Elastic Pipelines: Concurrency Issues

Arvind
Computer Science & Artificial Intelligence Lab
Massachusetts Institute of Technology

Inelastic vs Elastic Pipelines

In a Inelastic pipeline:
- typically only one rule; the designer controls precisely which activities go on in parallel
- downside: The rule can get too complicated -- easy to make a mistake; difficult to make changes

In an Elastic pipeline:
- several smaller rules, each easy to write, easier to make changes
- downside: sometimes rules do not fire concurrently when they should
It is better to think in terms of FIFOs as opposed to pipeline registers.

March 3, 2010

http://csg.csail.mit.edu/6.375

SFIFO (glue between stages)

```
interface SFIFO#(type t, type tr);
    method Action enq(t); // enqueue an item
    method Action deq(); // remove oldest entry
    method t first(); // inspect oldest item
    method Action clear(); // make FIFO empty
    method Bool find(tr); // search FIFO
endinterface
```

March 3, 2010

http://csg.csail.mit.edu/6.375

L09-4
Two-Stage Pipeline

module mkCPU#(Mem iMem, Mem dMem)(Empty);
  Reg#(Iaddress) pc <- mkReg(0);
  RegFile#(RName, Bit#(32)) rf <- mkRegFileFull();
  SFIFO#(InstTemplate, RName) bu <- mkSFifo(findf);
  Iinstr instr = iMem.read(pc);
  Iiaddress predIa = pc + 1;
  InstTemplate it = bu.first();
  rule fetch_decode ...
  endmodule

rule decodeAdd(instr matches Add{dst:.rd,src1:.ra,src2:.rb})
  bu.enq (EAdd{dst:rd,op1:rf[ra],op2:rf[rb]});
  pc <= predIa;
endrule

rule executeAdd(it matches EAdd{dst:.rd,op1:.va,op2:.vb})
  rf.upd(rd, va + vb);
  bu.deq();
endrule
Fetch & Decode Rule:
Reexamined

```plaintext
rule decodeAdd (instr matches Add{dst:.rd, src1:.ra, src2:.rb})
    bu.enq (EAdd{dst:rd, op1:rf[ra], op2:rf[rb]});
    pc <= predIa;
endrule
```

stall !

Fetch & Decode Rule:
corrected

```plaintext
rule decodeAdd (instr matches Add{dst:.rd, src1:.ra, src2:.rb})
    &&& !bu.find(ra) &&& !bu.find(rb))
    bu.enq (EAdd{dst:rd, op1:rf[ra], op2:rf[rb]});
    pc <= predIa;
endrule
```

L09-7
L09-8
Rules for Branch

rule atomicity ensures that pc update, and discard of pre-fetched instrs in bu, are done consistently

rule decodeBz (Instr instr)
  if (Instr instr matches Bz (condR:.rc, addrR:.addr))
    &&&
    !bu.find(rc) &&& !bu.find(addr));
  bu.enq (EBz (cond: rf[rc], tAddr: rf[addr]));
  pc <= predIa;
endrule

rule bzTaken (Instr instr)
  if (Instr instr matches EBz (cond:.vc, tAddr:.va)) &&& (vc==0));
  pc <= va;   bu.clear(); endrule

rule bzNotTaken (Instr instr)
  if (Instr instr matches EBz (cond:.vc, tAddr:.va)) &&&
    (vc != 0));
  bu.deq; endrule

Fetch & Decode Rule

rule fetch_and_decode
  if (!stallFunc (Instr instr, bu));
  bu.enq (Instr instr);
  pc <= predIa;
endrule

function InstrTemplate newIt (Instr instr);
  case (Instr instr) matches
    tagged Add {dst:.rd, src1:.ra, src2:.rb}:
      return EAdd {dst:rd, op1:rf[ra], op2:rf[rb]};
    tagged Bz {condR:.rc, addrR:.addr}:
      return EBz {cond: rf[rc], tAddr: rf[addr]};
    tagged Load {dst:.rd, addrR:.addr}:
      return ELoad {dst:rd, addrR:rf[addr]};
    tagged Store{valueR:.v, addrR:.addr}:
      return EStore {val:rf[v], addr:rf[addr]};
  endcase
endfunction
The Stall Signal

```haskell
Bool stall = stallFunc(instr, bu);
function Bool stallFunc (Instr instr,
    SFIFO#(InstTemplate, RName) bu);
    case (instr) matches
        tagged Add {dst:.rd,src1:.ra,src2:.rb}:
            return (bu.find(ra) || bu.find(rb));
        tagged Bz    {condR:.rc,addrR:.addr}:
            return (bu.find(rc) || bu.find(addr));
        tagged Load  {dst:.rd,addrR:.addr}:
            return (bu.find(addr));
        tagged Store {valueR:.v,addrR:.addr}:
            return (bu.find(v)) || bu.find(addr));
    endcase
endfunction
```

This need to search the contents of the FIFO is why we need an SFIFO, not just a FIFO

March 3, 2010 L09-11

The findf function

- When we make a searchable FIFO we need to supply a function that determines if a register is going to be updated by an instruction template
- mkSFifo can be parameterized by such a search function

```haskell
function Bool findf (RName r, InstrTemplate it);
    case (it) matches
        tagged EAdd{dst:.rd,op1:.v1,op2:.v2}:
            return (r == rd);
        tagged EBz {cond:.c,tAddr:.a}:
            return (False);
        tagged ELoad{dst:.rd,address:.a}:
            return (r == rd);
        tagged EStore{val:.v,address:.a}:
            return (False);
    endcase
endfunction
```

March 3, 2010 L09-12
**Execute Rule**

```
rule execute (True);
    case (it) matches
        tagged EAdd{dst:.rd,op1:.va,op2:.vb}:
            begin rf.upd(rd, va+vb); bu.deq(); end
        tagged EBz {cond:.cv,tAddr:.av}:
            if (cv == 0) then
                begin pc <= av; bu.clear(); end
            else bu.deq();
        tagged ELoad{dst:.rd,addr:.av}:
            begin rf.upd(rd, dMem.read(av)); bu.deq(); end
        tagged EStore{val:.vv,addr:.av}:
            begin dMem.write(av, vv); bu.deq(); end
    endcase
endrule
```

March 3, 2010 L09-13

---

**Concurrency**

```
rule fetch_and_decode (!stallFunc(instr, bu));
    bu.enq(newIt(instr,rf));
    pc <= predIa;
endrule

rule execute (True);
    case (it) matches
        tagged EAdd{dst:.rd,op1:.va,op2:.vb}:
            begin rf.upd(rd, va+vb); bu.deq(); end
        tagged EBz {cond:.cv,tAddr:.av}:
            if (cv == 0) then
                begin pc <= av; bu.clear(); end
            else bu.deq();
        tagged ELoad{dst:.rd,addr:.av}:
            begin rf.upd(rd, dMem.read(av)); bu.deq(); end
        tagged EStore{val:.vv,addr:.av}:
            begin dMem.write(av, vv); bu.deq(); end
    endcase
endrule
```

March 3, 2010 L09-14
The tension

- If the two rules never fire in the same cycle then the machine can hardly be called a pipelined machine
  - Scheduling cannot be too conservative

- If both rules are enabled and are executed together then in some cases wrong results would be produced
  - Too aggressive a scheduling would violate one-rule-at-time-semantics

Case 1: Back-to-back dependencies?
Two rules won’t be enabled together (stall function)

Case 2: Branch taken?
Two rules will be enabled together but only one rule should fire. branch-taken should have priority

The compiler issue

- Can the compiler detect all the conflicting conditions?
  - Important for correctness
- Does the compiler detect conflicts that do not exist in reality?
  - False positives lower the performance
  - The main reason is that sometimes the compiler cannot detect under what conditions the two rules are mutually exclusive or conflict free
- What can the user specify easily?
  - Rule priorities to resolve nondeterministic choice

In many situations the correctness of the design is not enough; the design is not done unless the performance goals are met
Execution rules

Split the execution rule for analysis

```mermaid
graph LR
B --> C[CPU]
C --> D[rf]
D --> E[execAdd]
```

```mermaid
graph LR
B --> C[CPU]
C --> D[rf]
D --> E[execAdd]
```

```mermaid
graph LR
B --> C[CPU]
C --> D[rf]
D --> E[execAdd]
```

```mermaid
graph LR
B --> C[CPU]
C --> D[rf]
D --> E[execAdd]
```

March 3, 2010 L09-17
http://csg.csail.mit.edu/6.375

Concurrency analysis

Add Rule

```mermaid
graph LR
B --> C[CPU]
C --> D[rf]
D --> E[execAdd]
```

March 3, 2010 L09-17
http://csg.csail.mit.edu/6.375

March 3, 2010 L09-17
http://csg.csail.mit.edu/6.375

March 3, 2010 L09-17
http://csg.csail.mit.edu/6.375

March 3, 2010 L09-17
http://csg.csail.mit.edu/6.375

March 3, 2010 L09-17
http://csg.csail.mit.edu/6.375
What concurrency do we want?

- If fetch and execAdd happened in the same cycle and the meaning was:
  - fetch < execAdd
    - instructions will fly through the FIFO (No pipelining!)
    - rf and bu modules will need the properties;
      rf: sub < upd
      bu: {find, enq} < {first, deq}
  - execAdd < fetch
    - execAdd will make space for the fetched instructions (i.e., how pipelining is supposed to work)
    - rf and bu modules will need the properties;
      rf: upd < sub
      bu: {first, deq} < {find, enq}

Now we will focus only on the pipeline case.

Concurrency analysis

Branch Rules

- Rule bzTaken(it matches tagged EBz {cond:.cv,tAddr:.av} && (cv == 0));
  pc <= av; bu.clear(); endrule
- Rule bzNotTaken(it matches tagged EBz {cond:.cv,tAddr:.av} && !(cv == 0));
  bu.deq(); endrule

- bzTaken < fetch ⇒
  - Should be treated as a conflict; give priority to bzTaken
- bzNotTaken < fetch ⇒
  - bu: {first, deq} < {find, enq}
Concurrent analysis

Load-Store Rules

**Rule fetch_and_decode**: (!stallfunc(instr, bu));

bu.enq(newIt(instr, rf));

pc <= predIa;

endrule

**Rule execLoad**: (it matches tagged ELoad{dest: rd, addr: av});

rf.upd(rd, dMem.read(av)); bu.deq();

endrule

**Rule execStore**: (it matches tagged EStore{val: vv, addr: av});

dMem.write(av, vv); bu.deq();

endrule


- **execLoad < fetch** ⇒
  - rf: upd < sub; bu: {first, deq} < {find, enq}
- **execStore < fetch** ⇒
  - bu: {first, deq} < {find, enq}

---

Properties Required of Register File and FIFO for Instruction Pipelining

- **Register File**:
  - rf.upd(r1, v) < rf.sub(r2)
  - **Bypass RF**
- **FIFO**
  - bu: {first, deq} < {find, enq} ⇒
    - bu.first < bu.find
    - bu.first < bu.enq
    - bu.deq < bu.find
    - bu.deq < bu.enq
  - **Pipeline SFIFO**
One Element Searchable Pipeline SFIFO

module mkSFIFO1#(function Bool findf(tr r, t x))(SFIFO#(t, tr));

Reg#(t) data <- mkRegU();
Reg#(Bool) full <- mkReg(False);
RWire#(void) deqEN <- mkRWire();
Bool deqp = isValid (deqEN.wget());
method Action enq(t x) if (!full || deqp);
  full <= True; data <= x;
endmethod
method Action deq() if (full);
  full <= False; deqEN.wset(?);
endmethod
method t first() if (full);
  return (data);
endmethod
method Action clear();
  full <= False;
endmethod
method Bool find(tr r);
  return (findf(r, data) && full);
endmethod endmodule

Register File concurrency properties

◆ Normal Register File implementation guarantees:
  ■ rf.sub < rf.upd
    * that is, reads happen before writes in concurrent execution
◆ But concurrent rf.sub(r1) and rf.upd(r2,v)
  where r1 ≠ r2 behaves like both
  ■ rf.sub(r1) < rf.upd(r2,v)
  ■ rf.sub(r1) > rf.upd(r2,v)
◆ To guarantee rf.upd < rf.sub
  ■ Either bypass the input value to output when register names match
  ■ Or make sure that on concurrent calls rf.upd and rf.sub do not operate on the same register
Bypass Register File

module mkBypassRFFull(RegFile#(RName,Value));

RegFile#(RName,Value) rf <- mkRegFileFullWCF();
RWire#(Tuple2#(RName,Value)) rw <- mkRWire();

method Action upd (RName r, Value d);
  rf.upd(r,d);
  rw.wset(tuple2(r,d));
endmethod

method Value sub(RName r);
  case rw.wget() matches
    case
      tagged Valid (.wr,.d):
        return (wr==r) ? d :
         _rf.sub(r);
      tagged Invalid:      return rf.sub(r);
    endcase
endmethod
endmodule

Since our rules do not really require a Bypass Register File, the overhead of bypassing can be avoided by simply using the “Config Regfile”
Concurrency analysis

Two-stage Pipeline

rule fetch_and_decode (!stallfunc(instr, bu));
  bu.enq(newIt(instr, rf));
  pc <= predIa;
endrule

rule execAdd
  (it matches tagged EAdd{dst:.rd, src1:.va, src2:.vb});
  rf.upd(rd, va+vb); bu.deq(); endrule

rule BzTaken(it matches tagged Bz {cond:.cv, addr:.av})
  &&& (cv == 0);
  pc <= av; bu.clear(); endrule

rule BzNotTaken(it matches tagged Bz {cond: cv, addr: av});
  &&& !(cv == 0);
  bu.deq(); endrule

rule execLoad(it matches tagged ELoad{dst:.rd, addr:.av});
  rf.upd(rd, dMem.read(av)); bu.deq(); endrule

rule execStore(it matches tagged EStore{value:.vv, addr:.av});
  dMem.write(av, vv); bu.deq(); endrule

Lot of nontrivial analysis but no change in processor code!

Needed Fifos and Register files with the appropriate concurrency properties
Bypassing

- After decoding the `newIt` function must read the new register values if available (i.e., the values that are still to be committed in the register file)
  - Will happen automatically if we use `bypassRF`

- The instruction fetch must not stall if the new value of the register to be read exists
  - The old stall function is correct but unable to take advantage of bypassing and stalls unnecessarily

The stall function for the elastic pipeline

```plaintext
function Bool newStallFunc (Instr instr, SFIFO#(InstTemplate, RName) bu);
  case (instr) matches
    tagged Add (dst:.rd,srcl:.ra,srcc:.rb):
      return (bu.find(ra) || bu.find(rb));
    tagged Bz    (cond:.rc,addr:.addr):
      return (bu.find(rc) || bu.find(addr));
  ...

bu.find in our Pipeline SFIFO happens after dq. This means that if bu can hold at most one instruction like in the inelastic case, we do not have to stall. Otherwise, we will still need to check for hazards and stall.

No change in the stall function
```