Dynamic Memory Scheduling

EECS 470: Computer Architecture
Lecture Overview

• Dynamic Scheduling Overview
• The Memory Scheduling Problem
• Conventional Solutions
  – In-order Load/Store Scheduling
  – Blind Dependence Scheduling
  – Conservative Dataflow Scheduling
• Recent Developments
  – Memory Dependence Speculation
  – Memory Renaming
• Future Directions
Dynamic Scheduling Overview

Program Order

Static Schedule
(in-order 2-way)

Dynamic Schedule
(out-of-order 2-way)

• goal: improve performance via effective exploitation of ILP
  – most effective around branches, stores, with few registers

• *dynamic scheduler* selects schedule, manages communication and synchronization
Dynamic Scheduling Pipeline

In-order Dispatch

Out-of-order Execute

In-order Commit

Dynamic Scheduler
(One) Dynamic Scheduler Implementation

- **synchronization** managed by scheduler logic
- **communication** through input/output networks
- infrastructure geared towards register communication
Benefits of Register Communication

• directly specified dependencies (contained in inst)
  – accurate description of communication
    • no false or missing dependency edges
    • permits realization of dataflow schedule
  – early description of communication
    • allows scheduler logic to be pipelined without impacting speed of communication

• small communication name space
  – fast access to communication storage
    • possible to map entire communication space (no tags)
    • possible to bypass communication storage
The Memory Scheduling Problem

• loads/stores also have dependencies through memory
  – described by effective addresses

• cannot directly leverage existing infrastructure
  – indirectly specified memory dependencies
    • dataflow schedule is a function of program computation, prevents
      accurate description of communication early in the pipeline
    • pipelined scheduler slow to react to addresses
      – large communication space \(2^{32-64}\) bytes!
    • cannot fully map communication space, requires more complicated
      cache and/or store forward network
Requirements for a Solution

• **accurate** description of memory dependencies
  – no (or few) missing or false dependencies
  – permit realization of dataflow schedule

• **early** presentation of dependencies
  – permit pipelining of scheduler logic

• **fast** access to communication space
  – preferably as fast as register communication
    (zero cycles)
Lecture Overview

• Dynamic Scheduling Overview
• The Memory Scheduling Problem
• Conventional Solutions
  – In-order Load/Store Scheduling
  – Blind Dependence Scheduling
  – Conservative Dataflow Scheduling
• Recent Developments
  – Memory Dependence Speculation
  – Memory Renaming
• Future Directions
In-order Load/Store Scheduling

- schedule all loads and stores in program order
  - cannot violate true data dependencies (non-speculative)

- capabilities/limitations:
  - not accurate - may add many false dependencies
  - early presentation of dependencies (no addresses)
  - not fast, all communication through memory structures

- found in in-order issue pipelines
In-order Load/Store Scheduling Example

Dependencies

<table>
<thead>
<tr>
<th>True Order</th>
<th>Realized Order</th>
</tr>
</thead>
<tbody>
<tr>
<td>st X</td>
<td>st X</td>
</tr>
<tr>
<td>ld Y</td>
<td>ld Y</td>
</tr>
<tr>
<td>st Z</td>
<td>st Z</td>
</tr>
<tr>
<td>ld X</td>
<td>ld X</td>
</tr>
<tr>
<td>ld Z</td>
<td>ld Z</td>
</tr>
</tbody>
</table>

program order

ld Y
st Z
ld X
ld Z

true realized

time

ld Y
st Z
ld X
ld Z
Blind Dependence Speculation

- schedule loads and stores when register dependencies satisfied
  - may violate true data dependencies (speculative)

- capabilities/limitations:
  - accurate - if little in-flight communication through memory
  - early presentation of dependencies (no dependencies!)
  - not fast, all communication through memory structures

- most common with small windows
Blind Dependence Speculation Example

Dependencies

true realized

st X st X st X st X st X st X
1d Y 1d Y 1d Y 1d Y 1d Y 1d Y
st Z st Z st Z st Z st Z st Z
1d X 1d X 1d X 1d X 1d X 1d X
1d Z 1d Z 1d Z 1d Z 1d Z 1d Z

mispeculation detected!
Questions

• Suggest two ways to detect blind load mispeculation

• Suggest two ways to recover from blind load mispeculation
The Case for More/Less Accurate Dependence Specification

- small windows: blind speculation is accurate for most programs, compiler can register allocate most short term communication
- large windows: blind speculation performs poorly, many memory communications in execution window

[For 099.go, from Moshovos97]
Conservative Dataflow Scheduling

- schedule loads and stores when all dependencies known satisfied
  - conservative - won’t violate true dependencies (non-speculative)

- capabilities/limitations:
  - accurate only if addresses arrive early
  - late presentation of dependencies (verified with addresses)
  - not fast, all communication through memory and/or complex store forward network

- common for larger windows
### Conservative Dataflow Scheduling

#### Dependencies

<table>
<thead>
<tr>
<th>True</th>
<th>Realized</th>
</tr>
</thead>
<tbody>
<tr>
<td>st X</td>
<td>st X</td>
</tr>
<tr>
<td>ld Y</td>
<td>ld Y</td>
</tr>
<tr>
<td>st?Z</td>
<td>st?Z</td>
</tr>
<tr>
<td>ld X</td>
<td>ld X</td>
</tr>
<tr>
<td>ld Z</td>
<td>ld Z</td>
</tr>
</tbody>
</table>

#### Time

- program order
  - st X
  - ld Y
  - st?Z
  - ld X
  - ld Z

- realized order
  - st X
  - ld Y
  - st?Z
  - ld X
  - ld Z

- stall cycle

---

Memory Scheduling

EECS 470

Advanced Computer Architecture Lab
University of Michigan
Questions

• What if no dependent store or unknown store address is found?

• Describe the logic used to locate dependent store instructions

• What is the tradeoff between small and large memory schedulers?

• How should uncached loads/stores be handled? Video memory?
Lecture Overview

- Dynamic Scheduling Overview
- The Memory Scheduling Problem
- Conventional Solutions
  - In-order Load/Store Scheduling
  - Blind Dependence Scheduling
  - Conservative Dataflow Scheduling
- Recent Developments
  - Memory Dependence Speculation [Moshovos97]
  - Memory Renaming [Tyson/Austin97]
- Future Directions
Memory Dependence Speculation [Moshovos97]

- schedule loads and stores when **data** dependencies satisfied
  - uses dependence predictor to match sourcing stores to loads
  - doesn't wait for addresses, may violate true dependencies (speculative)

- capabilities/limitations:
  - accurate as predictor
  - early presentation of dependencies (data addresses not used in prediction)
  - not fast, all communication through memory structures

Dependencies

true realized

\[
\begin{align*}
\text{st?X} \\
\text{ld Y} \\
\text{st?Z} \\
\text{ld X} \\
\text{ld Z}
\end{align*}
\]
Dependence Speculation - In a Nutshell

- assumes static placement of dependence edges is persistent
  - good assumption!
- common cases:
  - accesses to global variables
  - stack accesses
  - accesses to aliased heap data
- predictor tracks store/load PCs, reproduces last sourcing store PC given load PC
Memory Dependence Speculation Example

Dependencies

True realized

Program order

Time

ld Y

st?Z

ld X

ld Z

st?X

ld Y

st?Z

ld X

ld Z

ld Y

st?X

ld Y

st?Z

ld X

ld Z

st X

ld Y

st?Z

ld X

ld Z

st X

ld Y

st?Z

ld X

ld Z

ld X

ld Z
Memory Renaming [Tyson/Austin97]

- design maxims:
  - Registers Good, Memory Bad
  - Stores/Loads Contribute Nothing to Program Results

- basic idea:
  - leverage dependence predictor to map memory communication onto register synchronization and communication infrastructure

- benefits:
  - accurate dependence info if predictor is accurate
  - early presentation of dependence predictions
  - fast communication through register infrastructure
• renamed dependence edges operate at bypass speed
• load/store address stream becomes “checker” stream
  – need only be high-B/W (if predictor performs well)
  – risky to remove memory accesses completely
speculative loads require recovery mechanism

enhancements muddy boundaries between dependence, address, and value prediction

- long lived edges reside in rename table as addresses
- semi-constants also promoted into rename table
Experimental Evaluation

- implemented on SimpleScalar 2.0 baseline
- dynamic scheduling timing simulation (sim-outorder)
  - 256 instruction RUU
  - aggressive front end
  - typical 2-level cache memory hierarchy
- aggressive memory renaming support
  - 4k entries in dependence predictor
  - 512 edge names, LRU allocated
- load speculation support
  - squash recovery
  - selective re-execution recovery
Dependence Predictor Performance

- good coverage of “in-flight” communication
- lots of room for improvement
Dependence Prediction Breakdown

Breakdown by Segment

- Global
- Stack
- Heap

Applications:
- CC1
- Comp
- Go
- M88ksim
- Perl
- Xlisp
- Hydro-2D
- Mgrid
- Su2Cor
- Tomcatv

Advanced Computer Architecture Lab
University of Michigan

Memory Scheduling
EECS 470
Program Performance

- performance predicated on:
  - high-B/W fetch mechanism
  - efficient mispeculation recovery mechanism

- better speedups with:
  - larger execution windows
  - increased store forward latency
  - confidence mechanism
Future Directions

• turning of the crank - continue to improve base mechanisms
  – predictors (loop carried dependencies, better stack/global prediction)
  – improve mispeculation recovery performance
• value-oriented memory hierarchy
• data value speculation
• compiler-based renaming (tagged stores and loads):

  store r1,(r2):t1
  store r3,(r4):t2
  load  r5,(r6):t1
Summary

• dynamic scheduling provides infrastructure to exploit instruction level parallelism
  – currently geared towards register dependencies
  – memory scheduler required to deal with memory deps
• many solutions to memory scheduling problem
  – conventional: in-order, blind, conservative dataflow
  – research: memory dependence speculation, renaming
• memory renaming
  – uses dependence prediction to overlay memory communication onto register infrastructure
  – decent performance, will improve with trends and work