6.175: Constructive Computer Architecture

Tutorial 8

Project Part 2: Coherence

Quan Nguyen
(Will try to stay coherent)

Debugging Techniques

- Deficiency about `$display()`
  - Everything shows up together
- Distinct log file for each module: write to file
  - Also see `src/unit_test/sc-test/Tb.bsv`

```
Ehr#(2, File) file <- mkEhr(InvalidFile);
Reg#(Bool) opened <- mkReg(False);

rule doOpenFile(!opened);
  let f <- $fopen("a.txt", "w");
  if (f == InvalidFile) $finish;
  file[0] <= f; opened <= True;
endrule

rule doPrint;
  $fwrite(file[1], "Hello world\n");
endrule
```

Writing to InvalidFile will cause segfault.

Use EHR if the logic will call `$fwrite()` in the first cycle

Debugging Techniques

- Deficiency about cycle counter
  - Rule for printing cycle may be scheduled before/after the rule we are interested in
  - Don't want to create a counter in each module
- Use simulation time
  - `$display("%t: evict cache line", $time);`
  - `$time()` returns Bit#(64) representing time
  - In SceMi simulation, `$time()` outputs: 10, 30, ...

Debugging Techniques

- Add sanity checks
- Example 1
  - Parent is handling upgrade request
  - No other child has incompatible state
  - Parent decides to send upgrade response
  - Check: parent is not waiting for any child (waitc)
- Example 2
  - D$ receives upgrade response from memory
  - Check: must be in WaitFillResp state
  - Process the upgrade response
  - Check: if in I state, then data in response must be valid, otherwise data must be invalid (data field is Maybe type in the lab)

Coherence Protocol: Differences From Lecture

- In lecture: address type for **byte** address
  - Implementation: only uses **cache line** address
  - `(addr >> 6)` for 64-byte cache line
- In lecture: parent reads data in zero cycles
  - Implementation: read from memory, long latency
- In lecture: voluntary downgrade rule
  - No need in implementation
- In lecture: Parent directory tracks states for all address
  - 32-bit address space → huge directory
  - Implementation: usually parent is an L2 cache, so only track address in L2 cache
  - But we don't have an L2 cache

Coherence Protocol: Differences From Lecture

- Workaround for large directory
  - For each child, only tracks addresses in its L1 D$

```
Vector#(CoreNum, Vector#(CacheRows, Reg#(CacheTag)))
    tags <- replicateM(replicateM(mkRegU));
Vector#(CoreNum, Vector#(CacheRows, Reg#(MSI))
    states <- replicateM(replicateM(mkReg(I)));
```

  - To get MSI state for address `a` in core `i`

```
MSI s = tags[i][getIndex(a)] == getTag(a) ? states[i][getIndex(a)] : I;
```

Load-Reserve (lr.w) and Store-Conditional (sc.w)

- New state in D$
  - `Reg#(Maybe#(CacheLineAddr)) la <- mkReg(Invalid);`
  - Cache line address reserved by `lr.w`
- Load-reserved: `lr.w rd, 0(rs1)`
  - `rd <= mem[rs1]`
  - Make reservation: `la <= Valid(getLineAddr(rs1));`
- Store-conditional: `sc.w rd, rs2, 0(rs1)`
  - Check `la`: `la` invalid or addresses don't match: `rd <= 1`
  - Otherwise: get exclusive permission (upgrade to M)
    - Check `la` again
    - If address match: `mem[rs1] <= rs2; rd <= 0`
    - Otherwise: `rd <= 1`
    - If cache hit, no need to check again (address already match)
  - Always clear reservation: `la <= Invalid`

Load-Reserve (lr.w) and Store-Conditional (sc.w)

- Cache line eviction
  - Due to replacement, invalidation request …
  - May lose track of reserved cache line
    - Then clear reservation
  - Compare evicted cache line with `la`
    - If match: `la <= invalid`
  - This is how an LR/SC pair ensures atomicity

Reference Memory Model

- Debug interface returned by reference model is passed into every D$

```
interface RefDMem;
  method Action issue(MemReq req);
  method Action commit(MemReq req,
      Maybe#(CacheLine) line, Maybe#(MemResp) resp);
endinterface

module mkDCache#(CoreID id)(
    MessageGet fromMem, MessagePut toMem,
    RefDMem refDMem, DCache ifc);
```

  - D$ calls the into debug interface `refDMem`
  - Reference model will for coherence violations
- Reference model: `src/ref`

Reference Memory Model

- `issue(MemReq req)`
  - Called when `req` issued to D$
  - in `req()` method of D$
  - Give program order to reference model
- `commit(MemReq req, Maybe#(CacheLine) line, Maybe#(MemResp) resp);`
  - Called when `req()` finishes processing (commit)
  - `line`: cache line accessed by `req`, set to Invalid if unknown
  - `resp`: response to the core, set to Invalid if no repsonse
- When commit() is called, reference model checks whether:
  - `req` can be committed
  - `line` value is correct (not checked if Invalid)
  - `resp` is correct

Adding Store Queue

- New behavior for memory requests
  - Ld: can start processing when store queue is not empty
  - St: enqueue to store queue
  - Lr, Sc: wait for store queue to be empty
  - Fence: wait for all previous requests to commit (i.e. store queue must be empty)
    - Ordering memory accesses
- Issuing stores from store queue to process
  - Only stall when there is a Ld request

Multicore Programs

- Run programs on 2-core system
- Single-thread programs
  - Found in `programs/assembly`, `programs/benchmarks`
  - core 1 starts looping forever at the very beginning
- Multithread programs
  - Find them in `programs/mc_bench`
  - startup code (`crt.S`): allocate 128KB local stack for each core
  - `main()` function: fork based on core id

```
int main() {
  int coreid = getCoreId();
  if (coreid == 0) { return core0(); }
  else { return core1(); }
}
```

Multicore Programs: `mc_print`

- Easiest one
- Two cores print “0” and “1” respectively
- Sample output:
  - (no cycle/inst count printed)

```
---- ../../programs/build/mc_bench/vmh/mc_print.riscv.vmh ----
01
PASSED
```

Multicore Programs: mc_hello

- Core 0 passes each character of a string to core 1
- Core 1 prints each character it receives
- Sample output:
  - (no cycle/inst count printed)

```
---- ../../programs/build/mc_bench/vmh/mc_hello.riscv.vmh ----
Hello World!
This message has been written to a software FIFO by core 0 and read and printed by
core 1.
PASSED
```

Multicore Programs: mc_produce_consume

- Larger version of mc_hello
- Core 1 passes each element of an array to core 0
- Core 0 checks the data
- Sample output:

```
---- ../../programs/build/mc_bench/vmh/mc_produce_consume.riscv.vmh ----
Benchmark mc_produce_consume
Cycles (core 0) = xxx
Insts  (core 0) = xxx
Cycles (core 1) = xxx
Insts  (core 1) = xxx
Cycles  (total) = xxx
Insts   (total) = xxx
Return 0
PASSED
```

Instruction counts may vary due to variation in busy waiting time, so IPC is not a good performance metric. Execute time is a better metric.

Multicore Programs: mc_{median,vvadd,multiply}

- Data parallel: fork-join style
- Core 0 calculates first half results
- Core 1 calculates second half results
- Sample output:

```
---- ../../programs/build/mc_bench/vmh/mc_median.riscv.vmh ----
Benchmark mc_median
Cycles (core 0) = xxx
Insts  (core 0) = xxx
Cycles (core 1) = xxx
Insts  (core 1) = xxx
Cycles  (total) = xxx
Insts   (total) = xxx
Return 0
PASSED
```

Multicore Programs: mc_dekker

- Two cores contend for a mutex (Dekker's algorithm)
- After getting into critical section
  - increment/decrement shared counter, print core ID
- Sample output:

```
---- ../../programs/build/mc_bench/vmh/mc_dekker.riscv.vmh ----
Benchm1ark mc_1dekker1
100110...000
Core 0 decrements counter by 600
Core 1 increments counter by 900
Final counter value = 300
Cycles (core 0) = xxx
Insts  (core 0) = xxx
Cycles (core 1) = xxx
Insts  (core 1) = xxx
Cycles  (total) = xxx
Insts   (total) = xxx
Return 0
PASSED
```

For implementation with store queue, a fence is inserted in mc_dekker.

Multicore Programs: mc_spin_lock

- Similar to mc_dekker, but use spin lock implemented by lr.w/sc.w
- Sample output:

```
---- ../../programs/build/mc_bench/vmh/mc_spin_lock.riscv.vmh ----
Bench1mark mc1_spin_l1ock
10101...000
Core 0 increments counter by 300
Core 1 increments counter by 600
Final counter value = 900
Cycles (core 0) = xxx
Insts  (core 0) = xxx
Cycles (core 1) = xxx
Insts  (core 1) = xxx
Cycles  (total) = xxx
Insts   (total) = xxx
Return 0
PASSED
```

Multicore Programs: mc_incrementers

- Similar to mc_dekker, but use atomic fetch-and-add implemented by lr.w/sc.w
- Core ID is not printed
- Sample output:

```
---- ../../programs/build/mc_bench/vmh/mc_incrementers.riscv.vmh ----
Benchmark mc_incrementers

core0 had 1000 successes out of xxx tries
core1 had 1000 successes out of xxx tries
shared_count = 2000
Cycles (core 0) = xxx
Insts  (core 0) = xxx
Cycles (core 1) = xxx
Insts  (core 1) = xxx
Cycles  (total) = xxx
Insts   (total) = xxx
Return 0
PASSED
```

Some Reminders

- Use CF regfile and scoreboard
  - Compiler creates a conflict in Sizhuo's implementation with bypass regfile and pipelined scoreboard
- Sign up for project meeting
- Project deadline: 3:00pm Dec 14
- Final presentation (10min)