Readings

• Read on your own:

• To be discussed in class:
Outline

• Memory Data Flow
  – Scalable Load/Store Queues
  – Memory-level parallelism (MLP)

• Register Data Flow
  – Instruction scheduling overview
    • Scheduling atomicity
    • Speculative scheduling
    • Scheduling recovery
  – EOLE: Effective Implementation of Value Prediction

• Instruction Flow
  – Revolver: efficient loop execution
  – Transparent Control Independence
Memory Dataflow
Scalable Load/Store Queues

- Load queue/store queue
  - Large instruction window: many loads and stores have to be buffered (25%/15% of mix)
  - Expensive searches
    - positional-associative searches in SQ,
    - associative lookups in LQ
    - coherence, speculative load scheduling
  - Power/area/delay are prohibitive
Store Queue/Load Queue Scaling

- Multilevel store queue [Akkary et al., MICRO 03]
- Bloom filters (quick check for independence) [Sethumadhavan et al., MICRO 03]
- Eliminate associative load queue via replay [Cain, ISCA 2004]
  - Issue loads again at commit, in order
  - Check to see if same value is returned
  - Filter load checks for efficiency:
    - Most loads don’t issue out of order (no speculation)
    - Most loads don’t coincide with coherence traffic
Store Vulnerability Window (SVW) [Roth, ISCA 05]

- Assign sequence numbers to stores
- Track writes to cache with sequence numbers
- Efficiently filter out safe loads/stores by only checking against writes in vulnerability window
  
  - At dispatch, load captures SN of oldest uncommitted store
  
  - At commit, if cache SN is older, then load is safe
    - Stores write SN to bloom filter to make check cheaper
  
  - Otherwise, load gets replayed at commit
NoSQ

[Sha et al., MICRO 06]

• Rely on load/store alias prediction to directly connect dependent pairs
  – Memory cloaking [Moshovos/Sohi, ISCA 1997]
  – More accurate, path-dependent predictor

• Use SVW technique to check
  – Replay load only if necessary
  – Train load/store alias predictor
Store/Load Optimizations

• Several other similar concurrent proposals
  – DMDC [Castro et al., MICRO 06]
  – Fire-and-forget [Subramanian/Loh, MICRO 06]

• Weakness: predictor still fails
  – Glass jaw: should fail gracefully, not fall off a cliff
  – Risk of new/unknown workload that is unpredictable
Outline

• Memory Data Flow
  – Scalable Load/Store Queues
  – Memory-level parallelism (MLP)

• Register Data Flow
  – Instruction scheduling overview
    • Scheduling atomicity
    • Speculative scheduling
    • Scheduling recovery
  – EOLE: Effective Implementation of Value Prediction

• Instruction Flow
  – Revolver: efficient loop execution
  – Transparent Control Independence
Memory-Level Parallelism

[Glew, ASPLOS 98 “Wild and Crazy Ideas Session”]

• Tolerate/overlap memory latency
  – Once first miss is encountered, find another one

• Naïve solution
  – Implement a very large ROB, IQ, LSQ
  – Power/area/delay make this infeasible

• Instead, build *virtual* instruction window
Runahead Execution

• Use poison bits to eliminate miss-dependent load program slice
  – Forward load slice processing is a very old idea
    • Massive Memory Machine [Garcia-Molina et al. 84]
    • Datascalor [Burger, Kaxiras, Goodman 97]
  – Runahead proposed by [Dundas, Mudge 97]

• Checkpoint state, keep running beyond miss

• When miss completes, return to checkpoint
  – May need runahead cache for store/load communication [Mutlu et al, HPCA 03]

• All runahead activity is wasted (re-execute everything)
Waiting Instruction Buffer

[Lebeck et al. ISCA 2002]

• Capture forward load slice in separate buffer
  – Propagate poison bits to identify slice
• Relieve pressure on issue queue
• Reinsert instructions when load completes
• Very similar to Intel Pentium 4 replay mechanism
  – But not publicly known at the time
• Makes recovery from load latency mispredicts easier/cheaper
• Scope still limited by ROB size
Continual Flow Pipelines

[Srinivasan et al. ASPLOS 2004]

• Slice buffer extension of WIB
  – Store operands in slice buffer as well to free up ROB/buffer entries in OOO window
  – Also relieve pressure on rename/physical registers

• Applicable to
  – data-capture machines (Intel P6) or
  – physical register file machines (Pentium 4)

• iCFP extends idea to in-order CPUs [Hilton et al., HPCA 09]

• Challenge: how to buffer loads/stores
  – See [Gandhi et al, ISCA 05]
Long Term Parking

[Sembrant et al., MICRO 2015]

- Proactively defers allocation of microarchitectural resources to non-critical instructions
  - WIB, CFP are reactive (after miss occurs)
- Relies on predictor, LTP structure
Outline

• Memory Data Flow
  – Scalable Load/Store Queues
  – Memory-level parallelism (MLP)

• Register Data Flow
  – Instruction scheduling overview
    • Scheduling atomicity
    • Speculative scheduling
    • Scheduling recovery
  – EOLE: Effective Implementation of Value Prediction

• Instruction Flow
  – Revolver: efficient loop execution
  – Transparent Control Independence
Register Dataflow
Instruction scheduling

- A process of mapping a series of instructions into execution resources
  - Decides *when* and *where* an instruction is executed

- Data dependence graph
- Mapped to two FUs
Instruction scheduling

• A set of **wakeup** and **select** operations
  – **Wakeup**
    • Broadcasts the tags of parent instructions selected
    • Dependent instruction gets matching tags, determines if source operands are ready
    • Resolves true data dependences
  – **Select**
    • Picks instructions to issue among a pool of ready instructions
    • Resolves resource conflicts
      – Issue bandwidth
      – Limited number of functional units / memory ports
Scheduling loop

• Basic wakeup and select operations

Select logic

Wakeup logic

broadcast the tag of the selected inst

grant 0
request 0

grant 1
request 1

......

grant n
request n

selected

ready - request

issue to FU
## Wakeup and Select

<table>
<thead>
<tr>
<th></th>
<th>FU0</th>
<th>FU1</th>
<th>Ready inst to issue</th>
<th>Wakeup / select</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>Select 1, 2, 3, 4</td>
</tr>
<tr>
<td>n+1</td>
<td>2</td>
<td>3</td>
<td>2, 3, 4</td>
<td>Select 2, 3, 5, 6</td>
</tr>
<tr>
<td>n+2</td>
<td>5</td>
<td>4</td>
<td>4, 5</td>
<td>Select 4, 5, 6</td>
</tr>
<tr>
<td>n+3</td>
<td>6</td>
<td></td>
<td>6</td>
<td>Select 6</td>
</tr>
</tbody>
</table>

The diagram on the left illustrates the flow of instructions, with arrows indicating the direction of execution.

The table outlines the process of waking up and selecting instructions for execution at each step. The columns for FU0 and FU1 show the ready instructions that can be issued, and the rightmost column indicates the corresponding actions for each step.
Operations in the scheduling loop must occur within a single clock cycle

- For back-to-back execution of dependent instructions

<table>
<thead>
<tr>
<th>cycle</th>
<th>Atomic scheduling</th>
<th>Non-Atomic 2-cycle scheduling</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>select 1</td>
<td>select 1</td>
</tr>
<tr>
<td></td>
<td>wakeup 2, 3</td>
<td></td>
</tr>
<tr>
<td>n+1</td>
<td>Select 2, 3</td>
<td>wakeup 2, 3</td>
</tr>
<tr>
<td></td>
<td>wakeup 4</td>
<td></td>
</tr>
<tr>
<td>n+2</td>
<td>Select 4</td>
<td>select 2, 3</td>
</tr>
<tr>
<td>n+3</td>
<td></td>
<td>wakeup 4</td>
</tr>
<tr>
<td>n+4</td>
<td></td>
<td>select 4</td>
</tr>
</tbody>
</table>
Implication of scheduling atomicity

• Pipelining is a standard way to improve clock frequency

• Hard to pipeline instruction scheduling logic without losing ILP
  – ~10% IPC loss in 2-cycle scheduling
  – ~19% IPC loss in 3-cycle scheduling

• A major obstacle to building high-frequency microprocessors
Scheduling atomicity & non-data-capture scheduler

- Multi-cycle scheduling loop

- Scheduling atomicity is not maintained
  - Separated by extra pipeline stages (Disp, RF)
  - Unable to issue dependent instructions consecutively

→ solution: speculative scheduling
Speculative Scheduling

- Speculatively wakeup dependent instructions even before the parent instruction starts execution
  - Keep the scheduling loop within a single clock cycle

- But, nobody knows what will happen in the future

- Source of uncertainty in instruction scheduling: loads
  - Cache hit / miss, bank conflict
  - Store-to-load aliasing
    - eventually affects timing decisions

- Scheduler assumes that all types of instructions have pre-determined fixed latencies
  - Load instructions are assumed to have a common case (over 90% in general)
    - $DL1$ hit latency
  - If incorrect, subsequent (dependent) instructions are replayed
Speculative Scheduling

• Overview

Unlike the original Tomasulo’s algorithm
  ■ Instructions are scheduled BEFORE actual execution occurs
  ■ Assumes instructions have pre-determined fixed latencies
    ■ ALU operations: fixed latency
    ■ Load operations: assumes $DL1$ latency (common case)
Scheduling replay

• Speculation needs verification / recovery
  – There’s no free lunch

• If the actual load latency is longer (i.e. cache miss) than what was speculated
  – Best solution (disregarding complexity): replay data-dependent instructions issued under load shadow
Wavefront propagation

- Speculative execution wavefront
  - speculative image of execution (from scheduler’s perspective)

- Both wavefronts propagate along dependence edges at the same rate (1 level / cycle)
  - the real wavefront runs behind the speculative wavefront

- The load resolution loop delay complicates the recovery process
  - scheduling miss is notified a couple of clock cycles later after issue
Load resolution feedback delay in instruction scheduling

- Scheduling runs multiple clock cycles ahead of execution
  - But, instructions can keep track of only one level of dependence at a time (using source operand identifiers)
Issues in scheduling replay

- Cannot stop speculative wavefront propagation
  - Both wavefronts propagate at the same rate
  - Dependent instructions are unnecessarily issued under load misses
Requirements of scheduling replay

- Propagation of recovery status should be faster than speculative wavefront propagation
- Recovery should be performed on the transitive closure of dependent instructions

- Conditions for ideal scheduling replay
  - All mis-scheduled dependent instructions are invalidated instantly
  - Independent instructions are unaffected

- Multiple levels of dependence tracking are needed
  - e.g. Am I dependent on the current cache miss?
  - Longer load resolution loop delay → tracking more levels
Scheduling replay schemes

- Alpha 21264: Non-selective replay
  - Replays all dependent and independent instructions issued under load shadow
  - Analogous to squashing recovery in branch misprediction
  - Simple but high performance penalty
    - Independent instructions are unnecessarily replayed
Position-based selective replay

- Ideal selective recovery
  - replay dependent instructions only
- Dependence tracking is managed in a matrix form
  - Column: load issue slot, row: pipeline stages
Outline

• Memory Data Flow
  – Scalable Load/Store Queues
  – Memory-level parallelism (MLP)

• Register Data Flow
  – Instruction scheduling overview
    • Scheduling atomicity
    • Speculative scheduling
    • Scheduling recovery
  – EOLE: Effective Implementation of Value Prediction

• Instruction Flow
  – Revolver: efficient loop execution
  – Transparent Control Independence
Definition

• What is value prediction? Broadly, three salient attributes:
  1. Generate a speculative value (predict)
  2. Consume speculative value (execute)
  3. Verify speculative value (compare/recover)

• This subsumes branch prediction
  Focus here on operand values

Limited ILP = 1.3

Dataflow Execution

Enhanced ILP = 4

Beyond Dataflow

Verify Predictions

Predict
Some History

• “Classical” value prediction
  – Independently invented by 4 groups in 1995-1996
Why?

• Possible explanations:
  1. Natural evolution from branch prediction
  2. Natural evolution from memoization
  3. Natural evolution from rampant speculation
     • Cache hit speculation
     • Memory independence speculation
     • Speculative address generation [Zero Cycle Loads – Austin/Sohi]
  4. **Improvements in tracing/simulation technology**
     • “There’s a lot of zeroes out there.” (C. Wilkerson)
     • Values, not just instructions & addresses
        – TRIP6000 [A. Martin-de-Nicolas, IBM]
What Happened?

• Considerable academic interest
  – Dozens of research groups, papers, proposals
• No industry uptake for a long time
  – Intel (x86), IBM (PowerPC), HAL (SPARC) all failed
• Why?
  – Modest performance benefit (< 10%)
  – Power consumption
    • Dynamic power for extra activity
    • Static power (area) for prediction tables
  – Complexity and correctness (risk)
    • Subtle memory ordering issues [MICRO ’01]
    • Misprediction recovery [HPCA ’04]
Performance?

• Relationship between timely fetch and value prediction benefit [Gabbay & Mendelson, ISCA’98]
  Value prediction doesn’t help when the result can be computed before the consumer instruction is fetched
• Accurate, high-bandwidth fetch helped
  – Wide trace caches studied in late 1990s
  – Much better branch prediction today (neural, TAGE)
• Industry was pursuing frequency, not ILP (GHz race)
Future Adoption?

- Classical value prediction will only make it in the context of a very different microarchitecture
  - One that explicitly and aggressively exposes ILP
- Promising trends
  - Deep pipelining craze is over
  - High frequency mania is over
- Architects are pursuing ILP once again
  - Value prediction may have another opportunity
  - Rumors of 4 design teams considering it
Some Recent Interest

- **VTAGE** [Perais/Seznec, HPCA 14]
  - Solves many practical problems in the predictor

- **EOLE** [Perais/Seznec, ISCA 14]
  - Value predicted operands reduce need for OoO
  - Execute some ops early, some late, outside OoO
  - Smaller, faster OoO window

- **Load Value Approximation**
  [San Miguel/Badr/Enright Jerger, MICRO-47][Thwaites et al., PACT 2014]

- **DLVP** [Sheikh/Cain/Damodaran, MICRO-50]
Introducing Early Execution

Execute ready single-cycle instructions in parallel with *Rename*, in-order.

**Do not** dispatch to the IQ.
Values come from:
- Decode (Immediate)
- Value Predictor
- Bypass Network

Execute what you can, write in the PRF with the ports provisioned for VP.
Introducing Late Execution

Execute single-cycle predicted instructions just before retirement, **in-order**.

**Do not** dispatch to the IQ either.
Execute just before validation and retirement by leveraging the ports provisioned for validation.
Much less instructions enter the IQ: We may be able to reduce the issue-width:
  • Simpler IQ.
  • Less ports on the PRF.
  • Less bypass.
 ➢ Simpler OoO.

Non critical predictions become useful as the instructions can be late-executed.

What about hardware cost?
Outline

• Memory Data Flow
  – Scalable Load/Store Queues
  – Memory-level parallelism (MLP)

• Register Data Flow
  – Instruction scheduling overview
    • Scheduling atomicity
    • Speculative scheduling
    • Scheduling recovery
  – EOLE: Effective Implementation of Value Prediction

• Instruction Flow
  – Revolver: efficient loop execution
  – Transparent Control Independence
Motivation – Loop Evolution

AMD 29K
AMD Jaguar / Qualcomm Krait
Intel Core 2 / ARM Cortex-A9
Intel Nehalem / ARM Cortex-A15

In-Place Execution

Mikko Lipasti - University of Wisconsin
Motivation – Loop Opportunity

SD-VBS  MiBench  SpecInt  SpecFP

Total Dynamic Instructions Captured (%)

Loop Buffer Capacity (instructions)
In-Place Loop Execution

• Execute loops in-place
  – Eliminate fetch/branch/dispatch overheads
  – Reuse back-end structures
• Necessary Modifications
  – Loop Detection / Dispatch Logic
  – Dependence Linking
  – Reusable backend structures
    • IQ Entries, LSQ Entries, Physical Registers
Frontend Loop Logic

- **Primary Responsibilities**
  - Identify loops and resource requirements
  - Dispatch loops
  - Incorporate feedback

- **Loop Identification**
  - Triggered by backwards branch
  - Unlimited control flow
  - Utilizes simple state machine and registers

- **Details in** [Hayenga, HPCA 2014]
Loop Types

Simple

start:  ld r0,[r1, r3]
        str r0, [r2, r3]
        sub r3, r3, #4
        cmp r3,#0
        bne start

Complex

start:  ld r0, [r1, r2]
        cmp r0, #0
        beq skip
        str #0xF, [r1, r2]
skip:   sub r2, r2, #4
        cmp r2, 0
        bne start

Early Exit

start:  ldr r0, [r1]
        cmp r0, #0
        beq exit
        add r1,r1, #1
        b start
Queue Management

Source Code
while(*dst++ = *src++) {}
LSQ – Conventional Ordering

<table>
<thead>
<tr>
<th>Store Color</th>
<th>I-Stream</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$ld \ r6, [r7]$</td>
</tr>
<tr>
<td></td>
<td>$st \ r8, [r9]$</td>
</tr>
<tr>
<td>1</td>
<td>$ld \ r0, [r1, r3]$</td>
</tr>
<tr>
<td></td>
<td>$st \ r0, [r2, r3]$</td>
</tr>
<tr>
<td></td>
<td>$add \ r3, r3, #1$</td>
</tr>
<tr>
<td></td>
<td>$cmp \ r0, #0$</td>
</tr>
<tr>
<td></td>
<td>$bne$</td>
</tr>
<tr>
<td>2</td>
<td>$ld \ r0, [r1, r3]$</td>
</tr>
<tr>
<td></td>
<td>$st \ r0, [r2, r3]$</td>
</tr>
<tr>
<td></td>
<td>$add \ r3, r3, #1$</td>
</tr>
<tr>
<td></td>
<td>$cmp \ r0, #0$</td>
</tr>
<tr>
<td></td>
<td>$bne$</td>
</tr>
<tr>
<td>3</td>
<td>$ld \ r0, [r7]$</td>
</tr>
<tr>
<td></td>
<td>$st \ r1, [r5]$</td>
</tr>
</tbody>
</table>

Before Loop

Iteration #1

Iteration #2

After Loop
## LSQ – Loop Ordering

<table>
<thead>
<tr>
<th>Store Color</th>
<th>I-Stream</th>
<th>Before Loop</th>
<th>Iteration #1</th>
<th>Iteration #2</th>
<th>After Loop</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1</td>
<td>ld r6, [r7]</td>
<td></td>
<td>st r8, [r9]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 2</td>
<td>ld r0, [r1, r3]</td>
<td></td>
<td>st r0, [r2, r3]</td>
<td>add r3, r3, #1</td>
<td>cmp r0, #0 bne</td>
</tr>
<tr>
<td>1 2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3 4</td>
<td>ld r0, [r7]</td>
<td></td>
<td>st r1, [r5]</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
In-place Loop Cache Benefit

On average 20% fewer instructions fetched
Still significant opportunity remaining
Outline

• Memory Data Flow
  – Scalable Load/Store Queues
  – Memory-level parallelism (MLP)

• Register Data Flow
  – Instruction scheduling overview
    • Scheduling atomicity
    • Speculative scheduling
    • Scheduling recovery
  – EOLE: Effective Implementation of Value Prediction

• Instruction Flow
  – Revolver: efficient loop execution
  – Transparent Control Independence
Transparent Control Independence

[Al-Zawawi et al., ISCA 07]

• Control flow graph convergence
  – Execution reconverges after branches
  – If-then-else constructs, etc.

• Can we fetch/execute instructions beyond convergence point?
  – Significant potential for ILP shown by limit study [Lam/Wilson, ISCA 92]

• How do we resolve ambiguous register and memory dependences?

• Slides from Al-Zawawi ISCA presentation follow
Control independence basics

- Control-dependent (CD)
  - R5 ←
- Control-independent data-dependent (CIDD)
  - reconv.
  - ← R5
- Control-independent data-independent (CIDI)
Four steps for exploiting CI
Four steps for exploiting CI

1. Identify reconv. point
Four steps for exploiting CI

1. Identify reconv. point

2. Remove/Insert CD inst.
Four steps for exploiting CI

1. Identify reconv. point
2. Remove/Insert CD inst.
3. Identify CIDD inst.
Four steps for exploiting CI

1. Identify reconv. point
2. Remove/Insert CD inst.
3. Identify CIDD inst.
4. Repair CIDD inst.
   a) Fix data dependencies
   b) Re-execute CIDD inst.
TCI misprediction recovery

Leverage checkpointed source values to mimic the effect of program order.

Exploit coarse-grain checkpoint-based retirement to relax ordering constraints.

Misprediction recovery program

Leverage branch checkpoint for correct CIDD instructions.

Completed instructions free their resources.

Checkpoint-based retirement enables aggressive register reclamation (e.g., CPR).

Completed instructions (no CIDD).

CIDD instructions provides checkpoint capabilities.

Misprediction branch checkpoint.

Checkpoint CIDI-supplied source values.
Transparent Control Independence

TCI repairs program state, not program order

TCI pipeline is recovery-free

• Transparent recovery by fetching additional instructions with checkpointed source values

TCI pipeline is free-flowing

• Leverage conventional speculation to execute correct and incorrect instructions quickly and efficiently
• Completed instructions free their resources
TCI microarchitecture

- Add repair rename map
- Add selective re-execution buffer (RXB)
Predict the branch

Instructions execute and leave the pipeline when done
Construct recovery program

Copy duplicate of CIDD inst. with their source values into RXB
Insert correct CD instructions

Load branch checkpoint into repair rename map, then fetch correct CD inst.
Inject duplicate CIDD inst. with their checkpointed source values

4 re-execute CIDD
Copy corrected register mappings from repair map to spec. map
Transparent Control Independence

• TCI employs CFP-like slice buffer to reconstruct state
  – Instructions with ambiguous dependences buffered
  – Reinsert them the same way forward load miss slice is reinserted
• “Best” CI proposal to date, but still very complex and expensive, with moderate payback
• Main reason to pursue CI: mispredicted branches
  – This is a moving target
  – Branch misprediction rates have dropped significantly even since 2007
Summary

• Memory Data Flow
  – Scalable Load/Store Queues
  – Memory-level parallelism (MLP)

• Register Data Flow
  – Instruction scheduling overview
    • Scheduling atomicity
    • Speculative scheduling
    • Scheduling recovery
  – EOLE: Effective Implementation of Value Prediction

• Instruction Flow
  – Revolver: efficient loop execution
  – Transparent Control Independence