Georgia Institute of Technology

CS4290/CS6290/ECE4100/ECE6100

Programming assignment #4
Progm due: Nov. 18 (Sun) 6:00 pm T-square (+5:55 grace period)
Prof. Hyesoon Kim
Prof. Moinuddin Qureshi

This is an individual assignment. You can discuss this assignment with other classmates but you should do your assignment individually.

Part 1 (80 points) Simulator
Overview
In this assignment, you will extend your simulator to support fine-grained MT.

We need multiple steps to support the MT feature. Adding MT feature requires modifications in multiple places in the simulator. Additional data structures must be added. add_me4.txt file and lab4.tar.gz (contains only knobs for the time being) are provided

First, you need to make the simulator run correctly right after you add add_me4.txt file. add_me4.txt file changes get_op function and adds thread_id into op_struct, additional knobs and modify stat prints. You need to convert your stat data structures to be arrays. (Note: use HW_MAX_THREAD to convert a scalar variable to an array variable. i.e.) dcache_miss_count --> dcache_miss_count[HW_MAX_THREAD]. The difference between HW_MAX_THREAD and KNOB_RUN_THREAD_NUM is that HW_MAX_THREAD is the maximum number of threads that the hardware can support and KNOB_THREAD_NUM is the number of running threads at a given time. The hardware can support 4 threads but a user can decide to run 1 ~ 4 threads.)
Because now the system handles multiple traces, before you add MT feature, you need to make it sure that your simulator still runs one thread just like before and then you add features to support multi traces.

In a real architecture simulator, simulation ending condition should be more sophisticated. However, in this assignment, we do not change the ending conditions. Therefore, simulator reads from only remaining traces until all the traces are finished. max_inst_count is based on the sum of all threads.

- *Fetch Stage*
  - Fetch needs to fetch from multiple threads.
  - get_op(Op *) function is now updated to fetch from multiple threads.
  - Deciding which thread to fetch is also a research topic. Several papers have been proposed to increase processor's performance. In this assignment, we just simply use a

round-robin fashion. add_me4.txt file already has this feature.
- op->thread_id shows thread id of each op.
- After fetching, the simulator accesses a branch predictor just like before.
- In the real hardware, the hardware needs multiple PCs.
- You need to have different GHR for each thread. However, a 2-bit counter table is shared among all threads

- Branch misprediction handling:
  - When a thread is mispredicted, you must set br_stall[op->thread_id] = true.
  - When the mispredicted instruction is resolved in the memory stage, you should reset br_stall[op->thread_id]=false at the write-back stage.
  - get_op function checks br_stall and if there is a misprediction, it doesn't fetch instructions from the mispredicted thread.

- *Decode Stage*

  - We need to have multiple of reg, so we need reg[HW_MAX_THREAD][NUM_REG].
  - Don't add a new pipeline stage but you have to have a scheduler in the decode stage.
  - The scheduler is just simply an array that holds one op from each thread.
  - The scheduler should implement the Round-robin policy among ready threads.

- *Memory*

  - Each thread has its own physical memory addresses. Virtual memory addresses will be translated correctly for each thread. Since you are building VIPT cache , you don't need to check thread Id. All memory addresses are physical memory address. However, you can still keep thread Id information inside the tag structure for other usages such as cache partitioing.

- *WB*

  - When instructions are retired, it has to be in-order within a thread. Across threads, the processor can retire instructions out of order.
  - In programming assignment #3, when an op is not finished, we stop the retirement. In this assignment, even though an op is not finished, if there are finished ops from other threads, the processor should retire them. We out of order retirement is possible across threads.

- *Stats*

  - Now, we need to collect stats per thread rather than all threads together.
  - We collect separate retired_instruction_thread, bp_miss_count_thread, bp_corr_predict_thread, dcache_miss_count_thread, dcache_hit_count_thread counters. The simulator should update retired_instruction, bp_miss_count, bp_corr_predict, dcache_miss_count, dcache_hit_count with all threads also.
  - For example, for branch misprediction counts,

```
if (bp_corr) {
   bp_corr_predict++;
```

```
   bp_corr_predict_thread[op->thread_id] = bp_corr_predict_thread[op->thread_id]+1;
}
```

- *Knobs related to this assignment*
  - KNOB_RUN_THREAD_NUM: number of threads that are running in this simulation (default 1, max 4)
  - KNOB_TRACE_NAME2: set the input trace file name2
  - KNOB_TRACE_NAME3: set the input trace file name3
  - KNOB_TRACE_NAME4: set the input trace file name4

- IPC behavior running multiple threads (20%)
- Branch predictor accuracy with MT (20%)
- Cache hit & miss behavior with MT (20%)
- TLB hit & miss behavior with MT (20%)
- IPC with memory-intensive and non-memory-intensive benchmarks (20%)
- IPC difference within 1.5% is acceptable for above cases
- We check branch predictor accuracy and cache hit/miss counters, TLB hit/miss counter values. The error ranges are the same as the previous assignments.

Submission Guide
Please follow the submission instructions. If you do not follow the submission file names, you will not receive the full credit. Please check the class homepage to see the latest update. Your code must run on the shuttle clusters with g++4.1.

Please do not turn in pzip files(trace files). Trace file sizes are huge so they will cause a lot of problems.
(Tar the lab4 directory. Gzip the tarfile and submit lab4.tar.gz file at T-square)
Please make it sure the directory name is lab4!
cd lab4
make clean
rm *.pzip
cd ..
tar cvf lab4.tar lab4
gzip lab4.tar

Part 2:
Simulations (20 points)
Include your simulation results in a report. You do not need to submit any traces. Please note that there are many simulation cases so it will take several hours to simulate all of them. 10M instructions will provide enough data so you can reduce the simulation time by simulating only 10M instructions.

The default configuration is
gshare history length: 12
MSHR size: 4
cache size, way: 512KB 4 way
Dcache latency: 5 cycles
DRAM row buffer hit latency: 100 cycles
DRAM row buffer miss latency: 200 cycles
MSHR size:4, DRAM_BANK_INDEX_SIZE: 2, DRAM_BANK_ROW_BITS: 20
Use 2 benchmarks: one is very memory intensive and the other is non-memory intensive. *2-way MT architecture*.

- For MT, create 2 benchmarks using 3 traces (trace1+trace2, trace1+trace3, trace2+trace3). Show weighted Speedup results.
  weighted speedup = Sigma(i,n) (ipc_smt(i)/ipc_alone(i))/n

- Classify three traces as memory intensive or not. Discuss the performance impacts.
- Discuss branch prediction accuracy effect in MT. Do you see degradation compared to a single thread running? why or why not?
- You could notice that one benchmark finishes much faster than others. Discuss how we should simulate benchmarks to reduce the simulation errors.

The following questions are provided to prepare Mid-term II.

- [0 point] Assume that there are two processors and each processor has a cache which is described in Problem 5 (see the below). The processors use the MSI coherence protocol. The coherence stats are denoted M, S, and I for Modified, Shared, and Invalid. A sequence of one or more CPU operations is specified P#: op, address, -> value. P# designates the CPU (e.g., P0), op is the CPU operation (LDB: load byte, STB: store byte). address denotes the memory address, and -> value indicates the new word to be assigned on a write operation.
  Each action occurs after the previous action is finished.
  (1) Show only the cache blocks that change, for example P0.B0: (I, 1, 0x01) indicates that CPU P0's block B0 (0 is the index number) has the final state of I, tag of 1, and data 0x1. Assume that the memory is initialized by all 0 at the beginning of the program.
  (2) Show the final state of the caches after all the following operations. (coherence state, cache tags, data, and index). To simplify the problem you do not need to draw the entire cache. You just need to fill out the relevant entries

  P0: LDB 0x450

  P0: STB 0x448 -> 0x4

  P1: STB 0x450 -> 0x5

  P1: LDB 0x450

  P1: STB 0x448 -> 0x6

P0: LDB 0x450

P0: LDB 0x052

P0: STB 0x452 -> 0x7

P0: STB 0x852 -> 0x8

P0: LDB 0x1850

P1: LDB 0x850

P1: LDB 0x450

SOLUTION:

Address 0x52 falls under block 0x5
Address 0x448 falls under block 0x44
Addresses 0x450 and 0x452 falls under block 0x45
Addresses 0x850 and 0x852 falls under block 0x85
Address 0x1850 falls under block 0x185

All addresses have tag: 0x0

For the given sequence of LDB, STB operations, the following MSI protocol operations occur:

P0: LDB 0x450

P0.B45: (S,0,0x0)

P0: STB 0x448 -> 0x4

P0.B44: (M,0,0x4)

P1: STB 0x450 -> 0x5

P0.B45: (I,0,0x0) | P1.B45: (M,0,0x5)

P1: LDB 0x450

P1.B45: (M,0,0x5)

P1: STB 0x448 -> 0x6

P1.B44: (M,0,0x6) | P0.B44: (I,0,0x4)

P0: LDB 0x450

P0.B45: (S,0,0x5) | P1.B45: (S,0,0x5)

P0: LDB 0x052

```
P0.B5: (S,0,0x0)

P0: STB 0x452 -> 0x7

P0.B45: (M,0,0x7) | P1.B45: (I,0,0x5)

P0: STB 0x852 -> 0x8

P0.B85: (M,0,0x8)

P0: LDB 0x1850

P0.B185: (S,0,0x0)

P1: LDB 0x850

P1.B85: (S,0,0x8) | P0.B85: (S,0,0x8)

P1: LDB 0x450

P1.B45: (S,0,0x7) | P0.B45: (S,0,0x7)



Final Cache Entries are:

Processor 0

        B5 :- State: Shared, Tag: 0x0, Data: 0x0, Index: 0x5
        B44:- State: Invalid, Tag: 0x0, Data: 0x4, Index: 0x44
        B45:- State: Shared, Tag: 0x0, Data:0x7, Index: 0x45
        B85:- State: Shared, Tag: 0x0, Data: 0x8, Index: 0x85
        B185:- State: Shared, Tag: 0x0, Data: 0x0, Index: 0x185

Processor 1

        B5 :- State: Invalid, Tag: 0x0, Data: 0x0, Index: 0x5
        B44:- State: Modified, Tag: 0x0, Data: 0x6, Index: 0x44
        B45:- State: Shared, Tag: 0x0, Data: 0x7, Index: 0x45
        B85:- State: Shared, Tag: 0x0, Data: 0x8, Index: 0x85
        B185:- State: Invalid, Tag: 0x0, Data: 0x0, Index: 0x185
```

(cache description problem-5) A computer has a 128KB write back cache and 2MB physical memory. Each cache block is 16B, the cache is 2-way set associative and uses the true LRU replacement policy. Assume 26-bit virtual address space and byte-addressable memory. It uses a 1KB page size. The cache uses a physical tag and physical index. How big (in bits) is the tag store? The cache uses MSI protocol and assume that we need 2 bits to indicate cache coherence states.

Physical tag and physical index:
2MB: 21-bit address space
128KB, 2way, 16B block: (2^17/(2^4*2)) --> 12-bit index

B-offset bits: 4 bits
Address tag space = 21 - 12 - 4 = 5 bits
tag bits: address (5) + 1 (dirty) + 1 (valid) + 0.5 (lru) + 2 bits (coherence) = 9.5 bits.
total tag storage: (9.5 bits)* (2^17/16) = 77824 bits

- [0pt] p% of a program is vectorizable. We wish to run it on a multiprocessor. Assume we have an unlimited number of processing elements. If the maximum speedup achievable on this program is 30, what is p?
  s = 1/ ((1-p) + p/N)
  s = 30
  p = limit (N->infinite) (-10/30 * 1/(1/N-1)) = 0.97
  p = 0.97 or 97%