Gem5 Learning Notes

Name: Haoxuan Zhang

Research Project Fall 2020
Learning Notes

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.

1.1 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.

1.2 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;`

1.3 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.

donť 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

1.4 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.

1.5 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.

1.6 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

1.7 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

1.8 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?

2. How to modify the memory?

cpu drain, set PIM_mode to true in system.

memory see PIM_mode and set MEM_PIM_Mode_received

cpu in tick()(not sure whether still have tick now), see MEM_PIM_Mode_received and resume. However, memory is not guranteed to have events after cpu drain() starts.

Idea2: have a new port to connect cpu and memory. cpu send a packet to schedule an event to make memory to change parameters and use trydrain() to deschedule() the tick events.

after memory has finished changing the parameters, send a packet back to cpu to resume draining.

The packet data can be used to tell whether the memory should switch to or from PIM_mode

There should be an EventWrapper in memory controller to be scheduled to change the parameters and send a packet back to cpu so that cpu can resume from drain().

3. Problems encountered

A loop offloaded to PIM leads to repeated switch from PIM to non PIM. Here is the assambly of the loop:

```
...
4010df: eb 17                   jmp    4010f8 <__libc_start_main+0x1f8>
4010e1: 83 7b 08 25             cmpl   $0x25,0x8(%rbx)
4010e5: 4c 8b 33                mov    (%rbx),%r14
4010e8: 0f 85 12 02 00 00       jne    401300 <__libc_start_main+0x400>
4010ee: ff 53 10                callq  *0x10(%rbx)
4010f1: 48 83 c3 18             add    $0x18,%rbx
4010f5: 49 89 06                mov    %rax,(%r14)
4010f8: 48 81 fb 00 04 40 00    cmp    $0x400400,%rbx
4010ff: 72 e0                   jb     4010e1 <__libc_start_main+0x1e1>
...

0000000000423440 <strchr_ifunc>:
423440: 8b 15 e6 89 29 00       mov    0x2989e6(%rip),%edx        # 6bbe2c <_dl_x86_cpu_featu
423446: 48 8d 05 33 c5 01 00    lea    0x1c533(%rip),%rax        # 43f980 <__strchr_avx2>
42344d: 89 d1                   mov    %edx,%ecx
42344f: 81 e1 00 0c 02 00       and    $0x20c00,%ecx
423455: 81 f9 00 0c 00 00       cmp    $0xc00,%ecx
42345b: 74 15                   je     423472 <strchr_ifunc+0x32>
42345d: 83 e2 04                and    $0x4,%edx
423460: 48 8d 05 e9 c0 01 00    lea    0x1c0e9(%rip),%rax        # 43f550 <__strchr_sse2>
423467: 48 8d 15 c2 0a 02 00    lea    0x20ac2(%rip),%rdx        # 443f30 <__strchr_sse2_no_b
42346e: 48 0f 45 c2             cmovne %rdx,%rax
423472: f3 c3                   repz retq
```

```
423474: 66 2e 0f 1f 84 00 00  nopw   %cs:0x0(%rax,%rax,1)
42347b: 00 00 00
42347e: 66 90                 xchg   %ax,%ax
...

0000000000484bb0 <strspn_ifunc>:
484bb0: f6 05 3b 72 23 00 10  testb  $0x10,0x23723b(%rip)        # 6bbdf2 <_dl_x86_cpu_feat
484bb7: 48 8d 15 02 01 00 00  lea    0x102(%rip),%rdx        # 484cc0 <__strspn_sse42>
484bbe: 48 8d 05 5b 00 00 00  lea    0x5b(%rip),%rax        # 484c20 <__strspn_sse2>
484bc5: 48 0f 45 c2           cmovne %rdx,%rax
484bc9: c3                    retq
484bca: 66 0f 1f 44 00 00     nopw   0x0(%rax,%rax,1)
```

Question: Switching Policy? switch only at the boarder of the loop? Or switch whenever the next pc's projected execution location doesn't match with the cpu mode?

Draining and Squashing Process After draining, the cpu drain() call commit drain() which set drain pending to be true. then if drain pending is true, commit stage call squashAfter() which set commitStatus to be SquashAfterPending. Then in the next commit() get called, if SquashAfterPending is true, squashFromSquashAfter(tid) will be called. Then in squashFromSquashAfter(tid), squashAll() will be called which will inform the fetch about the pc state to resume. The pc is updated whenever before calling bool commit_success = commitHead(head_inst, num_committed); in commitInsts().

Therefore, a squash is not needed after the drain because the drain itself has done the squashing.

Jan 25: next step: make sure that the switching sequencing is correct. Without drain, with drain, and why with some special loop(containing call), the decode will encounter assertion failure of decode.stall[] when unblocking? why? Why there is a very long delay after finished send signal to memory print?

Jan26: Result:

with draining(without actually changing anything):
24082408000 ticks.

without draining:
24082702000 ticks.
with draining (4 loops offloaded)
24082262000 ticks.
with draining (10000 tick delay in DRAM_CTRL):
24081660000 ticks.

with draining (1000000 tick delay in DRAM_CTRL):
24091549000 ticks.

with draining (1000000 tick delay in DRAM_CTRL) and shrinking the mem_ctrl delay:
16385135000 ticks.

why this is the case? why draining has fewer cycles?

Problem encountered: from decode to fetch, decodeunblock[tid] is true across the drain and drain resume. That is caused by blocking in decode in second last tick and decode unblocking in last tick. The fetch will make the stall to be true when it detects the deocode block or remain the stall if decode block remains in true. The stall should be cleared when the fetch in the next tick see the deocode unblock. However, in drain resume, stall is cleared but decode unblocking is not.

My current solution is to only clear stall if decodeUnblock is not true.

Why they have block and unblock? maybe they need to determine whether get inst in inst queue or skid queue?

3.1 Create My Own Draining

a) detect the loop in fetch and stall fetch, drain and then switch

b) detect the loop in commit and squash all not committed or reset all stages

CPU pc is the commit pc

Switching Strategy

when commit detect the next pc is a switch, reset all stages, store the next pc somewhere in cpu, reset rob. send package to mem, deschedule ticks. After receiving package from mem, set fetch pc to be the previously stored pc and reschedule the cpu tick event.

Helper Functions to Create:

a) cpu::ResetAllStages()

b) cpu::PIMRestorePC

c) cpu::PIMRestoreFlag: Used to indicate that the cpu is restoring from PIM and the fetch should start to fetch the nextPC.

d) cpu::switchToPIM()

e) cpu::switchFromPIM()

Experiment on d0eb0255e0740a1fad93c062117d27004dadbce0 node:

a) 4 loops offloaded(8 drains) and memory divided: 16385135000

Experiment on 3852adc62ba801c7da4ec6a1616b04fc97a2d59f node(10 tick delay in DRAM_CTRL):

a) 4 loops offloaded(8 drains) and memory not divided:

9295000 start to drain
9338000 drain done
9338010 drain resume start(tick event scheduled in nextCycle)
9340000 new tick event started
24082262000 total ticks

b) only first 2 loops offloaded(4 drains) and memory not divided:

9295000 start to drain
43000 ticks -¿ 43 cycles 9338000 drain done
9338010 drain resume
9340000 new tick event started
9398000 to get to the next next PC
8389000 ticks -¿ 8389 cycles
17729000 second drain detected start to drain (ROB has 29 insts)
17734000 ROB has 0 insts.
8000 ticks -¿ 8 cycles
17737000 Drain done
17739000 new tick event continues and cpu pc is the nextPC
30689000 ticks -¿ 30589 cycles
48428000 third drain detected start to drain
285 cycles
48713000 Drain done
48715000 new tick continues
2233 cycles
50948000 forth drain detected start to drain
684 cycles
51632000 Drain done
51634000 new tick continues
24029922 cycles
24081556000 total ticks (955 branch mispredicted, 14268 squashed insts, 2137730 committed insts, 135 int alu not available)
24080660000 (2 int alus) 24082272000 (1 int alus, 968 int alu not available) ? The number of ticks didn't changed much after changing the fupool.

c) only 1 loop offloaded(2 drains) and memory not divided:
24082272000 ticks (loop 39, second loop offloaded)
(952 branch mispredicted, 14250 squashed insts, 2137730 committed insts)
24081556000 ticks (loop 15, first loop offloaded)

d) not actually drain! first 2 loops(4 drains)
9295000 recognized the draining point
9296000 already working at pc of the nextPC(in pim loop)
9379000 to get to the next next PC 8434 cycles
17729000 second drain detected
17730000 already in nextPC
30555 cycles
48284000 third drain detected
48285000 already in nextPC
2454 cycles
50738000 forth drain detected
50739000 already in nextPC
24031964 cycles
24082702000 total ticks

e) not actually drain! 4 loops (8 drains)
24082702000 total ticks

version	to 1	1	1 to 2	2	2 to 3	3	3 to 4	4	4 to end
drain	9295	43(9338)	8389 + 2(17729)	8 (17737)	30689 + 2(48428)	285(48713)	2233 + 2(resuming to nextPC)	684 (51632)	24029922
only second	-	-	-	-	48284(all ticks)	285(48569)	2235 (50804)	684(51488)	24030784
no drain	9295	-	8434	-	30555	-	2454	-	24031964

Table 1: tick time line

After cpu commit stage has detected the nextPC in PIM, it will deschedule the tick event, set cpu::PIMRestorePC and cpu::PIMRestoreFlag and schedule the switch to PIM or switch from PIM event which will reset all stages

The commit stage pc will be advanced to the next pc therefore after every commit, the cpu-¿pcState() would be the next pc and it seems that cpu::PIMRestorePC is not needed.

In lsq_unit_impl, I can set the dcacheport of that unit(a thread) to the pim dataport when switch to pim port. Add a function to lsq to switch the data port.

3.2 FU Pool Shrinking Test

Using Option: –l2_size='1MB' –l1d_size='128kB' –l1i_size='32kB'

With actual shrinking of the FU Pool(Int Add from 6 to 1, Int Mul from 4 to 1, FP_ALU from 4 to 1, FP_Mul from 2 to 1, SIMD_unit from 4 to 1): 3944829000 ticks (9718 Int ALU full)

Without actual shrinking of FU Pool: 3944153000 ticks (8560 Int ALU Full)

Using loop2(fewer cross loop dependency, without optimization): 4102073000 ticks (1581164 Int ALU FULL, 114639 Mem read FULL, 41639 dcache demand miss, 421520 dcache demand hits, 0.089902 dcache demand miss rate, 0.664883 l2 demand miss rate)

Without actual shrinking of FU Pool: 4102655000 ticks (4519 Int ALU FULL, 69634 MemRead FULL, 58000 dcache demand miss, 405116 hits, 0.125239 demand miss rate, 16654 l2 data miss, 8395 l2 data hits, 0.664857 l2 data miss rate, 0.984720 l2 inst miss rate)

Using loop3(-O2 of loop2): 3837063000 ticks(1069434 IntALU full, almost no other full, 0.149472 dcache demand miss rate, 0.669696 l2 data miss rate, 0 iew idle cycle, 888121 iew LDST queue full event)

Without actual shrinking of FU Pool: 3880152000 ticks (8568 IntALU full and almost no other full, 0.198803 dcache demand miss rate, 0.669817 l2cache demand miss rate, also iew idle cycle is 0, 888121 iew LDST queue full event)

Using 10000 repeat loop3:

Shrink: 9182069466000 ticks
not Shrink: 6316238743000 ticks

3.3 ROB Shrinking Test

Using loop3(-O2 of loop2):

With shrinking ROB entries by 4: 3837282000 ticks, 102477 ROB Full Events.

Without shrinking ROB entries: 3880152000, 46 ROB Full Events.

3.4 Fetch Size Test

Using loop3(-O2 of loop2):
With Shrinking Fetch Size: 3837282000 ticks(??!! why same as ROB shrinking), 46 ROB full events, 0.210884 insts fetched per cycle, 3018 cachelines fetched, iew inst execution rate(?): 0.400081, icache hits: 2212, icache miss: 803, icache miss rate: 0.266335, l2 inst miss rate: 0.983278(?), int ALU full event: 382, iew block cycles: 2688

Without Shrinking Fetch Size: 3880152000 ticks, 46 rob full, 0.208564 inst fetched per cycle(even lower than shrinking, this means fetch width of 2 is totally abundant?), iew inst execution rate: 0.395665, icache hits: 2215, icache miss: 804, icache miss rate: 0.266313, l2 inst miss rate: 0.983278, int ALU full event: 8568, iew block cycles: 2753

3.5 Shrinking ROB, Fetch and FU

ROB and Fetch:3880803000 ticks (loop3)

ROB, Fetch and FU: 3880780000 ticks (loop3): (2631 iew block cycles, 0.395519 iew exe rate(?), inst fetched per cycle: 0.208496, 1065335 ALU IntALU full, 237664 ROB Full Event)

Next step, change the memory port, firstly don't decrease memory delay and sim cycle number should increase compared to not changing port. secondly decrease memory delay and sim cycle number should decrease compared to changing port but not decreasing memory delay.

3.6 Switching inst Port to Connecting memory

Using loop 3 and the same cache config as before:

Changing the port: 3882596000 ticks (fetch rate: 0.208383 inst per cycle, 23650 icache stall cycle)

Not Changing the port: 3880152000 ticks (fetch rage: 0.208564 inst per cycle, 22620 icache stall cycle)

How to let fetching speed matter in the system? Why program pattern will make fetch speed the bottleneck?

3.7 Switching data Port to Connecting memory

Using loop 3 and the same cache config as before(without changing fetch, icache port, fu pool, ROB and the memory timing parameters):

invalidating cache blocks: 4008917000 ticks

not invalidating cache blocks: 3880152000 ticks

Changing the data port(also switch the inst port): 22126171000 ticks

Next steps: how to tune the parameters(including Xbar latencies) to change bottleneck so that modifying fu pool, fetch width, ROB will actually impact the run time on a large scale.

Develop testbenches, which has high locality and low locality.

4. Things TODO Now: Feb 18

a) modify the program or the cache size so that a high locality one won't even fit in l2 cache. make the array size to be 128 * 4 * 4 KB = 2 MB >1MB (l2 cache size)

b) run profiling for it

c) add caches to the PIM port and make them 4 times smaller than the main cpu cache

d) simulate PIM, normal system, only low locality PIM and pure PIM