SystemC Multi-processor Simple Cache Coherency Protocol

Victoria Peterson - 15476758

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

I. Introduction

BASED on the single-level cache simulation from assignment 1, assignment 2 extends the model to a multiprocessor system with a simple coherency protocol, implementing support for multiple processors, each with its own local 32KB, 8-way set-associative LRU L1 cache, all connected by a shared Bus to a single main memory module. The protocol for cache coherence across processors involves a VALID-INVALID bus snooping protocol that requires that each cache monitors all memory requests on the bus, modifying its local cache state to maintain consistency across caches.

Each CPU cache continues to follow a write-back policy on dirty evicted cache lines and the write-allocate policy continues to require cache line reads directly from memory. Requests to main memory still have 100 ns of latency, but continued accesses to the bus for communication is not blocked due to a split-transaction mechanism.

This simulation will allow for performance evaluation of 1 to 8 CPUs using predefined trace files with measurements of key metrics such as cache hit and miss rates, main memory accesses, bus contention, and overall execution time.

II. Method

Instead of using the Cache to directly communicate with Main Memory as the only cache of the system, the new implementation adds a split-transaction Bus module to distribute communications to additional Caches or Main Memory. The CPU will send abstracted Reads and Writes to their local Cache which will then communicate their own tasks to the Bus such a Writes/Reads to/from Main Memory.

Tasks that require both a request and response now have their execution split into two parts in the split-transaction Bus such that requests from other modules are queued until processing is completed, at which point a response is added to another queue which is processing in parallel. Implementing these queues in each of the modules allows each to continue handling tasks while also communicating between each-other without blocking.

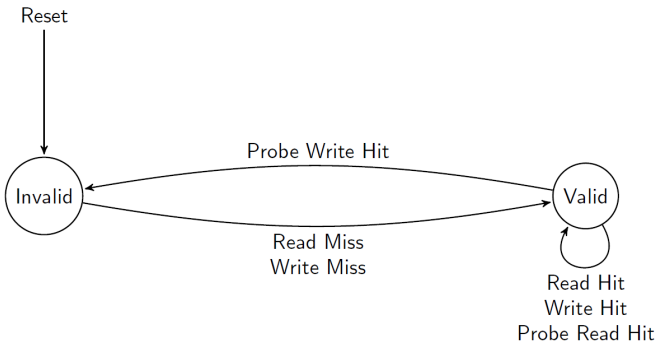

Fig. 1: Simple VALID-INVALID Cache Coherency Protocol state transition diagram

Additionally, a Simple Coherency VALID-INVALID Protocol is implemented as per Figure 1 where-in Cache Line states now depend on the state of other Caches. On top of the existing Cache Structure with Read/Write processing depending on Cache Line Hits and Dirty bits, an additional Valid bit is implemented that changes based on the state of the other Caches. This means that any CPU Read Miss or Read Invalid on a Cache Line requires a Read probe on all other Caches, sending a Read request to the Bus which is then ”snooped” by all of the Caches, letting the Bus know the correct data if it is present in any of them or reading directly from Main Memory if the data is not. A successful Read probe from another Cache makes the both of the matching Cache lines Valid, which may at worst cause all Caches to have copies of the same Valid data, but if a CPU write occurs on one of these Cache lines, a Write probe must be sent to the Bus in order to Invalidate all other Cache lines with the old data.

III. Results

By starting with the initial *dbg_p1* trace file of 100 CPU requests, Table I presents an initially abysmal view of the new system of Cache Coherency as no Read or Write Hits occurred over the 100 CPU requests

for the debug trace. This may first be thought as an issue with the code, but in reality it's a feature of the interaction between the Cache Coherency Protocol and the split-transaction Bus – the new system allows for non-blocking requests across the system using request and response queues, but the Cache Coherency Protocol has no way of maintaining coherency of the Cache based on requests that are in-flight or being processed by the Bus or Main Memory.

.	RHitrate	WHitrate	Hitrate	Sim Time
Assignment 1	47.5	43.33	45	5727 ns
Assignment 1	0	0	0	10001 ns

TABLE I: Hit Rates (%) for dbg_p1

So a Cache request that requires access to the Main Memory will be sent to the Bus and processed for 100 ns, but no blocking occurs and is thus able to continue to process CPU requests which may occur on the same Cache Line without knowing there is a request currently in Main Memory. An example of this would be two subsequent CPU Reads on the same Cache Line which Miss and require a Read directly from Main Memory: the first Read is sent to the Bus and the second read also sees a Miss and is also sent to the Bus even though the Line would be a Hit if the Cache waited for the response from Main Memory. This behavior is shown in Table II where-in approximately 100 Read requests on the same address are processed by the Cache as Misses while waiting for the first Main Memory request to respond with the requested data, thus making the Cache valid and letting the remaining 900 CPU Read requests be Hits.

Reads	RHit	RMiss	RHitrate (%)	Hitrate (%)
1000	896	104	89.6	89.6

TABLE II: 1000 Read only requests

This may be further expanded to an extreme case in Table III and Table IV where the same trace file containing 100000 randomly allocated Reads and Writes with random addresses sees a higher Hit rate of 53% in the implementation from Assignment 1 compared to the second assignment implementation where it is only 21.15%. This experiment shows a consistently lower Hit rate in a situation where the chance of re-selecting an address to Read or Write from would be lower and thus have less opportunity to require waiting for a Main Memory access response.

.	RHit	RMiss	WHit	WMiss
Assignment 1	26443	23575	26554	23428
Assignment 2	10549	39469	10602	39380

TABLE III: Processed Requests for 100000 random Reads/Writes on a Single CPU Run (Assignment 1 vs Assignment 2)

.	RH%	WH%	Hit%
Assignment 1	52.87	53.13	53.00
Assignment 2	21.09	21.21	21.15

TABLE IV: CPU Hit Rates (%) for 100000 random Reads/Writes on a Single CPU Run (Assignment 1 vs Assignment 2)

CPU	Reads	RHit	RMiss	Writes	WHit	WMiss
cpu_0	6271	28	6243	6229	28	6201
cpu_1	6211	25	6186	6289	16	6273
cpu_2	6264	19	6245	6236	21	6215
cpu_3	6220	19	6201	6280	31	6249
cpu_4	6239	28	6211	6261	26	6235
cpu_5	6223	21	6202	6277	22	6255
cpu_6	6308	25	6283	6192	21	6171
cpu_7	6277	17	6260	6223	24	6199

TABLE V: Processed Requests for 100000 random Reads/Writes on 8 CPUs

CPU	RH%	WH%	Hit%	MEMread	MEMwrite
cpu_0	0.4495	0.4495	0.448	99629	22699
cpu_1	0.4025	0.2544	0.328	99629	22699
cpu_2	0.3033	0.3368	0.32	99629	22699
cpu_3	0.3055	0.4936	0.4	99629	22699
cpu_4	0.4488	0.4153	0.432	99629	22699
cpu_5	0.3375	0.3505	0.344	99629	22699
cpu_6	0.3963	0.3391	0.368	99629	22699
cpu_7	0.2708	0.3857	0.328	99629	22699

TABLE VI: CPU Hit Rates (%) for 100000 random Reads/Writes on 8 CPUs

The same test can be expanded to multiple CPUs using the new Bus communication implementation and thus test the comparative Hit Rates for the different CPUs when splitting the same 100000 Read/Write requests between them. Looking at Table V and Table VI, it

Run Type	Read Hit Rate (%)	Write Hit Rate (%)	Overall Hit Rate (%)
All Reads on Same Address	96.34	N/A	96.34
All Writes on Same Address	N/A	96.34	96.34
All Reads on Random Addresses	0.389	N/A	0.389
All Writes on Random Addresses	N/A	0.412	0.412
Alternating Read/Writes on Same Address	96.34	N/A	96.34
Random Read/Writes on Random Addresses	0.364	0.378	0.371

TABLE VII: Average hit rates for different Traces over 100000 requests on 8 CPUs

also becomes apparent that the behavior of the system continues to result in Hit rates close to 0.5% across all the CPUs, requiring 99629 Read requests and 22699 Write requests to Main Memory over a total of 12355029 ns. This ultimately shows that the CPUs are properly distributing request load, but the individual Caches still lacks a way to coordinate when requests are pending in Main Memory, thus causing a pile-up of Main Memory latency that only ends up reinforcing itself.

Other than having more frequent accesses to the same Cache Lines, there seems to be little difference in the low Hit rates across different Read/Write paradigms in Table VII as accesses to the same address see Hit Rates approach 100% while randomized methods approach 0.

Additionally, while Caches process requests more quickly compared to the sequential blocking approach in Assignment 1, the total timing of these systems still balloon upwards due to the increasing number of Cache Misses that increase the number of Main Memory requests and thus the total overall processing time as the Cache must wait for responses from Main Memory. So while the Cache is able to finish request processing quickly, the system as a whole must still wait for the rest of the system to finish responding.

IV. Discussion

A possible solution that I spent much of my time on due to a personal misunderstanding of assignment specifications was the use of a local Cache data structure to track sequential Read/Write requests. This implementation was based on the use of unordered maps to map queues of pending requests to their target addresses – if then a CPU request targeted an address with an already pending request, it would get pushed to the associated queue instead of going to the general Request queue.

When a Bus response was received for a completed pending Read, the queue would remove all the Read requests in the front of the queue and consider each a Read Hit while the next Write in the queue would be immediately sent to the request queue. Likewise, when a Bus response was received for a completed pending Write, the next Read or Write in the address-mapped pending queue would be sent to the general Request queue, while any invalidations originating from snooping other queues would be pushed to the front of their address-mapped pending queue.

This allows each address to act as its own sequential program-order request queue while other requests on other addresses may be addressed without blocking.

V. Conclusion

The Simple VALID-INVALID Cache Coherency Protocol gives the ability to retain coherency between difference modules in a perfect system where every Cache Line state in every Cache is know instantly and requests are sent and received in perfect program-order, but what we see in actuality is a mess of requests and responses interspersed between multiple Caches, the Bus, and Main Memory. This simple model lacks cohesiveness as each has ways to communicate state changes through snooping the Bus, but none are able to keep track of requests that have already been sent into the system, leading to a failure to maintain the correct Cache Line states and a snowballing growth of Main Memory accesses across any number of CPUs. There are ultimately some possible solutions in the form of local Cache request tracking or additional Cache state variables which may allow for these systems to maintain program-order and minimize overhead across the entire system of modules.