysis section, we can conclude the following:

1. The bandwidth requirement for set-associative caching is less as compared to the direct-mapping technique. Due to a higher number of read/write hits in set-associative caching, the cache latency is also less in this case compared to direct caching due to lesser accesses to main memory.
   With an increase in the number of cores, the number of read and write hits decreases for each type of protocol. Thus with an increase in the number of cores, the bandwidth requirement is slightly reduced depending on the instruction set.
2. The bandwidth requirement for MESI and MOESI protocol is lesser as compared to MSI since there is a lesser need to send a message from the directory to other cores to either invalidate them or send data. The latency in MESI and MOESI is also lesser due to fewer invalidation messages that need to be sent by the directory to the other cores.
   The number of invalidations and directory transfers increases with an increase in the number of cores and an increase in the associativity factor for each protocol. Thus for a higher number of cores and higher associativity, the bandwidth requirement will be higher for a given protocol.
3. The bandwidth requirement for MOSI and MOESI protocol is also lesser compared to MSI since there is a lesser need to access the memory as the owned state delays writebacks. Hence the latency in MOSI and MOESI will also be lesser due to fewer accesses to the main memory.
   However, for a given protocol, the number of writebacks increases with an increase in the number of cores and an increase in the associativity factor. Thus the bandwidth requirement will be higher for a larger number of cores and associativity for any given protocol.
4. By making use of the writeback method instead of write-through method, bandwidth requirements can be further reduced as there are fewer accesses to main memory in the former method thereby also reducing the latency. We also observe that the MOESI protocol makes use of the features of both MESI and MOSI protocol thereby making it a good choice where there are several reads and writes to particular memory locations.

References

1. Vijay Nagarajan, Daniel J. Sorin, Mark D. Hill, David A Wood, A primer on Memory Consistency and Cache Coherence, Second Edition, 2020
2. Anop Tiwari, Performance Comparison of Cache Coherence Protocol on Multi-Core Architecture, 2014
3. Adve. et all, comparison of hardware and software cache coherence schemes, 1991.
4. Richard Simoni, Implementing Directory-based Cache coherence protocols, 1990
5. Lenoski et all, Directory based cache coherence protocol for DASH multiprocessor.
6. Roy et all, Comparative study on Cache Coherence Protocols, 2015