Cache Coherency Protocols

Victoria Peterson - 15476758

***Repository**: https://github.com/VictorianHues/CacheCoherencyProtocolModels*

I. Introduction

DATA access on computer systems relies heavily on the ability to access information locations both efficiently and accurately as the delay in accessing memory locations within DRAM can be orders of magnitude greater than that of accessing SRAM. This SRAM Cache access is the fundamental basis for the access of data very quickly, relying on locality to the processor sending the request, frequency of data use, and recency. When multiple processors and their connected caches become involved, the system must be able to arbitrate data requests so that each cache may be able to remain informed of the most recent data on the other caches without continuous communication between each.

The solution to this synchronization issue comes mainly in the form of *Cache Coherency Protocols* which are used to ensure that multiple copies of the same memory location in different processor caches remain consistent. These protocols rely on a tracked Cache State that determines what actions the cache will take when certain requests occur, possibly changing the cache state in a process that may be described using a *State Transition Diagram* such as that seen in Figure 1.

The simple state transition diagram may be used as a baseline for comparison with more complex Cache Coherency Protocols such as the MOESI in Figure 2 that includes three additional states to manage synchronization. The comparison of the two allows for an observation of Read and Write Hit Rates, Bus arbitration wait times, request processing time, and Main Memory access counts, all of which give a better understanding of why one may be chosen over another on real-systems.

II. Theory

Caches are comprised of *Cache Lines* typically of the size 32-256 bytes and composed of a *Tag* identifying which block of memory the line holds, a *State* used by the cache coherency protocol, and the *Data* itself. When a CPU sends a read or write to the Cache, it communicates a *Memory Address* that points to a specific byte in memory determined by the *Cache Mapping* architecture.

The structure of the memory address differs depending on the cache map, but is generally composed of a *Tag* which identifies which block in memory the request is being made on, the *Index* which determines the cache line being mapped to, and the *Offset/Byte-in-line* which determines the specific byte being accessed within the block. The cache map itself may be *Fully Associative*, mapping any memory block to any cache, *Direct Mapped*, with a unique cache line for any block in memory, or *Set-associative*, where each cache contains cache lines grouped into sets based on *set associativity*.

Cache requests from the CPU may be Reads or Writes to and from memory that result in *Cache Hits* if the memory address maps successfully to a specific cache line, or a *Cache Miss* if it does not. Cache misses may incur large time costs due to the delay when accessing Main Memory, and thus a number of design specifications must be considered when implementing cache protocols.

One such specification is the write-depth when writing to a cache line determining where in memory data is written to between the different cache levels and the main memory. A *Write-through* policy writes data to the cache and sends it down to lower levels, while a *write-around* policy sends data down to lower levels but doesn't write to the cache, and *copy-back* which writes to the cache, but doesn't send it down the hierarchy.

The decision between different cache actions depends largely on the Cache Coherency Protocol which is used to ensure that multiple copies of the same memory location in different processor caches and main memory remain consistent across the system. These protocols are required as updates to memory address in one processor may lead to inconsistencies in others such that coherence must be maintained on different copies of the same block of memory.

The simple VALID-INVALID protocol in Figure 1 is a *Snoopy* protocol which uses the shared Bus access between caches to "see" every memory transaction in the system and thus maintain the information on shared cache lines. This process is visualized in a state transition diagram which describes what happens to a cache line state when reads and writes occur from the local CPU or from "probe" reads and writes snooped from the Bus. The simple VALID-INVALID state transition diagram describes a system by which write hits on a local cache

will be observed by other caches and cause their state to change from Valid to Invalid while local read and write misses will cause state transitions from Invalid to Valid.

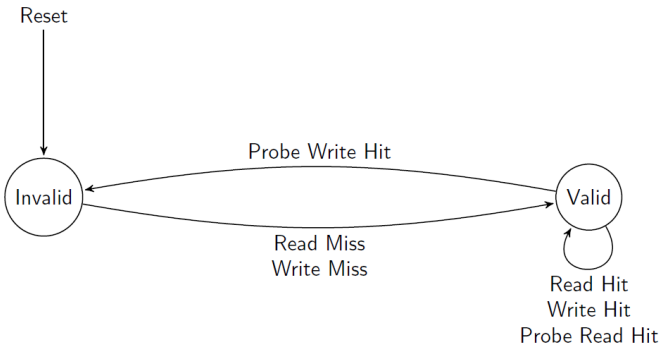


Fig. 1: State Transition Diagram for a Simple VALID-INVALID Cache Coherency Protocol

The *MOESI* cache coherency protocol in Figure 2 takes the simple VALID-INVALID model and expands it to be more aware of the state of the data stored in Main Memory. To achieve this, each cache line tracks a state with five possible options:

- Modified (M): The cache line is exclusive to the cache, has been modified, and is different from main memory. The owner must write it back to main memory before another processor reads it.
- Owned (O): The cache line holds the most recent copy of the data, but the copy in main memory is stale.
- Exclusive (E): The cache line is present only in this cache and matches the main memory content.
- Shared (S): Multiple caches have the same block, and all copies match main memory.
- Invalid (I): The cache line is not valid.

The state of the cache line in this protocol now depends on the state of the local cache line, the parallel caches' cache lines, and the state of the data in main memory. Without going over each specific state transition, generally it may be understood that any local writes without write-through will make that data the most recent and all other matching blocks stale, and local reads will cause state changes in the local and parallel caches depending on whether that data was being held exclusively or not.

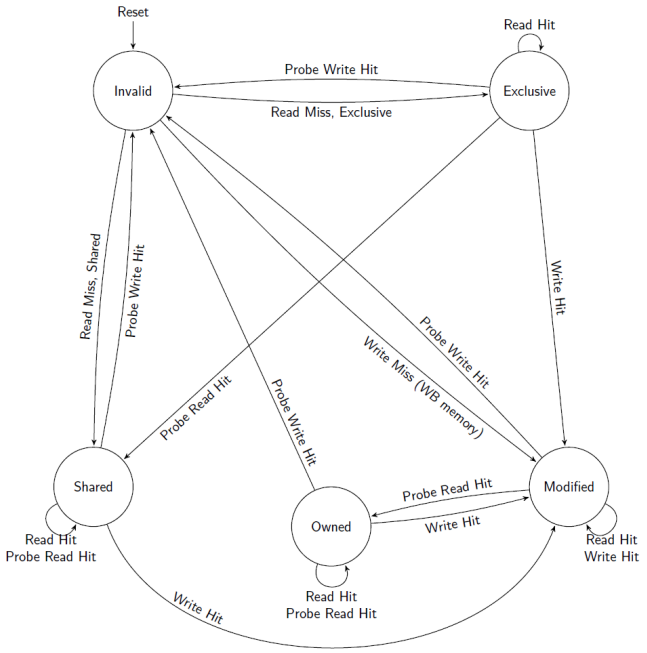


Fig. 2: State Transition Diagram for a MOESI Cache Coherency Protocol

III. Method

This implementation of the cache uses a 32KB 8-way set-associative L1 cache with 32-byte cache lines and a least-recently-used replacement algorithm for each cache set which writes-back evicted cache lines to main memory before replacing. Cache access in this model incurs a single-cycle latency, while main memory access incurs a 100-cycle latency.

Communication between modules relies on a transaction-level model that simulates port connections between each other using interface functions that pass arguments data between different modules. These connections facilitate communication between CPUs, L1 caches, Bus, and Main Memory as per Figure 3.

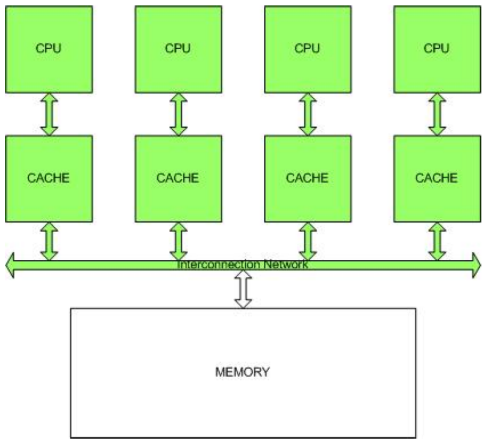


Fig. 3: Shared Memory System Diagram

The Bus arbitrates communication access between different modules by maintaining a FIFO queue that prioritizes Main Memory responses. Each module sends a request to the bus arbiter for access to the Bus on the positive clock edge, which is added to a queue and processed with a FIFO policy on the negative clock edge, notifying the next cache in line that the bus is open for their waiting request as per Figure 4.

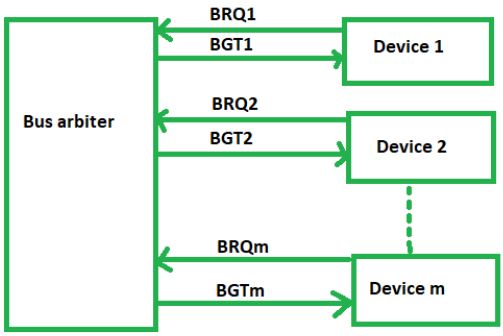


Fig. 4: Independent Request Bus Arbitration Method

Requests to the Bus are followed up by corresponding responses from the Bus whenever an action has been completed, triggering the next step in the local module processing which may trigger additional requests and responses.

In both the MOESI and the VALID-INVALID protocols, read requests from the CPU to the Cache will trigger a Cache Hit if a Cache Line with a matching address Tag is found and if the line is in a non-INVALID state, meaning the data can be directly read by the CPU from the Cache.

Otherwise, a read miss is triggered and a read request is made on the Bus which then determines if the data is snooped in another Cache or if it must be read from Main Memory. A response from the Bus will be sent to the Cache when the data is received from either another Cache or Main Memory, and the requester cache will then replace the least-recently-used cache line – writing-through the data to Main Memory in a VALID-INVALID protocol, or checking the state of the LRU line in the MOESI protocol and writing-back the data if it is MODIFIED or OWNED.

When a Write Hit occurs, both protocols will notify the Bus so that all snooping Caches may change their matching cache lines' states to be INVALID, but the VALID-INVALID protocol will use the write-through policy to once again write the data in both the Cache and the Main Memory.

A Write Miss in both protocols requires a read from Main Memory to *Write-allocate* the invalid/missed cache line. The VALID-INVALID protocol will read directly from Main Memory without looking for the data in other Caches, responding with the read data to the local Cache and completing a write to the Cache line without a write-through since the data is alread in Main Memory. The MOESI protocol will read and respond with the data from either the snooping Caches or Main Memory, then complete the same process as the Read miss when writing the read data to the local cache line.

This process tracks read hits and write hits/misses depending on the immediate result of the CPU read/write request, but when a read miss occurs, it considers a read miss to occur when a read from Main Memory is required and a read hit to result from a read from another Cache. These statistics are tracked and resulting hit rates are calculated alongside the total time spent completing pre-defined trace-files containing a list of read and write requests on specified addresses and CPU number specifications. Additional information collected during these trace file experiments includes the total amount of time each cache spends waiting for bus arbitration and the total number of times reads and writes from and to Main Memory are done.

System workload will be defined using trace files of various structures to test different aspects of the Cache Coherency Protocols, including robustness tests, edge case tests, and functional tests.

TABLE I: Trace File Experiments and Testing Purposes

Experiment	**Testing Purpose**
All WRITES (same address)	Worst-case, high contention, frequent invalidation.
All READS (same address)	Tests read-handling efficiency.
All WRITES (random addresses)	Benchmarks write-handling mechanisms.
All READS (random addresses)	Benchmarks read-handling mechanisms.
Alternating READS/WRITES (same address)	Edge case, frequent ownership changes.
Random READS/WRITES (random addresses)	Stress test for unpredictable workloads.
Matrix Multiplication	Realistic workload test.
Matrix-Vector Multiplication	Realistic workload test.
Fast Fourier Transform (FFT)	Realistic workload test.

IV. Results

The completely random read/writes on random addresses trace file described in Table I is useful for gaining a benchmark over multiple simulations as each randomized selection of requests can be averaged together for a general picture of system functionality. This process is used in Figure 5 for the VALID-INVALID protocol and Figure 6 for the MOESI protocol to show that the average hit rate and the average total bus arbitration waiting times are effectively equal across multiple CPUs.

This confirms that the Bus arbitration process is properly allocating the random read/write requests to

each processor uniformly and that the cache processing system is not prioritizing any individual cache over any others.

The results also show that this unstructured read/write process results in a high read hit rate and a low write hit rate for both protocols, with the VALID-INVALID protocol seeing an average write hit rate of 20.08% and read hit rate of 76.73% while MOESI sees a write hit rate of 22.23% and read hit rate of 80.83%.

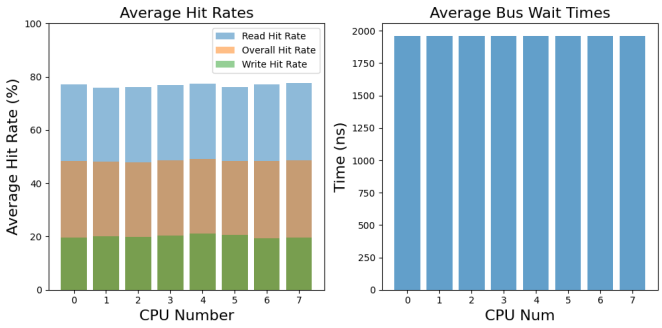


Fig. 5: Hit Rates and Bus Arbitration waiting times for 8 processors in in the VALID-INVALID protocol averaged over 10 simulations using 100000 different random read/write trace files for each

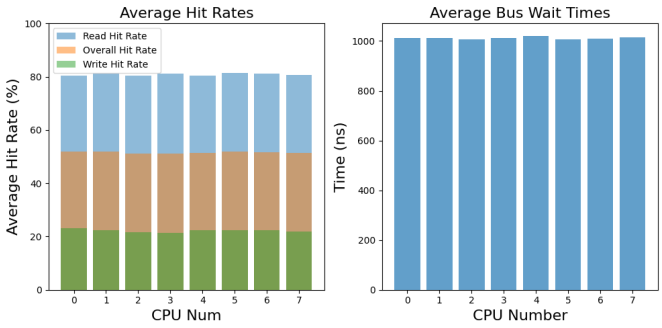


Fig. 6: Hit Rates and Bus Arbitration waiting times for 8 processors in the MOESI protocol averaged over 10 simulations using 100000 different random read/write trace files for each

By taking individual simulations of different trace files and averaging the hit rates over the different processors in Figure 7, it becomes possible to observe the cases in which the cache coherency protocol has an out-sized influence on the resulting hit rates of the system. The ”All Random” experiments match closely with the expected results from Figure 5 and Figure 5, but sees a divergence between the two protocols in the the writes on the same address test. All reads on the same address approaches a hit rate of 100% while the realistic workload traces see hit rates similarly close to 100%.

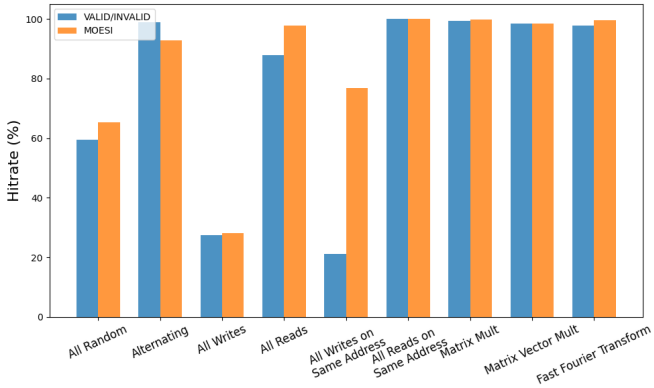


Fig. 7: Hit rate averaged over 8 CPUs for different trace file experiments over 100000 requests on both VALID-INVALID protocol and MOESI protocol

Observing the specific read hit rate in Figure 8 for each experiment shows that the read hit rate is generally high with MOESI doing better than the VALID-INVALID protocol in the random cases and the fast fourier transform case, but slightly worse in the alternating read/writes case.

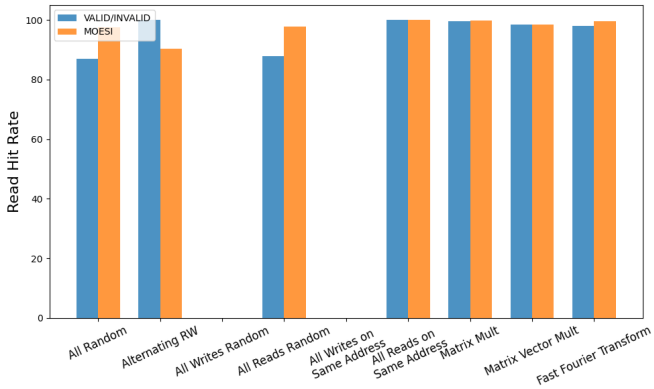


Fig. 8: READ Hit rate averaged over 8 CPUs for different trace file experiments over 100000 requests on both VALID-INVALID protocol and MOESI protocol

The write hit rate in Figure 9 shows source of the large difference in the case where all requests are writes on the same address, with a much larger write rate for the MOESI protocol than the VALID-INVALID. This behavior is a direct result of the new state additions to the MOESI protocol that allow for caches to share data rather than outright invalidating other cache's data outright, allowing for a larger hit rate in cases where writes occur on the same address across multiple caches.

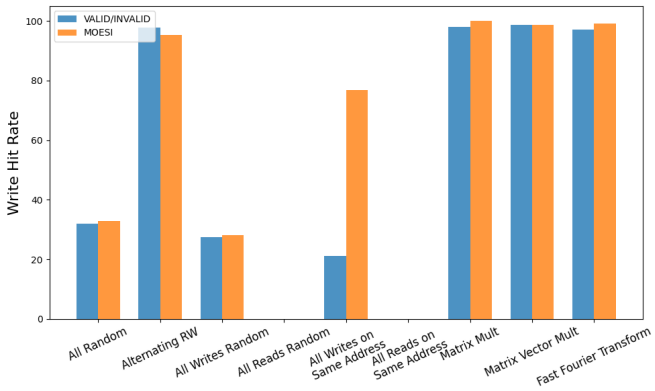


Fig. 9: WRITE Hit rate averaged over 8 CPUs for different trace file experiments over 100000 requests on both VALID-INVALID protocol and MOESI protocol

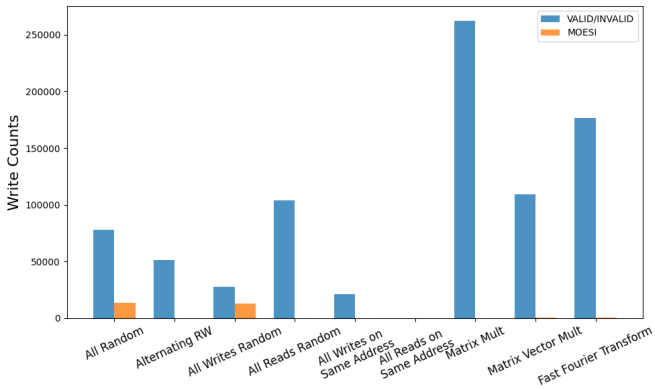


Fig. 10: Total Main Memory WRITES for different 8 CPU trace file experiments over 100000 requests on both VALID-INVALID protocol and MOESI protocol

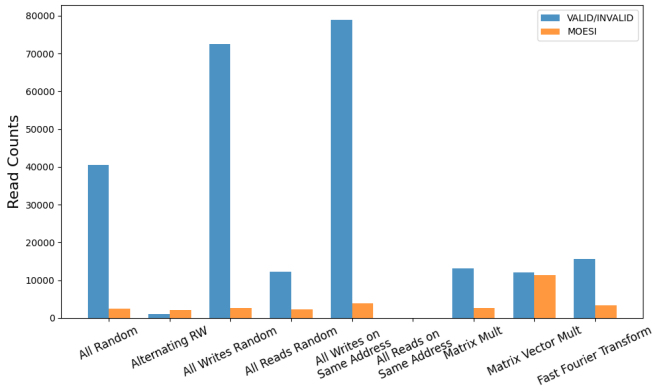


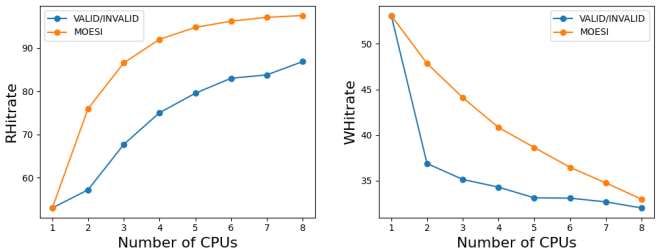
Fig. 11: Total Main Memory READS for different 8 CPU trace file experiments over 100000 requests on both VALID-INVALID protocol and MOESI protocol

The additional states for each cache line also allows for fewer accesses to main memory as observed in Figure 10 and Figure 11, which contributes to the the increased overall hit rate for the MOESI protocol as caches snooping reads are less likely to miss if there are fewer caches waiting for a response from main memory.

It also becomes apparent that the magnitude of reads-to-writes for the VALID-INVALID protocol differs substantially with realistic request workloads, incurring substantial writes to main memory in the range of 100000 to 250000. This is likely due to the nature of the calculations which would target many of the same addresses often with new data, causing many write-throughs to main memory due to write-hits. Additionally, the VALID-INVALID protocol sees an influx of reads from main memory for the ”random reads/writes” trace, the ”all writes on random addresses” trace, and the ”all writes on the same address” trace, likely caused by a coupling of write-allocations requiring reads from Main Memory and snooped invalidations on other caches.

With the understanding that different request orders and address accesses influence the resulting hit rates for both the MOESI and VALID-INVALID protocols, it becomes useful to return to the baseline test using random read/writes on random addresses to determine how these results compare between systems with different processor counts.

With the exploration of a range of CPU counts from 1 to 8, a trend is observable in Figure 12 such that an increase in CPUs sees an increase in Read hit rates and a decrease in Write hit rates. Starting with the initial single-CPU system as a baseline, the Hit rate behavior is the same on both protocols, which lines up with the expected behavior as the additional states of the MOESI protocol are only applicable in cases of shared address spaces. Both the Read and Write Hit rates are also noticeably worse for the VALID-INVALID case even though both protocols see similar behavior across multiple CPUs, showing an increased effectiveness of the MOESI protocol as expected from previous observations.



(a) Read Hit Rate (b) Write Hit Rate

Fig. 12: Average READ and WRITE Hit rates in $1-8$ CPU systems for VALID-INVALID protocol and MOESI protocol

With the increasing effectiveness of CPU counts for Read Hits and a decreasing effectiveness for Write Hits, taking the overall Hit rate in Figure 13 can shine light on the efficiency of adding CPUs to the system. The plot shows an upwards trend for the overall Hit rate in both protocols, although the MOESI protocol peaks around 5 CPUs with a Hit rate of 66.676% while the VALID-INVALID protocol seems to retain an increasing hit rate for a larger number of CPUs. The VALID-INVALID line apparently maintains its efficiency with increasing CPU count longer than the MOESI, but it overall fails to achieve the Hit rates present in the latter protocol.

The VALID-INVALID protocol also sees a sharp decline in hit rate at 2 CPUs, which seems to be due to a trend in both the read and write hit rates that does not continue when using a single CPU since the system would effectively return to the ”base” cache implementation without snooping, thus breaking down the state transition diagram.

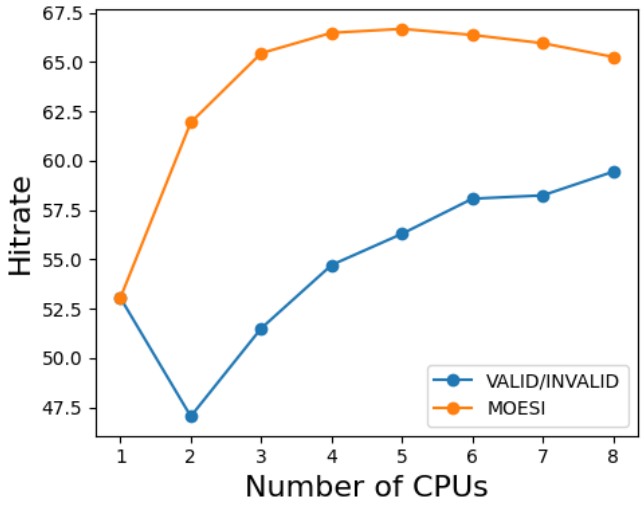
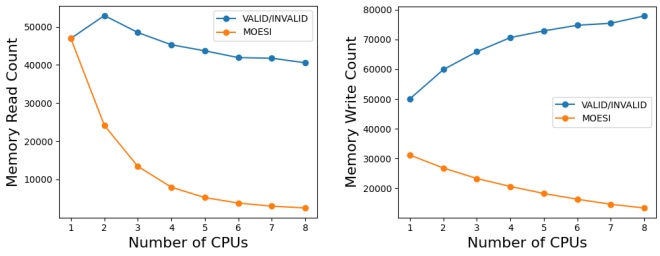


Fig. 13: Average Hit rates in $1-8$ CPU systems for VALID-INVALID protocol and MOESI protocol

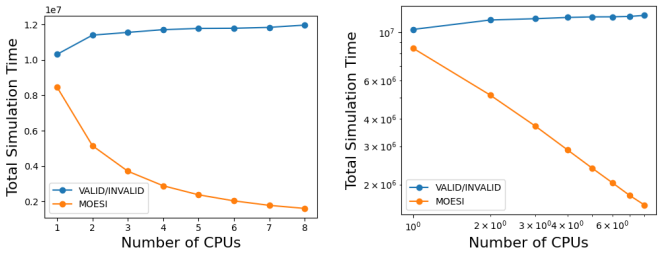
This inverse trend between Reads and Writes with an increasing number of CPUs does not hold up in the case of Reads and Writes to Main Memory in Figure 14 as the number of accesses to Main Memory greatly decreases with an increasing number of CPUs in the MOESI protocol. The inverse trend better describes the VALID-INVALID protocol as it sees a slight decrease in Memory Reads and a greater increase in Memory Writes. Additionally, the spike in data seen with 2 CPUs in only present in the Memory Reads plot, indicating that the trend was not interrupted when returning to a single CPU system, likely due to the write-through policy applying to single CPUs as well as multiple CPUs.



(a) Read Hit Rate (b) Write Hit Rate

Fig. 14: Main Memory READ and WRITE count in $1-8$ CPU systems for VALID-INVALID protocol and MOESI protocol

The number of memory accesses has an out-sized influence on the runtime for the workloads of the CPUs in the system as per Figure 15 where the increasing number of CPUs sees a longer runtime for the VALID-INVALID protocol and a faster runtime for the MOESI protocol. Using a log-log plot, a possibly scale-free behavior in the form of the Power Law may be present in the MOESI system, indicating some emergent behavior in the runtime caused by the MOESI protocol.



(a) Linear axes (b) Log-Log Axes

Fig. 15: Total Simulation time in $1-8$ CPU systems for VALID-INVALID protocol and MOESI protocol

V. Conclusion

These results demonstrate the importance selecting both the correct Cache Coherency Protocol as well as the most effective number of CPUs for system tasks to maximize Hit rates and minimize runtime. For many tasks, the ownership management and write-back policy of the MOESI protocol saw improved Hit Rates across-the-board, with only the alternating read-write on one address test case seeing an increased hit rate likely caused by the VALID-INVALID Write-through policy.

The write-through policy also showed the distinct disadvantage with the process of writing to Main Memory whenever a Cache Line is updated in the form of excessive Main Memory accesses leading to bloated runtimes.

On a completely randomized workload, it was also possible to observe this issue with the VALID-INVALID protocol runtime trending upwards with an increasing number of CPUs in the system. This coincides with an increasing hit rate with the increase in CPUs, but a lower hit rate across all CPU counts when compared to the MOESI protocol, indicating that the latter coherency protocol implementation is both more effective at Cache Hits *and* has a decreasing runtime with additional CPUs.

Understanding the behavior and performance of difference Cache Coherency Protocols is crucial for optimizing multi-core systems and involves deciding on cache mapping, write policies, and CPU number. The types of tasks being completed by the system also influence the outcomes of the selected protocol and the specifications of the system should be changed to reflect that – whether those tasks be continuous random requests, repeated requests on common addresses, or read/write-heavy workloads. Balancing consistency, performance, and scalability in computing environments is key when designing a multi-core system, and the Cache Coherency Protocol is but one of those pieces used to build the whole.