Sequential Program

Physically Distributed Processing Units (e.g. CMP, Multiscalar, ...)

Xiaobin Li
xiaobinL@uci.edu
for (i=0; i<100; i++)
{
    A[i] += 10;
    if (A[i] > 1000)
        A[i] = 1000;
}

R1 = -1 ;initial induction variable i
R0 = b ;assign loop-invariable b(=10) to R0
L:  R1 = R1 +1 ;increment i
    R2 = mem[R1+A] ;load A[i]
    R2 = R2 + R0 ;add b
    BLT   R2, 1000, B ;branch to B if A[i] < 1000
    R2 = 1000 ;set A[i] to 1000
B:  mem[R1+A] = R2 ;store A[i]
    BLT   R1, 99,L ;branch to L if i<99
Speculative Multithreading

PU1

L: R1 = R1+1
   R2 = mem[R1+A]
   R2 = R2 + R0
   BLT R2, 1000, B
   R2 = 1000
B: mem[R1+A] = R2
   BLT R1, 99, L

Time

PU2

L: R1 = R1+1
   R2 = mem[R1+A]
   R2 = R2 + R0
   BLT R2, 1000, B
   R2 = 1000
B: mem[R1+A] = R2
   BLT R1, 99, L

PU3

L: R1 = R1+1
   R2 = mem[R1+A]
   R2 = R2 + R0
   BLT R2, 1000, B
   R2 = 1000
B: mem[R1+A] = R2
   BLT R1, 99, L
Inter-Thread Dependencies

L:  
R1 = R1+1  
R2 = mem[R1+A]  
R2 = R2 + R0  
BLT R2, 1000, B  
R2 = 1000  
B:  
mem[R1+A] = R2  
BLT R1, 99, L

L:  
R1 = R1+1  
R2 = mem[R1+A]  
R2 = R2 + R0  
BLT R2, 1000, B  
R2 = 1000  
B:  
mem[R1+A] = R2  
BLT R1, 99, L

L:  
R1 = R1+1  
R2 = mem[R1+A]  
R2 = R2 + R0  
BLT R2, 1000, B  
R2 = 1000  
B:  
mem[R1+A] = R2  
BLT R1, 99, L

Non-speculative thread  
Speculative thread 1  
Speculative thread 2

Inter-Thread Control Dependency  
Inter-Thread Register Dependency  
Inter-Thread Potential Memory Dependency
Proving Sequential Program Semantics

- Inter-thread dependencies
  - Control dependency
  - Register-level data dependency
  - Memory-level data dependency

- “Latest data version” requirement
About the Presentation

- Problem statement
- **Handling threads**
  - Register-Level
  - Memory-Level
  - Evaluation
  - Conclusion
Thread Identifying

Forming from a control-flow-graph (CFG)
Loop Iteration Threads

- “Currently, we limit the threads to loop iterations.”
- “At present, our current approach of analyzing sequential binaries is restricted to loop iterations.”

► Post-compiler step: “… …, we mark the entry and exit points of each loop.”

► At runtime: “… …, when a loop entry point is reached, multiple threads are spawned to begin execution of successive iterations speculatively.”

😊 Already found one example from previous discussion
Analysis Assumptions

- 4-processor CMP
- Each processor guarantees itself sequential program semantics $\rightarrow$ thread-level sequential semantics
- 4-processor configuration as circular queue (explicitly at Multiscalar)
- Sequential thread termination: until thread becomes non-speculative (for example, its predecessor resolved conditional branch so its execution approved and got non-speculative state), it can commit and new thread can initiate on same processor.
Thread Commits and Squashes

(a) Thread assigns  (b) Thread squashes  (c) Thread commits
About the Presentation

- Problem Statement
- Handling threads
- **Register-Level**
- Memory-Level
- Evaluation
- Conclusion
Register - Level

- SW+HW
- Requirements:
  - Communication mechanism
  - Data dependency
  - Latest data version
  - Last copy problem
  - Complexity
Register-Level SW Support (1)

- Looplive register identifying
  - Specify which registers will be communicated between loop iteration threads, (loop-carried) and, for a thread “Me”, am I a producer? A consumer?
  - Live at loop entry/exits and may also be redefined in the loop

- Early validation of prediction
Register-Level SW Support (2)

Safe definitions + release points
→ Guarantee “latest data version”

**EX.** For a given looplive register, say, R1 as shown at the figure, SW tells the time point “no more update it until loop exit”, which is time point (2) provided between (2) and (3) no update R1 instruction exist.

Send R1 after point (2), so that its successors get “latest version of R1”.
Register-Level SW Support (3)

Conditional branch fan-out; when and what value of R2; latest version problem

```
    R1 = R1 + 1
    R2 = R2 + 2
    BLT R1, 10, L_EXIT
    R2 = R2 + R1
L_EXIT:    ...  
```

Unsafe point
Safe point
Release point
Register-Level HW Support

1. Road – Synchronizing scoreboard bus (SS Bus)
2. Signal - Synchronizing scoreboard (SS)
3. Roadmap – Register-level Read/Write protocol
Synchronizing Scoreboard
Each register has an entry

- **S-bit (global):**
  - $S_i = 1 \rightarrow$ the thread running on $P_i$, will write this register, but not ready
  - $S_i = 0 \rightarrow$ data available

- **F-bit (global):**
  - $F_i = 1 \rightarrow$ the thread running on $P_i$ is a producer
  - $F_i = 0 \rightarrow$ not a producer

- **V-bit (local):**
  - $V = 1 \rightarrow$ data available
  - $V = 0 \rightarrow$ data not yet available
Consumer-Initiated Approach - Read

\[ \_ = R1 + 10 \]

\[ V = 1 \]

- **Y**: Read R1 locally
- **N**: Check F-bit to find out the closest predecessor who is the producer

- **Y**: Its S-bit = 1
  - **N**: Ask it for data

- **Y**: Wait for S-bit change to 0
Why the Closest Predecessor?
How to Find Out the Closest Predecessor?

“Me” Thread on P2 (ThreadMask = 1111)
→ It’s speculative thread #3
→ Its predecessors are speculative thread #2, speculative thread #1 and non-speculative thread.
→ The locations of those can be found out from ThreadMask associated with processors.
→ Read register R1 from P3
Consumer-Initiated Approach - Write

R1 = ... ...

“Latest version” write ?

N
Normal write to local register file and set V-bit correspondingly

Y
“Latest version” write to local register file and set V-bit, and S-bit
RAW Proving

Time

T

T+1

T+2

W

W

R

Only read from the closest predecessor, know where it is, and forced waiting for data if not available.
WAW Proving

Local available first

Only read from the closest predecessor, know where it is, and forced waiting for data if not available
WAR Proving
The Last Copy Problem

Non-Specu

Specu#1

Specu#2

Specu#3

T0

T1

T2

T3

T0, T1, T2 all committed

New threads T4, T5, T6 come
But T3 is about to commit

T3

T4

T5

T6

T3

T7
Sent (X) Bits

For each looplive register:
i = processor ID
\( Xi = 0 \) → when thread initials and hold until updated if any
\( Xi = 1 \) → sent out to Pi in the past or Pi asked for data before

When a thread ready to commit, checks X-bit.

**EX.** P0 checks, whose thread is non-speculative one:
if \( (X1(F1 + X2(F2 + X3)) == 0) \)
{
    re-sent data || wait for retrieving and then commit;
}


$X_1(F_1 + X_2(F_2 + X_3)) = 0$

- For $P_1$, $X_1 = 0$, not OK!

- For $P_2$, $F_1 = 0$ means $P_1$ is not a producer, so if $X_2 = 0$ at same time, not OK!

- For $P_3$, $F_1 = 0$ and $F_2 = 0$ means $P_1$ and $P_2$ are not producers, so if $X_3 = 0$ at same time, not OK!
Speculative Issues at Register-Level

- Only read latest version register
- No speculative read within a thread
- Thread is speculative and incorrect thread prediction
Register Reuse

- Once non-speculative thread committed, new thread may be initialized on the same processor. Since dealing with loop iteration thread, some register value can be reused, for performance.

**EX.**

```
R1 = -1 ;initial induction variable i
R0 = b ;assign loop-invariable b to R0
L:  R1 = R1 +1 ;increment i
   R2 = mem[R1+A] ;load A[i]
   R2 = R2 + R0 ;add b
   BLT R2, 1000, B ;branch to B if A[i] < 1000
   R2 = 1000 ;set A[i] to 1000
B:  mem[R1+A] = R2 ;store A[i]
   BLT R1, 99,L ;branch to L if i<99
```
Need Crossbar-like Connection between Register Files?

- Broadcast-like bus $\rightarrow$ Bus contention
- Register traffic analysis from the paper:
  M. Franklin and G. S. Sohi;, ”Register traffic analysis for streamlining inter-operation communication in fine-grain parallel processors”, MICRO-25, 1992
About the Presentation

- Problem Statement
- Handling threads
- Register-Level
- Memory-Level
- Evaluation
- Conclusion
Memory-Level

- HW
- Memory disambiguation
- Load/Store operations
- “Latest version” vs. “many-write”
- RAW/WAR/WAW proving
- Speculative issues
Potential Memory Dependency

- Without address calculation, we don’t know if two memory accesses go to the same location. So don’t know if any memory-level data dependency.

- Memory disambiguation: “the process of determining if two memory referencing instructions access the same memory location.”
L1-Cache Modification

For each cached word (not cache line!)

- **SW-bit:**
  - $SW = 1 \rightarrow$ OK write a word without informing MDT
  - $SW = 0 \rightarrow$ Write need inform MDT

- **SR-bit:**
  - $SR = 1 \rightarrow$ OK read a word without informing MDT
  - $SR = 0 \rightarrow$ Read need inform MDT
Memory Disambiguation Table (MDT)
### Entry of MDT

<table>
<thead>
<tr>
<th>V</th>
<th>Address Tag</th>
<th>Load-Bit</th>
<th>Store-Bit</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>L0L1L2L3</td>
<td>S0S1S2S3</td>
</tr>
</tbody>
</table>

\( i = \) processor ID

\( \text{Li} = 1 \) → thread on Pi read this word before

\( \text{Si} = 1 \) → thread on Pi write this word before
How SW-bit and SR-bit modified?

Thread initial
SW = 0

First time write

D=1

SW = 1

Successors read
SW = 0

Thread initial
SR = 0

First time read

SR = 1

Maybe violate RAW, if that case, thread squashed!
Memory Read Operation

= mem[R1+100]

Local L1-Cache SR = 1?  
Y  Read from local L1-cache

N

Check with MDT, got the closest predecessor whose S-bit is set

No predecessors’ S-bit set

Read from that predecessor’s L1-cache

Read from L2-cache
Memory Write Operation

- Non-speculative thread
  - L1-cache write-through
    - all operation check with MDT

- Speculative threads
  - L1-cache write-back
    - (miss in L1-cache) or (hit but SW = 0)
      - check with MDT
Unidirectional check-on-store

Mem[R1+100] = informing MDT

Did "Me" successors read before? N → For "Me", safe! Y →

Intervening threads are threads after "Me" but before checked thread (included)

Is there any intervening thread* wrote before? Y → Squash & restart speculative threads* N →

Checked Speculative-read thread and all its successors
Is SafeWrite-bit safe?

“Many-write” and “latest version”

How about “latest-version” problem?
Require RAW2 not only RAW1 is correct!
(SW = 1, after W1.)

After R, MDT notify thread
“T”: clear SW-bit (1->0).
When W2, check-on-store,
detect RAW violation! And hold “latest version”.
Is SafeRead-bit Safe (1)?

Case 1: If a write comes in between "Read1" and "Read2".

Case 2: If a write comes in after "Read2", "read1" left a record on MDT.

Case 3: Not case 2 & case 3. "speculative load"
Is SafeRead-bit Safe (2)?

- To exploit spatial locality, load a cache line while loading, so have the case “hits a word with SR = 0”, recall SR-bit is word-based.

- When write, “invalidation message is sent to all successors up to, but not including, the one whose S bit is set.”
  \[\rightarrow \text{“hits a word with SR = 0” becomes “misses a word with SR = 0”}.\]
RAW Proving

At time point of R, fine!

No Write between.

At time point of W, check-on-store detect T2’s R violated RAW, so T2 and T3 squash.
WAW Proving

Time

W

W

T0 (R) W (R)

T1

R

T2 (R) W (R)

T3

Read from the closest predecessor if available! If not, go to L2-cache.

T2 is speculative, its L1-cache works on write-back.
WAR Proving

T0

T1
(R)

T2

W

T3

Read from the closest predecessor if available, if not, go to L2-cache!

T2 is speculative, L1-cache works on write-back, not commit just hold data.
Speculative Issues at Memory-Level

- **Thread Initial**
  All words whose F-bit is set are invalidated; then F, SW, SR bits are cleared.

- **Thread Squash**
  - Speculative load data from memory
  - Thread is speculative and incorrect prediction
  When a thread is squashed, some additional words need to be invalidated:
    - those are dirty;
    - those were forward from a squashed predecessor thread, with the help from FD-bit.
**Alternative Approach – Speculative Versioning Cache (SVC)**

<table>
<thead>
<tr>
<th>Operation</th>
<th>Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load</td>
<td>Record use before; supply the closest previous version</td>
</tr>
<tr>
<td>Store</td>
<td>Communicate store to later task; later tasks look for memory dependence violations</td>
</tr>
<tr>
<td>Commit</td>
<td>Writeback buffered versions created by the task to main memory</td>
</tr>
<tr>
<td>Squash</td>
<td>Invalidate buffered versions created by the task</td>
</tr>
</tbody>
</table>
Base SVC Design

[Diagram showing a bus structure with P and SV$ symbols, a bus arbiter, version control logic, states of snooped lines, and VCL responses to each cache.]
SVC Load
SVC Store
About the Presentation

- Problem Statement
- Handling threads
- Register-Level
- Memory-Level
- Evaluation
- Conclusion
Evaluation Platform

- 4-processor CMP, each core processor is modeled as MIPS R10000
- Compare with:

<table>
<thead>
<tr>
<th>Issue Width</th>
<th># of F. U. (int/ld-st/fp)</th>
<th>Entries in Instr. Window</th>
<th># of Renaming Reg. (int/fp)</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>4/2/2</td>
<td>64</td>
<td>64/64</td>
</tr>
<tr>
<td>12</td>
<td>12/6/6</td>
<td>200</td>
<td>200/200</td>
</tr>
</tbody>
</table>
## Memory Hierarchy Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>CMP</th>
<th>4-Issue Superscalar</th>
<th>12-Issue Superscalar</th>
</tr>
</thead>
<tbody>
<tr>
<td>[L1,L2] size (KB)</td>
<td>[4X16,1024]</td>
<td>[64,1024]</td>
<td>[64,1024]</td>
</tr>
<tr>
<td>[L1,L2] line size (B)</td>
<td>[32,64]</td>
<td>[32,64]</td>
<td>[32,64]</td>
</tr>
<tr>
<td>[L1,L2] associativity</td>
<td>[2,4]</td>
<td>[2,4]</td>
<td>[2,4]</td>
</tr>
<tr>
<td>L1 banks</td>
<td>3</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>L1 latency (cycle)</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>L2 latency (cycle)</td>
<td>6</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>Mem. Latency (cycle)</td>
<td>26</td>
<td>26</td>
<td>26</td>
</tr>
</tbody>
</table>
Performance Evaluation
SS Bus BW Evaluation

![Graphs showing execution time and latency vs bandwidth and latency of the SS bus]
MDT Evaluation

- Left graph: Execution time vs. round-trip latency from L1 to MDT (Cycles).
- Right graph: Execution time vs. number of MDT entries.
About the Presentation

- Problem Statement
- Handling threads
- Register-Level
- Memory-Level
- Evaluation
- Conclusion
Conclusion

- Fundamental requirement - sequential program semantics
- Complicity variation between register-level and memory-level – crack differently
- Holding sequential program semantics, go to high-performance – more thread identifying

- Helpful on-line papers available:
  1. Dr. Sohi’s Multiscalar: [http://www.cs.wisc.edu/~mscalar](http://www.cs.wisc.edu/~mscalar)
  3. IEEE digital library