Parallel Systems Architecture
Assignment 2

Saidgani Musaev
11992077

January 28, 2019

1 Introduction

In current paper we discuss solution for the second assignment of the "Parallel Systems Architecture" course. The main goal of the second assignment is building basic cache coherency protocol with up to 8 cores. All cores share the same memory, but each core has private cache. Communications between cores is maintained by interconnection bus. This is **VALID-INVALID** protocol with **WRITE-THROUGH** policy. Since we have one bus, we have to implement basic arbiter. We first discuss some changes of previous assignment solution(because of it was incorrect), then we will move to the cache coherency itself.

2 Previous assignment

Since previous assignment contained some errors, we have to first fix them.

The first error was Indexing and Tagging problem. My solution wasn't implementing real 8-way cache sets. Instead of that my solution was implementing 8 cache sets with many entries inside. It made my cache less associative and performance was lower. I changed indexing and tagging to the following scheme:
Bits [0-4] - offset of byte within current cacheline;
Bits [5-11] - index which points to cache set;
Bits[11-31] - these bits are used for tagging cacheline;

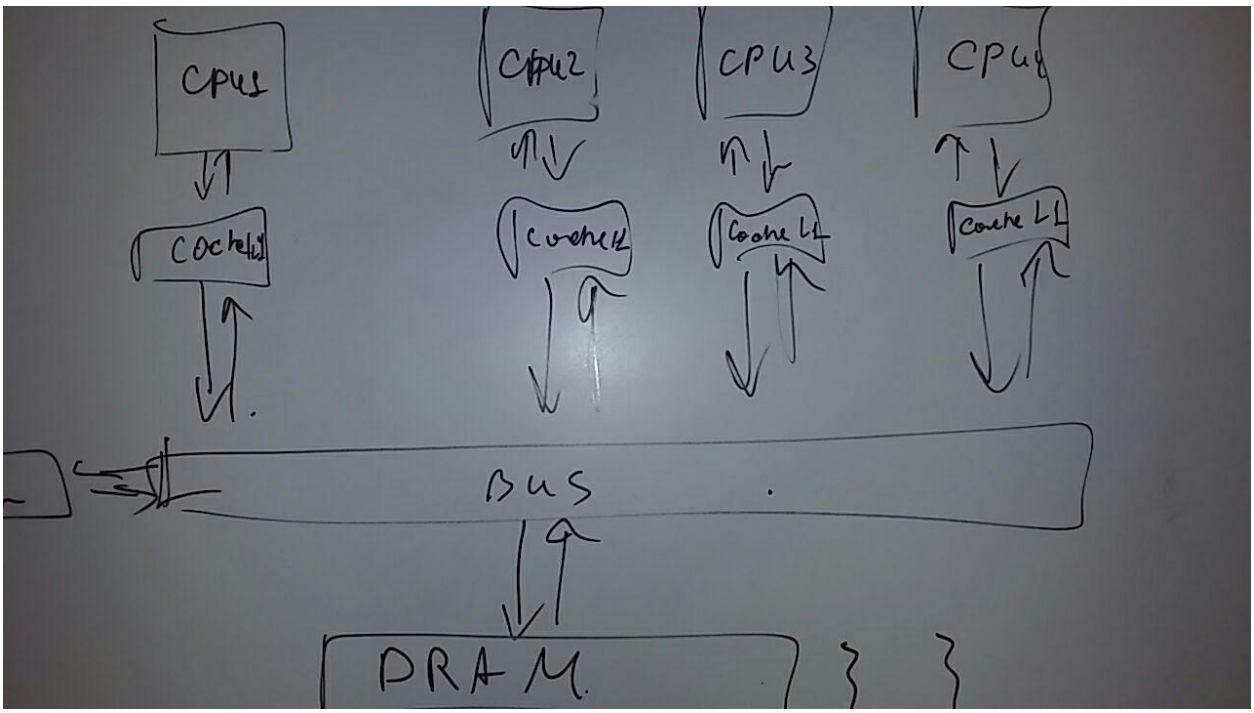
```
#define CACHEL_VALID        0x1
#define CACHEL_DIRTY        (0x1 << 1)
#define CACHETAG_MASK       (~0b111111111111)   // 0b11..10000000000000000
#define CACHEINDEX_MASK     0b111111100000
#define CACHETAG_SHIFT      12
#define CACHEINDEX_SHIFT    5
```

The second error was not considering dirty-bit state of the cacheline before writing back phase. Which costed extra cpu cycles.

The third mistake was in statistic. Program wasn't calculating writemiss and readmiss correctly.

The tarball which I applied to this report contains fixed version of the previous assignment with all fixed errors, to demonstrate that new assignment is based on correct version of basic Single cache implementation.

3 Cache coherency protocol



3.1 Overview

Our implementation of cache coherency protocol is based on simple split-transaction bus and pipelined memory.

Each CPU connected directly only to local cache, when cache is connected to the bus. We didn't add external arbiter to the system. Our arbiter is embedded into the bus, which makes bus a hierarchical channel. Arbiter is implemented using standard c++ mutex(which simulates hardware lock mechanism). We have separate Memory module, which accepts requests and put them into queue. accepting requests occurs via snooping the bus.

Our solution for the previous assignment was implementing real data transfer. We removed this part just to not over-complicate simulation with extra cycles(or using more

wires to transfer cacheline in one cycle). But we transfer addresses, cpu identifiers and function identifiers(READ, WRITE, NOP) over the wires to demonstrate that everything is strongly synchronized on clock cycles. Our implementation is semi-TLM, semi-classical SystemC(Everything is implemented using classical SystemC primitives, except part which writes to the Bus channels, this part is implemented using TLM).

Simulation of memory pipeline is done by just reducing clock cycles number, when there are several memory requests in the "pipe"(C++ std::queue class). First access takes in 100 cycles, when all the rest take 10 cycles per request.

3.2 Bus

Bus is implemented as C++ class which SystemC channels. All writing and reading to/from the channels are done via Bus class interface.

```
class Bus_if : public virtual sc_interface
{
  public:
    // These methods needed for CPUs to send requests
    virtual bool read(int proc_id, int addr) = 0;
    virtual bool write(int proc_id, int addr) = 0;
    virtual void wait_for_response(int proc_id) = 0;
    virtual struct request wait_for_any(int proc_id) = 0;

    // These methods needed for memory module to put high
    // priority responses on the bus and get requests
    virtual void memory_response(int proc_id, int addr) = 0;
    virtual struct request get_next_request() = 0;
    virtual void memory_controller_wait() = 0;
};
```

Bus is able to transfer different "messages"(when we mention message we mean chunk of data such as address, function, processor id). For example:

- READ - request from one of the processor to the memory slave module
- WRITE - request from one of the processor to the memory slave module
- RESPONSE - response from the memory module to one of processor.

As it's mentioned in manual, RESPONSE messages have priority over any READ/WRITE request. This part is implemented by having one extra atomic flag which forbids any processor to make bus busy, when this flag is risen. As soon as RESPONSE transfer is done - READ/WRITE requests could make bus busy again.

All modules listen to the bus channels in many different phases(waiting for response of the memory module, sending any request from core to memory module), so we use common *wait(event)* technique. We have faced that fact that all modules are shareing the same bus - bus becomes very "noisy" and sometimes some modules could be "awaken" by accident. To avoid it we changed Function enum type by adding FUNCTION_NOTHING == 0, to detect such "empty messages" and to not handle them.

3.3 Snooping

All snooping of each module are implemented in separate threads(memory snooping thread, cache snooping thread). Each such thread is listening the bus and handle every message according to protocol.

Cache snoopy thread is looking for all writes in a bus, which are sent by other processors and invalidate respective cacheline(if such is presented in private cache).

Memory snoopy thread is just listening for memory requests and handle them by adding to the queue.

3.4 Write-Through

To implement Write-Through cache policy We just added immediate write-back after changing memory locally. If memory is not cached, we still first bring the cacheline from the cache, modify it locally, then performing write-back phase.

3.5 Statistic measuring

To measure statistic of main memory access rate and average bus acquisition time I used following code snippet in Cache module:

```
for(int ii = 0; ii < 20; ++ii)sched_yield();
++_main_memory_access_rate;
clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &time1);
__main_memory_access_request();
clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &time2);
double result = (time2.tv_sec - time1.tv_sec) * 1e6
+ (time2.tv_nsec - time1.tv_nsec) / 1e3;
_time_for_bus_acquisition += result;
```

I use two global variable for calculating the statistic. I also call *sched_yield()* 20 times before measuring it to get more precise results.

4 Experiments and Conclusion

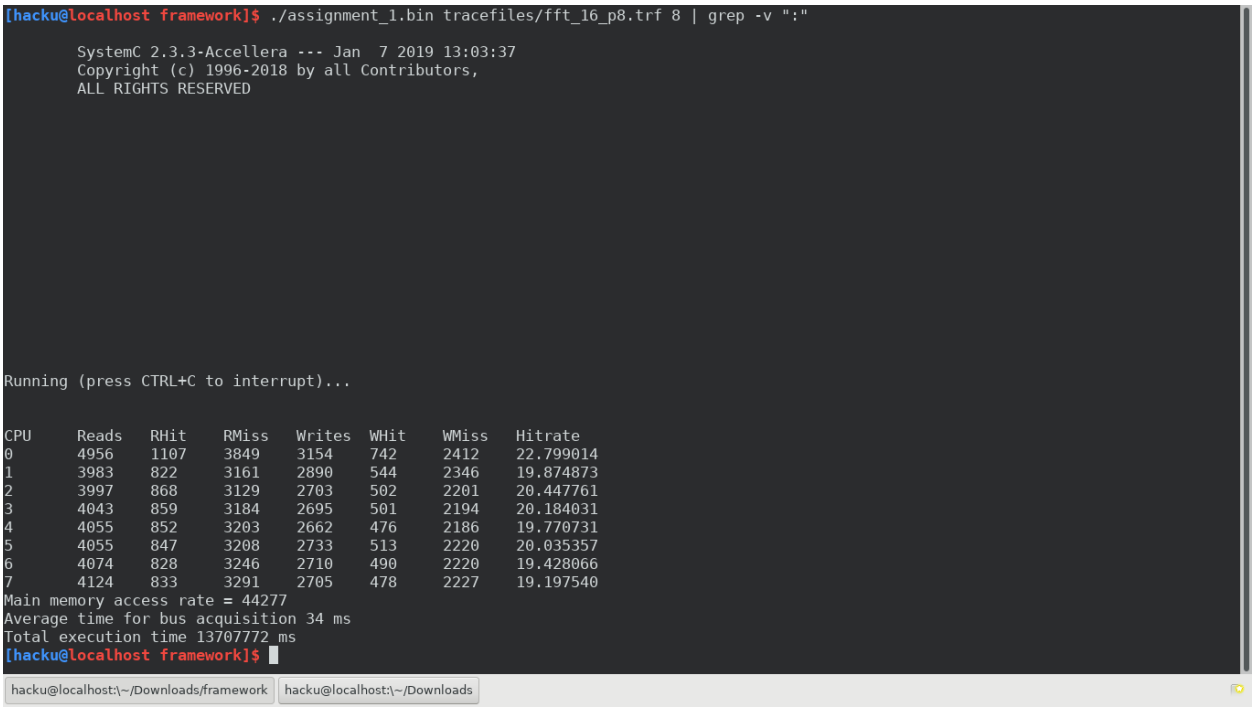
In this section we present results of simulation and share some conclusions with you.

```
[hacku@localhost framework]$ ./assignment_1.bin tracefiles/fft_16_p2.trf 2 | grep -v ":"

        SystemC 2.3.3-Accellera --- Jan  7 2019 13:03:37
        Copyright (c) 1996-2018 by all Contributors,
        ALL RIGHTS RESERVED



Running (press CTRL+C to interrupt)...

CPU     Reads   RHit    RMiss   Writes  WHit    WMiss   Hitrate
0       29313   3423    25890   15417   2013    13404   12.152918
1       29262   3176    26086   14801   1897    12904   11.513061
Main memory access rate = 78284
Average time for bus acquisition 21 ms
Total execution time 13998562 ms
[hacku@localhost framework]$
```

hacku@localhost:\~/Downloads/framework | hacku@localhost:\~/Downloads

```
[hacku@localhost framework]$ ./assignment_1.bin tracefiles/fft_16_p4.trf 4 | grep -v ":"

        SystemC 2.3.3-Accellera --- Jan  7 2019 13:03:37
        Copyright (c) 1996-2018 by all Contributors,
        ALL RIGHTS RESERVED



Running (press CTRL+C to interrupt)...

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
Main memory access rate = 55265
Average time for bus acquisition 26 ms
Total execution time 12347595 ms
[hacku@localhost framework]$
```

hacku@localhost:\~/Downloads/framework | hacku@localhost:\~/Downloads



Statistically our simulation shows worse hitrate than previous solution(with Write-Back policy). But it's hard to estimate how write-back caches would behave with current tracefiles and these numbers of cores. Average time for bus acquisition and Total execution time are represented in microseconds.

In current assignments we have faced several problems and implemented different technique to solve them(Transaction Level Modeling, Classical SystemC modeling, C++ standard library, multithreading, relaxed memory consistency model).