Gem5 Learning Notes

Name: Haoxuan Zhang

Research Project Fall 2020
Learning Notes

0.1 O3 CPU Attributes

a) From Base CPU:

protected:

1. instCnt
2. _cpuId
3. socketId
4. taskID
5. _pid
6. bool _switchedOut
7. _cacheLineSize
8. `std::vector<BaseInterrupts*>interrupts;`
9. `std::vector<ThreadContext *>threadContexts;`
10. `Trace::InstTracer * tracer;`
11. `Cycles previousCycle;`
12. `CPUState previousState;`

public:

1. `ThreadID numThreads;`
2. `System *system;`

b) From FullO3CPU:

protected:

1. EventFunctionWrapper tickEvent; The tick event used for scheduling CPU ticks. What is a CPU tick?
2. EventFunctionWrapper threadExitEvent;The exit event used for terminating all ready-to-exit threads
3. **typename CPUPolicy::Fetch fetch;** This is a class which needs Impl specified
4. **typename CPUPolicy::Decode decode;** This is a class which needs Impl specified
5. **typename CPUPolicy::IEW iew;** The issue/execute/writeback stages.
6. **typename CPUPolicy::Commit commit;** The commit stage.

7. **PhysRegFile regFile;** The physical Register File
8. **typename CPUPolicy::FreeList freeList;** What is this?
9. **typename CPUPolicy::RenameMap renameMap[Impl::MaxThreads];** The rename map
10. **typename CPUPolicy::RenameMap commitRenameMap[Impl::MaxThreads];** The commit rename map
11. **typename CPUPolicy::ROB rob;** The re-order buffer.
12. **std::list<ThreadID>activeThreads;** Active Threads List
13. **std::unordered_map<ThreadID, bool>exitingThreads;** This is a list of threads that are trying to exit. Each thread id is mapped to a boolean value denoting whether the thread is ready to exit.
14. **Scoreboard scoreboard;** The scoreboard
15. **TimeBuffer<TimeStruct>timeBuffer;** The main time buffer to do backwards communication.
16. **TimeBuffer<FetchStruct>fetchQueue;** The fetch stage's instruction queue.
17. **TimeBuffer<DecodeStruct>decodeQueue;** The decode stage's instruction queue.
18. **TimeBuffer<RenameStruct>renameQueue;** The rename stage's instruction queue.
19. **TimeBuffer<IEWStruct>iewQueue;** The IEW stage's instruction queue.
20. **ActivityRecorder activityRec;** The activity recorder; used to tell if the CPU has any activity remaining or if it can go to idle and deschedule itself.

private:

1. **System *system;** Pointer to the system.
2. **std::map<ThreadID,unsigned>threadMap;** Mapping for system thread id to cpu id
3. **std::vector<ThreadID>tids;** Available thread ids in the cpu
4. lots of Stats at the end of the class declaration

public:

1. ```
   enum Status {
         Running,
         Idle,
         Halted,
         Blocked,
         SwitchedOut
   };
   ```
2. `BaseTLB *itb;`
3. `BaseTLB *dtb;`
4. `Status _status;`
   ```

5. `int instcount;` with flag `NDEBUG` Count of total number of dynamic instructions in flight.
6. `std::list<DynInstPtr> instList;` List of all the instructions in flight.
7. `std::queue<ListIt> removeList;` List of all the instructions that will be removed at the end of this cycle.
8. `bool removeInstsThisCycle;` Records if instructions need to be removed this cycle due to being retired or squashed.

0.2 Sept 19 2020

a) What is LSQ request in Full O3 CPU?

b) if there are multiple lanes dispatching at the same time. how should they register themselves in the reorder buffer?

c) DefaultIEW is initializing width in the way: issueWidth(params->issueWidth) Maybe changing such parameters could change the configuration. next problem is how to change the bandwidth of the memory

d) It seems that the SimObject is polymorphic so I could convert a point to SimObject to a pointer to the derived class.

e) seems that all the parameters in the python classes are declared not in __init__() but in the class itself.

f) what is a probe?

g) every sim_object has a name and the plan is to split the name and check whether the name "cpu" or something is in the list of strings. If so, the object is found.

0.3 Dec 28 2020

a) `SimpleCPUPolicy<O3CPUImpl>`:
include fetch, decode, rename, iew, commit stages

   (a) `typedef DefaultFetch<Impl> Fetch;` :

   has pointer to CPUPol, DynInst, DynInstPtr and O3CPU

   from CPUPol, it has type of FetchStruct(The struct for communication between fetch and decode) and TimeStruct(The struct for all backwards communication)

   It has IcachePort as a MasterPort(base class)

   It has FetchTranslation as BaseTLC::Translation

It has a private FinishTranslationEvent as Event(base class). When processed, it assert that numInst is smaller than fetchWidth and then call finishTranslation(fault, req)

It has FetchStatus as Active or Inactive;

It has ThreadStatus

It has list of threads organized by priority

(b) `typedef DefaultDecode<Impl> Decode;`:

similar pointers It has pointer to FetchStruct and DecodeStruct and TimeStruct
It has DecodeStatus as Active or Inactive
It has ThreadStatus
It has

(c) `typedef DefaultRename<Impl> Rename;`

(d) `typedef DefaultIEW<Impl> IEW;`

(e) `typedef DefaultCommit<Impl> Commit;`

0.4 Jan 15 2021
Things to do:

a) read the selected loop files; detect them in fetch and drain and dump statistics; think about what to test to make sure the program runs correctly.

b) after detecting the drain, divide memory latency by 2 and compare the cycle count

c) after detecting the drain, modify the cpu width and compare the cycle count

d) change both. compare the cycle count and other statistics

Problems encountered:

a) If detect drain in fetch stage; How to deal with mis-branch prediction? Could I assume that without interrupt or exception, squash can only happen from branch prediction? what about memory speculation? Until when can I make sure a instruction is safe to commit? iew stage or commit?

The key is to know until which stage, we can say a inst is safe to commit. At that stage, if a inst's pc is the start of the loop, then we drain and switch.

Fetch Stage tick processing function:

a) checkSignalsAndUpdate() for all tid

b) DPRINGF(fetch, "Running stage.\n");

c) If **fullSystem**, Do sth

d) for all the threads, do fetch(status_change)

e) Record number of instructions fetched this cycle for distribution:
fetchNisnDist.sample(numInst);

f) if status_change then assign updateFetchStatus() to _status

g) Issue the next I-cache request if possible.

h) Send instructions enqueued into the fetch queue to decode. Limit rate by fetchWidth. Stall if decode is stalled.

i) Pick a random thread to start trying to grab instructions from.

dont́ care randomly picking thread because currently there is only one thread.

In this step, while there is available instruction and decode width not exceeded, if decode not stalled and fetch queue not empty, add inst to decode and set wrtoeToTimeBuffer to true

j) If wroteToTimeBuffer which means there was activity in the cycle, inform the CPU of it by calling cpu-¿activityThisCycle();

fetch():

This is the actual fetch:

a) thisPC = pc[tid]

b) fetchAddr = (thisPC.instAddr() + pcOffset) & BaseCPU::PCMask; //pcOffset is always multiple of 8.

c) if IcachAccessComplete, then set status to running and status_change = true

else if it is Running, then align fetch PC to the start of a fetch buffer segment
if buffer no longer valid or fetch addr moved to next cache block, fetchCacheLine() and return

else if interrupt, stall cpu and return

else idle cycle count ++ and return

d) increament fetch cycles; nextPC = thisPC; start to track predicted Branch which indicates whether a predicted branch ended this fetch block.

e) Loop through instruction memory from the cache. Keep issuing while fetchWidth is available and branch is not predicted taken. (This is a huge while loop); next PC is looked up in lookupAndUpdateNextPC(const DynInstPtr &, TheISA::PCState &)

f) pc[tid] = thisPC

0.5 Decode Stage tick processing function

a) for all threads, call decode

b) update status if status_change

decode():

a) call decodeInsts() only when status is Idle or Running or Unblocking

b) In decodeInsts(), the queue will be skidbuffer(buffer between fetch and decode) if unblocking or insts(queue of all instructions coming from fetch this cycle) otherwise

c) while insts_available >0; if inst->isSquashed() then continue;

d) setCanIssue() if inst don't require source register

e) add the instruction into the decode queue. The next instruction may not be valid, so check to see if branches were predicted correctly

f) check is branch if predicted as a branch. panic otherwise.

g) compute any PC-relative branches; if branchtarget is not the same as predicted target, the squash() is called, it clears the buffers from fetch and call cpu->removeInstsUntil(squash_seq_num, tid) which will squash all the inst in instList until this instruction; Seems that the branch instruction itself is not squashed. The predicted PC: nextPC is calculated in lookupAndUpdateNextPC() in fetch. In lookupAndUpdateNextPC(), if it is not control instruction, then just advance PC and return. Otherwise, do the prediction. It will predict both taken or not taken and the target address.

h) if insts to decode not empty then block and put all insts into skid buffer

i) QUestion: when resolving branches in decode, not considering conditional branches that are predicted as not taken?

Answer: it is only confirming the target, not the prediction correctness. Maybe prediction correctness will be checked in the next stage

Question: when are the reorder buffer tags in result shift registers are determined? In schedule stage with out-of-order dispatching?

What does dependency means? It means the second instruction cannot start before the first completes in the dependency pair.

WAR: the read will read the wrong value by the write

WAW: the system will see the wrong value as the first result instead of the second result.

0.6 Rename Stage tick processing function

New mapping of architectural and physical registers pair is made whenever an instruction write to a register? I don't think so

Question: In renaming, when can physical register be marked as available? when there is no RAW dependency on the architectural registers?

The renaming process is to translate the architectural registers into physical registers?
It do has a table mapping architectural registers into physical registers.

When an instruction can be dispatched? how can a hardware detect depencies and dispatch whenever the dependency has been eliminated? If the latest dependency has finished.

The idea of register renaming is to have more registers in physical space but not in architectural space.

Store the pointers in instruction queue and reorder buffer.

Scoreboarding is used to show data dependencies of every instruction logged to determine whether an instruction is ready to be released when no conflicts with previously issued and incomplete instructions. ALl the instructions decoded are having four stages: Issue(stalled if there is WAW dependency or structural hazard of the requested functional unit), Read operands(stall if RAW dependency), execution(seems that this stage won't be stalled and after execution is finished, the scoreboard will be notified.) and write result(in reorder buffer, stall on WAR dependency)

tick():

a) for each thread, check whether there are squash, stall signals and the status

b) call rename(status_change, tid), then if the status is running, idle or unblocking, renameInsts(tid) will be called; Inside renameInsts(): number of free rob entries calculated, number of free instruction queue entries calculated. min_free_entries is the minimum among the two. The stall source is marked as IQ or ROB

c) handle serializing(what is serializing? serializeafter make the next instruction as serializebefore which can only wait in rename until the ROB is empty. IPR is serialize before and store conditionals are serialize after.)

d) while resource and instruction available. LQ and SQ free entries are separately calculated. stall if not available. call renameMap[tid]-¿can Rename() to see whether enough registers are available. stall if not.

e) rename Src regs and rename Dest regs for that register. For rename src register, mark src register ready if scoreboard->getReg(renamed_register) is not Null.

question: if it is not register when rename, when will it be marked as ready after the result has been generated? I guess it is in the commit stage or in the execution stage. when a dest

register is calculated, it will inform the scoreboard. (will the scoreboard keep track of all the instruction's status and which instruction they are waiting for?)

Only rename and iew stage has scoreboard. Does that mean before the rename stage, the order of the instructions are following the program order?

f) when renaming dest register, a new entry is added into history buffer(**A per-thread list of all destination register renames, used to either undo rename mappings or free old physical registers.**). rename_result.first is the new physical register and the old physical register is rename_result.second. (How to determine whether a physical register is free? why bother keeping the old physical register?)

g) Put instruction in rename queue.

0.7 IEW Stage tick processing function

tick(): IssueWidth, dispatch width and wbwidth are all in IEW stage; decode width is in decode stage and fetch stage; fetch width is in fetch stage; skid buffer max is affected by renamewidth; rename stage has commitwidth and rename width

a) load/store queue tick()

b) sortInsts(); Sorts instructions coming from rename into lists separated by thread

c) Free function units marked as being freed this cycle(before it actually do anything? means this free is not blocked one which will only be effective in the next cycle?)

d) for each thread, checkSignalsAndUpdate(tid) and then dispatch(tid);

In dispatch(), Dispatch should try to dispatch as many instructions as its bandwidth will allow, as long as it is not currently blocked. **Question: which parameter determine how many functional units are there?** calldispatchInsts(tid) and if unblocking, insert skid buffer.

When dispatching instructions, call dispatchInsts(tid) and instructions will be obtained from skid buffer if unblocking or the insts buffer from rename otherwise.

if the instruction is squashed, then it will pop that instruction out from the insts to dispatch and increament dispatched flag in toRename and continue to next inst(**Question: why toRename? for ScoreBoarding? why not refer to Scoreboard object directly?**)

Otherwise, if inst queue is full, block; tell rename iweUnblock is false

if corresponding lsq is full and it is a ld/ST or Atomic inst then block; tell rename iweUnblock is false.

Otherwise, just issue the instruction(**Question: wait, did the comment just used issue? what is the difference between dispatch and issue here?**)

insert the inst to ldstqueue.insertNonSpec(inst) if inst is atomic and add_to_iq is false and increment toRename dispatchedtosq.

insert the inst to ldstqueue.insertLoad(inst) if the instruction is load, add_to_iq is true and increment toRenmae iewInfo dispatchedtoLQ

insert the isnt to ldstQueue.insertStore(inst) if inst is store, same as load if it is not store conditional.

insert instQueue.insertBarrier(inst) if inst is Mem Barrier; set inst to can commit if it is memBarrier; set add_to_iq as false(**Question: Why?**)

instQueue.recordProducer(inst) and set inst to issued, executed and setCanCommit if inst is a nop; add_to_iq is false.

Else, assert inst is not executed and set add_to_iq to true.

for all the previous conditions, if add_to_iq is true and inst is not speculative. Then set instruction to can commit. instQueue.insertNonSpec(inst)(meaning it is not speculative); Set add_to_iq to false.

If add_to_iq is still true at this point, instQueue.insert(inst)

pop this inst from insts_to_dispatch and increment dispatched and ppDispatch->notify(inst) (**What does this mean?**)

e) **After dispatch():** if not suqashing, do **executeInsts():**.

get number of insts to execute from fromIssue->size. the fromIssue is a Timebuffer::wire which seems to be registered in instQueue.

get the oldest scheduled instruction and removes it from the list of instructions waiting to execute by calling instQueue.getInstToExecute()

notify potential listeners that this isntruction has started executing by calling ppExecute->notify(inst)

Check if the instruction is squashed; if so then skip it

Execute instruction. Note that if the instruction faults, it will be handled at the commit stage.

if memref, inst executed in special way. Otherwise, if inst->getFault() is NoFault, execute and if not readPredicate, forwardOldRegs. set the inst to executed and send the inst to commit

Update ExeInstStats(inst)

Check if branch prediction was correct, if not then we need to tell commit to squash in flight instructions.

if nothing making the current branch to be ineffect(line 1361 iew_impl.hh), if inst->mispredicted() and not loadNotExecuted, set fetchRedirect[tid] to be true and squashDueToBranch(inst, tid); notify mispredict. The ROB will be signaled to squash the instructions after this one

f) a conditional change to exeStatus, updateLSQNextCycle and broadcast_free_entries.

g) ldstQueue.writebackStores(); Writeback any stores using any leftover bandwidth.

h) for each thread, Update structures(the ldstQueue and the instQueue) based on instructions committed.

i) if from commit told to commit non speculative instructions, schedule the non speculative inst.

j) broadcast free entries

0.8 Commit Stage tick processing function tick():

a) Check if any of the threads are done squashing. Change the status if they are done.

b) call commit():

1. check interrupt if fullsystem

2. do the squashing staffs if needed. Otherwise get the inst by calling getInsts(), seems that the commit and squash not happening in the same cycle is guaranteed here:

Read any renamed instructions and place them into the ROB.

get inst from rename and insert it into ROB and set youngestSeqNum[tid] = inst->seqNum; 3. call commitInsts() to try to commit any instructions:

Commit as many instructions as possible until the commit bandwidth limit is reached, or it becomes impossible to commit any more.

maybe call the cpu->drain() and squashAfter and reset the fetch pc; However, maybe don't squash it

c) markCompletedInsts():

d) for each thread: set rob status

0.9 CPU Switching Plan

helper functions to be created:

a) CPU::RunningLoop(inst) // return true if the inst pc is in a loop node in PIM_list

b) CPU::RunningLoopinMainCPU(inst) //return true if currently running on main cpu and the inst->pc is within a loop node in PIM_list.

c) CPU::switch_to_PIM() // set the PIM flag to be true and set the PIM flag in system also to be true

d) CPU::switch_back_from_PIM()

e) CPU:: shrink_width() // shrink the dispatch width. ? which parameter determines the number of function unit available?

helper class to create

a) PIM_FU_Pool : public FUPool // a functional unit pool which can switch betwenn pool with lots of functional units and pool with fewer functional units.

b) PIM_DDR_Controller: public DDR_Controller // a DDR controller that can switch between high latency and low latency

Scenario

in commitInsts(), if RunningLoopinMainCPU(inst) returns true, then drain.

In CPU::tick(), if trydrain returns true and flag drain from PIM loop is true, then check whether the next pc is still in loop. If so, call the switch function and set the flag drain from PIM loop to be false. Otherwise set the flag to false but not switch.

Or Should I make sure that all the instructions in the loop should be strictly run in PIM cores? In this way the stat is easier but It won't be too difficult for the former strategy.

In commitInsts, if in PIM and not in loop then drain.

In CPU::tick() if trydrain returns true and flag drain from restore main cpu is true, then check whether next pc is still out of loop, If so call the switch back to main cpu function.

The Squash Process From IEW

toCommit->squashedSeqNum[tid] = inst->seqNum; The seqNum to which insts should be squashed. This seqNum should be the branch instruction seqNum

TheISA::PCState pc = inst->pcState(); TheISA::advancePC(pc, inst->staticInst);

toCommit->pc[tid] = pc; This pc should be the pc of the first instruction in the correct branch.

The squash due to switching should be the same thing. Just tell the commit to squash until the inst whose next PC will be in the loop and set notify the fetch to reset pc and at the same time drain. After trydrain indicating that the drain has finished, switch the cpu and contiue.

If the inst::nextPC is out of loop in commit, squash until the inst and notify the fetch to reset pc and at the same time drain. After trydrain indicating that the drain has finished, switch the cpu and continue.

Assumption: all the actions are performed in tick() for all simobject. This assumption maybe used when making changes to the CPU and memory controller.

Seems that the profiling and stats at present are all micro insts. That explain why the stats has more counts than profiling. Multiple consecutive microop can be having the same pc. and the first jump will create the difference between profiling and stats.

I think I need to use microop. This should be correct. Because only the microops can separate arithmetic operation from memory reference operation.

Question: Why some instructions can be both (integer —— float) and (load —— store) such as mov (%rbx),%r14?

1. How to modify the memory?