Parallel Systems Architecture
Assignment 3

Saidgani Musaev
11992077

February 4, 2019

1 Introduction

In current paper we discuss solution for the last assignment of "Parallel Systems Architecture" course. The task is to implement MOESI cache coherency protocol. Protocol is described in slides and in the book, so we don't look into description and correctness of the protocol in this report. On the other hand, we discuss implementations details and corner cases.

2 Implementation

In current protocol every cacheline can be in one of the 5 following states:
- Modified
- Owned
- Exclusive
- Shared
- Invalid

transitions can happen between some of states on some events. We implement write-back cache which implements exactly all transitions as they are described in the paper.

However, paper doesn't tell anything about **Shared-Modified** transition with write-back policy. We implemented separate operation(INVALIDATE) which transfer a cacheline first to MODIFIED/EXCLUSIVE state, by sending invalidate request to all owners of the cacheline, and only then perform local modification. In some cases we synchronize cores on negative edge to let first snooping thread invalidate cacheline(if invalidation has to be done). We implement cache-to-cache communication to reduce number of cycles of bringing to cache operation(READ operation). Communication between caches happens via the same bus and the same wires, which are being used for regular memory operations. We perform cache-to-cache operation in 2 cycles(the first cycle - other cache gets the value, the second - it sends the value). Cache-to-cache messages has to have the highest priority, to not be lost. We added special SourceID wire, specifically for this cache-to-cache communications. This wires let know to the cache "who" is the source of data(other cache or DRAM). Based on

source , cache can make state shared(other cache) or exclusive(dram). Here is one corner case presented - what if two cores are requesting the same cacheline from the DRAM? We have to keep in mind that one request will occur before the other. So the core, "whose" request is approved first can send cache-to-cache message to the second core, that it also requested the cacheline, but it's not delivered yet. Both of the cores will wait for the response from DRAM, but both of them will know that cacheline state is shared now. That's exactly what implementation does do. We introduced special operation type FUNC_REQUESTED and special state CACHEL_REQUESTED, which notify that cacheline is requested, but not valid yet. So every cacheline holder(if it's in valid state) can modify this cacheline, but first invalidate signal has to be sent. Of course on every cycle of acquiring the bus, invalidating thread checks that it's copy is still valid - otherwise it has to pull actual copy from the memory/other cache again and then repeat the process again(I am sorry for implementing this logic with a lot of goto statements, but they are very local and readable, also I don't modify stack in those "goto" code areas).

We perform all cache-to-cache communication only on negative edge of the clock, because of some other core can invalidate the cacheline on positive edge. It should be done really precisely.

We didn't modify number of threads in the solution, but we create dynamic thread on every cache-to-cache message(trying to paralleling it by simulating special hardware for it.). Dynamic threads are being created via sc_spawn.

Everything else is happening according to the protocol description.

3 Experiments and Conclusion

Current solution with one core gives the same output as our first assignment, so nothing surprising there.

However, with more processors we can see different.

CPU	Reads	RHit	RMiss	Writes	WHit	WMiss	Hitrate
0	4113	22	4091	4030	35	3995	0.699988
1	18318	0	18318	18440	0	18440	0.000000
2	25834	21	25813	25470	30	25440	0.099407
3	29549	1275	28274	28801	1180	27621	4.207369
4	31366	2	31364	30548	1	30547	0.004845
5	32015	12	32003	31782	26	31756	0.059564
6	32327	1626	30701	32285	1653	30632	5.074909
7	32536	1687	30849	32584	1771	30813	5.310197

Random tracefile with 8 cores VALID-INVALID protocol

CPU	Reads	RHit	RMiss	Writes	WHit	WMiss	Hitrate
0	4113	22	4091	4030	35	3995	0.699988
1	18318	0	18318	18440	0	18440	0.000000
2	25834	21	25813	25470	30	25440	0.099407
3	29549	1275	28274	28801	1180	27621	4.207369
4	31366	2	31364	30548	1	30547	0.004845
5	32015	12	32003	31782	26	31756	0.059564
6	32327	1614	30713	32285	1647	30638	5.047050
7	32536	1681	30855	32584	1765	30819	5.291769

Random tracefile with 8 cores MOESI protocol

CPU	Reads	RHit	RMiss	Writes	WHit	WMiss	Hitrate
0	4956	1107	3849	3154	742	2412	22.799014
1	3983	822	3161	2890	544	2346	19.874873
2	3997	868	3129	2703	502	2201	20.447761
3	4043	859	3184	2695	501	2194	20.184031
4	4055	852	3203	2662	476	2186	19.770731
5	4055	847	3208	2733	513	2220	20.035357
6	4074	828	3246	2710	490	2220	19.428066
7	4124	833	3291	2705	478	2227	19.197540

FFT tracefile with 8 cores VALID-INVALID protocol

CPU	Reads	RHit	RMiss	Writes	WHit	WMiss	Hitrate
0	4956	1107	3849	3154	748	2406	22.872996
1	3983	825	3158	2890	546	2344	19.947621
2	3997	867	3130	2703	499	2204	20.388060
3	4043	856	3187	2695	501	2194	20.139507
4	4055	849	3206	2662	468	2194	19.606967
5	4055	846	3209	2733	512	2221	20.005893
6	4074	827	3247	2710	489	2221	19.398585
7	4124	831	3293	2705	478	2227	19.168253

FFT tracefile with 8 cores MOESI protocol

As we can see MOESI improves hitrates on some cores, but it on the other cores hitrates are becoming worse. But average time for bus acquisition changes from 30ms to 700ms for 8-cores, which is really bad. This demonstrates how overloaded bus in MOESI and what are the trades-off of MOESI protocol. Execution time also became more with implementing MOESI. With less cores we can see very similar results.

CPU	Reads	RHit	RMiss	Writes	WHit	WMiss	Hitrate
0	11274	1752	9522	6553	945	5608	15.128737
1	9417	1463	7954	6083	758	5325	14.329032
2	9834	1527	8307	5523	613	4910	13.935013
3	9881	1510	8371	5972	704	5268	13.965811

FFT tracefile with 4 cores VALID-INVALID protocol

CPU	Reads	RHit	RMiss	Writes	WHit	WMiss	Hitrate
0	11274	1753	9521	6553	949	5604	15.156785
1	9417	1464	7953	6083	759	5324	14.341935
2	9834	1528	8306	5523	612	4911	13.935013
3	9881	1509	8372	5972	704	5268	13.959503

FFT tracefile with 4 cores MOESI protocol

Here we see how average time for bus acquisition changes from 26ms to 280ms

Of course we cannot surely say that MOESI is better or worse here, we need to perform more tests and measuring with more representative files, however even now we can see how costly it is to implement MOESI protocol.