ECE #506

Paralleled Radix Sort with OpenMP

Zhiping Wang | Unity ID: zwang79 | Student ID: 200265085

11.24.2019

Objective

Implement a trace driven SMP simulator (shared multiprocessor simulator) that is able to work in a multiprocessor environment in C++. Build various coherence protocols on top of it.

Introduction

This report contains code explanation, design ideas, required experiments and analysis.

Code explanation

Makefile: I added a method to run all the required experiments.

```
report:
    mkdir -p results
    ./smp_cache 262144 8 64 4 0 ../trace/canneal.04t.longTrace -> results/cache_size_256K_MSI.log
    ./smp_cache 262144 8 64 4 1 ../trace/canneal.04t.longTrace -> results/cache_size_256K_MESI.log
    ./smp_cache 262144 8 64 4 2 ../trace/canneal.04t.longTrace -> results/cache_size_256K_Dragon.log
    ./smp_cache 524288 8 64 4 0 ../trace/canneal.04t.longTrace -> results/cache_size_512K_MSI.log
    ./smp_cache 524288 8 64 4 1 ../trace/canneal.04t.longTrace -> results/cache_size_512K_MESI.log
    ./smp_cache 524288 8 64 4 2 ../trace/canneal.04t.longTrace -> results/cache_size_512K_Dragon.log
```

This method creates a “results” folde, and then runs all the experiments.

main.cc: I added some configuration functions and improved the main function.

cache.h: add some functions and variables.

cache.cc: function improvements.

The way to generate result is:

1. “make”
2. “./smp_cache 8192 8 64 4 0 ../trace/canneal.04t.debug” or “make report”.

Design ideas

I will explain a little bit about the datapath in this section.

```
while (f.good()) {
    op = 'e';
    f >> cache_id >> op >> hex >> addr; // read input file line by line
    if (op == 'e') break;
    COPY_EXIST = false;

    L1_cache[cache_id]->Access(addr, op);

    BusOp = L1_cache[cache_id]->BusRequest();

    for (int i = 0; i < num_processors; ++i) { // bus snoop
        if (i != cache_id) {
            COPY_EXIST = (L1_cache[i]->BusOp(addr, BusOp))? 1 : COPY_EXIST;
        }
    }
    L1_cache[cache_id]->CopyResponse(COPY_EXIST, addr);

    if (protocol == 2) { // only Dragon has second bus request
        BusOp = L1_cache[cache_id]->BusRequest();
        for (int i = 0; i < num_processors; ++i) { // bus snoop
            if (i != cache_id) L1_cache[i]->BusOp(addr, BusOp);
        }}}
```

Access(addr, op) will execute cache functions of which is being accessed.

BusRequest() will put the bus request proposed by the accessed cache on common bus.

BusOp(addr, BusOp) is the snooping behavior. And this function will be run by all the peer caches.

CopyResponse(COPY_EXIST, addr) is designed for those pending states transitions that can be affected by copies in peer caches (MESI, Dragon).

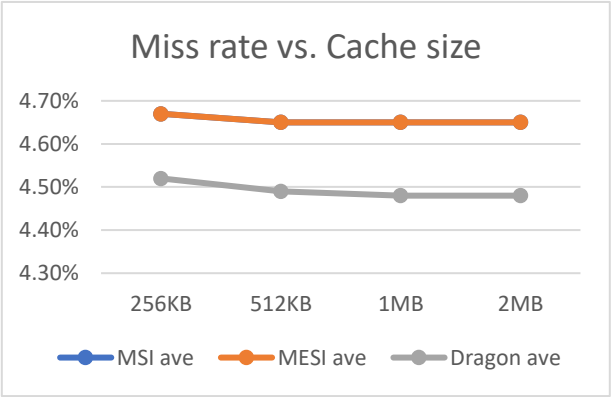
If the protocol is Dragon, there will be a second bus snooping for all peer caches.

Required experiments

1. Cache size: vary from 256KB, 512KB, 1MB, 2MB while keeping the cache associativity at 8 and block size at 64B.

These are the results for the average cache miss rate for 3 protocols under different cache size configuration:

Cache size	MSI ave	MESI ave	Dragon ave
256KB	4.67%	4.67%	4.52%
512KB	4.65%	4.65%	4.49%
1MB	4.65%	4.65%	4.48%
2MB	4.65%	4.65%	4.48%

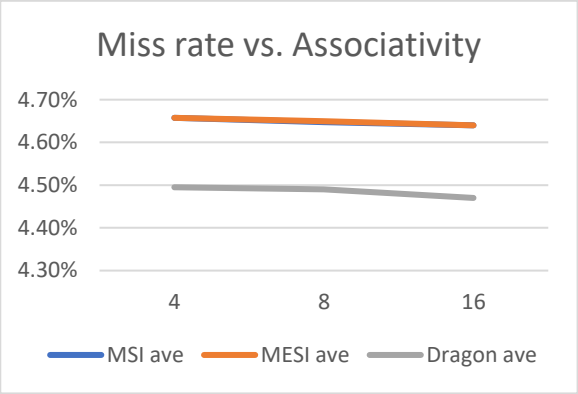


With cache size increasing, miss rate decrease accordingly. Though the reduction is not very notable.

Also, as one can see, miss rates of MSI and MESI are the same, which makes sense because the extra state E does not affect whether the access is hit or not.

2. Cache associativity: vary from 4-way, 8-way, and 16-way while keeping the cache size at 1MB and block size at 64B.

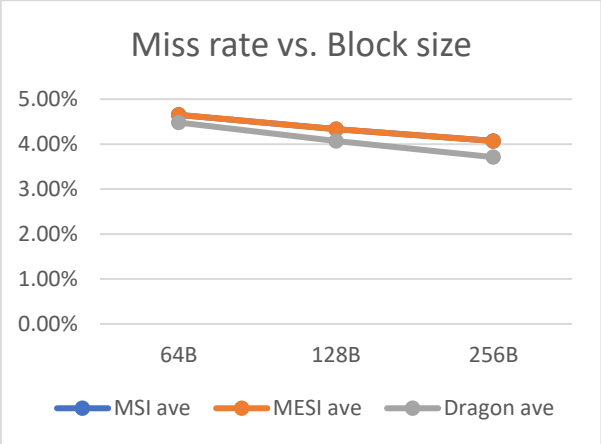
Associativity	MSI ave	MESI ave	Dragon ave
4	4.66%	4.66%	4.50%
8	4.65%	4.65%	4.49%
16	4.64%	4.64%	4.47%



With associativity increasing, miss rates of all protocols decrease.

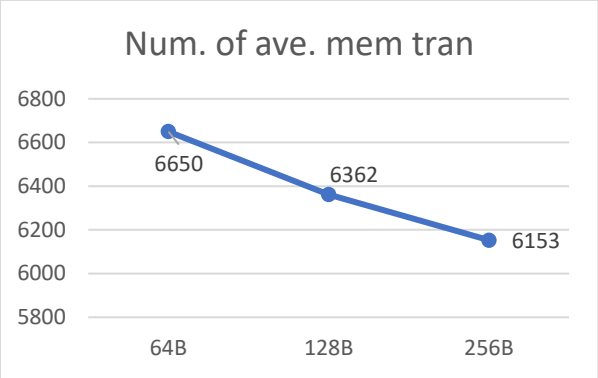
3. Cache block size: vary from 64B, 128B, and 256B, while keeping the cache size at 1MB and cache associativity at 8 way.

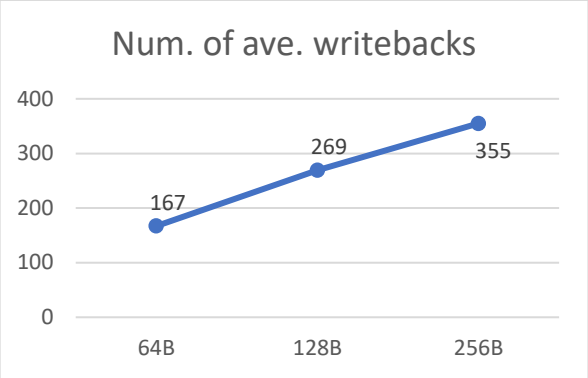
Block size	MSI ave	MESI ave	Dragon ave
64B	4.65%	4.65%	4.48%
128B	4.33%	4.33%	4.07%
256B	4.07%	4.07%	3.71%



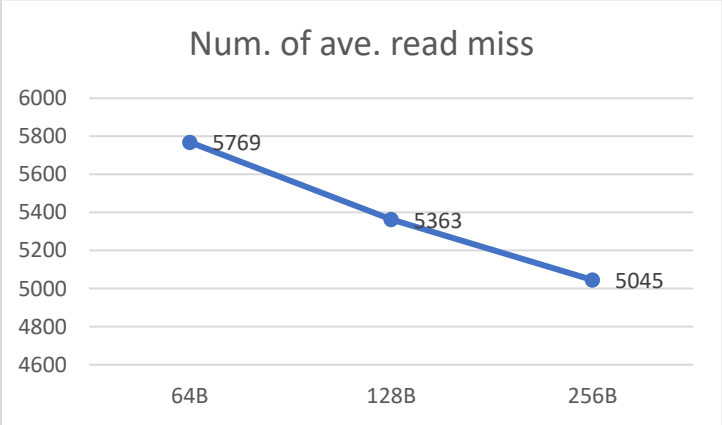
With block size increasing, miss rates of all protocols decrease. Besides, the reduction of miss rate is significant compare to cache size change and associativity change.

Here are some further discussions on memory transaction and writebacks for different block sizes.





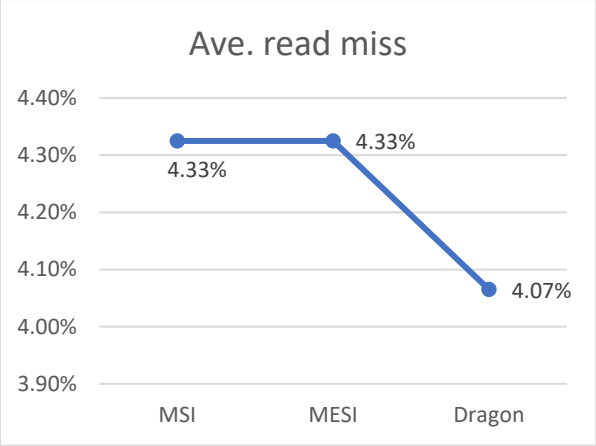
As one can see, the number of memory transaction decreases with the increase of block size, which makes sense, because most of the senarios when memory transaction happens remain almost the same except for read miss. When it comes to a read miss, memory controller needs to fetch block from memory thus causes memory transaction. And if we look at the trend of read miss as follows:

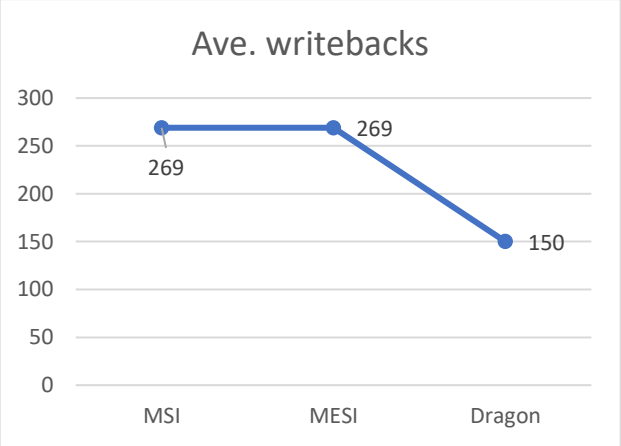


We notice that the decrease of read misses is basically the same as the decrease of memory transaction. However, if we do the math, the decrease of memory transaction is less than that of the read misses in a small scale: When block size increase from 64B to 128B, number of memory transaction decreases from 6650 to 6362 (-288). While the difference of read misses is 5769 – 5363 = 406. I believe the reason behind this is the increase of writebacks: write back operation also causes memory transaction. Number of writebacks increases from 167 to 269 with the change of block size, and the difference (102) roughly conpensate the difference between memory transactions and read misses.

4. Cache protocols: keeping the cache size at 1MB, cache associativity at 8 way and the block size at 128B. Compare cache performance under different protocols. Due to incorrect assumption made by the author, some of the data collect in Dragon protocol, like memory transactions and cache to cache transfers, is not valid. I did not take those into account.

   (1) Miss rate and Flush of all protocols:





The miss rates of MSI and MESI are equal, as I explained before. The E state does not make a difference on hit or miss. However, there is a significant miss rate reduction for Dragon protocol. I believe this is because of the existence of Sm state. A write operation will change the M state blocks in peer caches into Sm state without writeback, which means number of writebacks decreases (as shown above). For the case of MSI and MESI, M and S will go into invalid states on write operations which increases the miss possibility on following operations.

This also causes the reduction in number of flushes of Dragon protocol (average flushed of 7 compared with 149 in MSI and MESI).

Protocol	ave. flushes
MSI	149
MESI	149
Dragon	7

(2) Memory transactions and interventions in MSI and MESI.

Protocol	ave. mem tran	ave. interv
MSI	6362	110
MESI	1542	1367

As we can see, compared with MSI, the number of memory transaction decrease rapidly in MESI while the number of interventions increase. This is mainly because of the mechanism of FlushOpt, in other words, read miss can be satisfied by peer caches when copies exist. When read miss occurs in MSI, memory controller will fetch block from memory. When read miss occurs in MESI, copies in peer cache can provide the block with a FlushOpt without the participant of memory, thus memory transaction reduces. However, blocks in M or E state will go into S state (instead of I in MSI) during this process. As a result, intervention in MESI happens more often.

Conclusion

This project is fairly straight-forward once understood how different protocols work. Due to the "shortcut" provided by TA, implementation of Dragon protocol becomes easier than MESI.