Lecture 29(b)

Compiler Optimizations for Thread-Level Speculation

Speculation

\( T_i \)

store \(*p\)

\( T_{i+1} \)

load \(*q\)

Memory

\( \text{good when } p \neq q \)
Synchronization (and Forwarding)

(Ti) store *p
Memory

wait (stall)
load *q

(Ti+1) load *q

Memory

Signal

store *p

(Speculation)

good when p == q
Synchronizing Scalars

\[ T_i \]

\[ \ldots = a \]

\[ a = \ldots \]

\[ \text{signal}(a) \]

\[ T_{i+1} \]

\[ \text{wait}(a) \]

\[ \ldots = a \]

\[ a = \ldots \]

\[ \text{signal}(a) \]

wait(a);

\[ a > 0 \]

\[ a = a \times 7 \]

\[ \text{signal}(a) \]
Compiler's Tasks

Identifying Forwarding Scalar

... = a
a = ...;

Inserting Wait/Signal

Scheduling Instructions

Scheduling Instructions Speculatively
Compiler’s Tasks

Identifying Forwarding Scalar

Inserting Wait/Signal

Scheduling Instructions Speculatively

wait(a);
... = a;

a = ...

signal(a);

a = ...

... = a;
Cost of Synchronization

Detailed simulation:
- 4-processor
- Single chip multi-processor
- TLS support

Reducing the synchronization stall has great potential
The Critical Forwarding Path

Long Critical Path

Short Critical Path

**Shorter critical forwarding path** \( \rightarrow \) **less execution time**
Reducing the Critical Forwarding Path

Long Critical Path

Short Critical Path

Instruction scheduling can reduce critical forwarding path
Compiler's Tasks

Identifying

Forwarding Scalar

wait(a);
... = a

Inserting
Wait/Signal

wait(a);
... = a

a = ...
signal(a);

Scheduling Instructions

Scheduling Instructions Speculatively
Related Work and Contribution

Related work:
- Multiscalar instruction scheduling [Vijaykumar, Thesis ‘98]
  - Moving instructions backward one basic block at a time
  - Evaluated under the context of Multiscalar

Our contributions:
- A robust instruction scheduling algorithm
  - Deals with larger threads
  - Handles complex control flow
- Control and data dependence speculation
  - Extends our algorithm to accommodate speculative scheduling
  - Evaluates with detailed simulation
- Comparison with hardware techniques that reduce critical path
Scheduling Instructions

Dataflow analysis
Handles complex control flow

Define two dataflow analyses

Stack
Find the instructions to compute the forwarded value?

Earliest
Find the earliest node to compute the forwarded value?
Computation Stack

Stores the instructions to compute a forwarded value

Associating a stack with every node for every forwarded scalar

\[ a = a*11 \]

We know how to compute the forwarded value

We don’t know how to compute the forwarded value

We have not evaluated this node
A Simplified Example from GCC

```
do {
    ...
} while(p);
```
Stack Analysis

start

wait(p)
counter=0
p->jmp?

p=p->jmp

q = p
p->real?

q = p
p->next
q?

p=p->next

counter++;
q=q->next;
q?

p = p->next

signal p

end
Stack Analysis

start

wait(p) counter=0 p->jmp?

p=p->jmp

q = p
p->real?

p=p->next

q=q->next; q?

counter++;

p = p->next

signal p

signal p

end
Stack Analysis

start

wait(p)
counter=0
p->jmp?

p=p->jmp

q = p
p->real?

q=q->next;
q?

p=p->next

signal p

end

Carnegie Mellon

15-745: Optimizing TLS

Todd C. Mowry
Stack Analysis

start

wait(p)
counter=0

p=p->jmp

jmp?
signal p

p=p->next

q = p->real?
t

p=p->next

t = p->next

signal p

counter++;q=q->next; q?

p = p->next

p=p->next

signal p

done

end
Stack Analysis

start

wait(p)
counter=0
p->jmp?

p=p->jmp

q = p
p->real?

p=p->next
q=q->next;q?

counter++;q=q->next;q?

p=p->next

signal p

p=p->next

signal p

d = d->next

end

signal p

p=p->next

signal p
Stack Analysis

```
start

wait(p)
counter=0
p->jmp?

p=p->jmp

q = p
p->real?

p=p->next

signal p

end
```

```
p=p->next

q=q->next; q?

p = p->next

signal p
```

Carnegie Mellon

15-745: Optimizing TLS
Stack Analysis

start

wait(p)
counter=0
p->jmp?
p=p->jmp

q=p
p->real?
q=p->next
q?
signal p
default

p=p->next
signal p

p=p->next
signal p

p=p->next
signal p

p=p->next
signal p

end

15-745: Optimizing TLS

Carnegie Mellon

Todd C. Mowry
Stack Analysis

```
start
  wait(p)
  counter=0
  p->jmp?
    p=p->jmp
    q = p
    p=p->real?
      q=q->next
      q?
      counter++;
      p=p->next
      signal p
      p=p->next
      signal p
    end
  p=p->next
  signal p
  p=p->next
  signal p
  p=p->next
  signal p
  p=p->next
  signal p
  p=p->next
  signal p
```
Stack Analysis

start

wait(p)
counter=0
p->jmp?

p=p->jmp

p=p->next
signal p

q = p
p->real?

p=p->next
signal p

p=p->next
signal p

p=p->next
signal p

counter++;q=q->next;
q?

p = p->next

p=p->next
signal p

end

15-745: Optimizing TLS
Stack Analysis

start

wait(p) counter=0
p->jmp?

p=p->jmp

signal p

p=p->next

q=p->real?

p=p->next

q=q->next; q?

counter++;

signal p

p=p->next

signal p

p=p->next

signal p

p=p->next

signal p

end

15-745: Optimizing TLS

26

Todd C. Mowry
The Solution is Consistent

```c
start

p = p->next
signal p

p = p->next
signal p

p = p->next
signal p

counter++;
q = q->next;
q?

p = p->next

p = p->next
signal p

p = p->next
signal p

p = p->next
signal p

end
```

Todd C. Mowry
Carnegie Mellon

15-745: Optimizing TLS
Scheduling Instructions

Dataflow analysis
Handles complex control flow

Define two dataflow analyses

Stack
Find the instructions to compute the forwarded value?

Earliest
Find the earliest node to compute the forwarded value?
The Earliest Analysis

start

wait(p)
counter=0
p->jmp?

p->jmp

signal p

p->next

q = p
p->real?

p->next

signal p

p->next

signal p

q = q->next;
q?

p = p->next

signal p

end

Earliest
Not earliest
Not evaluated

Carnegie Mellon

15-745: Optimizing TLS

Todd C. Mowry
The Earliest Analysis

1. $p = p->jmp$
2. $p = p->next$
3. signal $p$
4. $p = p->jmp$?
   - Yes: $p = p->jmp$
   - No: $p = p->next$
5. $q = p$
6. $p->real?$
   - Yes: $q = q->next$; $q$?
   - No: $p = p->next$
7. signal $p$
8. Signal $p$
9. $p = p->next$
10. $p = p->next$
11. $p = p->next$
12. $p = p->next$
13. $p = p->next$
14. end

Earliest
Not earliest
Not evaluated
The Earliest Analysis

1. start
2. wait(p)
counter = 0
p->jmp?
3. p->jmp
4. signal p
5. p->next

6. q = p
7. p->real?
8. signal p
9. p->next

10. counter++;
11. q = q->next;
12. q?
13. p = p->next
14. signal p
15. p->next

16. p->next
17. signal p
18. end

Earliest
Not earliest
Not evaluated
Code Transformation

start

wait(p)
counter=0
p->jmp?

p2=p->jmp
p1=p->next
Signal(p1)

q = p
p->real?

p=p->next

p1=p->next
Signal(p1)

counter++;
q=q->next;
q?

end
Experimental Framework

Benchmarks
- from SPECint95 and SPECint2000, -O3 optimization

Underlying architecture
- 4-processor, single-chip multiprocessor
- speculation supported by coherence

Simulator
- superscalar, similar to MIPS R10K
- models all bandwidth and contention

☞ detailed simulation!
Instruction Scheduling

U=No Instruction Scheduling
A=Instruction Scheduling

Improves performance by 18% over no instruction scheduling
Compiler's Tasks

- Identifying Forwarding Scalar
- Inserting Wait/Signal
- Scheduling Instructions
- Scheduling Instructions Speculatively
Speculating Beyond a Control Dependence

\[ p = p \rightarrow \text{jmp} \]

\[ \text{signal } p \]

\[ p = p \rightarrow \text{jmp} \]

\[ p = p \rightarrow \text{next} \]

\[ \text{end} \]

\[ \text{violate}(p) \]

\[ pl = p \rightarrow \text{jmp} \]

\[ \text{signal}(pl) \]

\[ \text{signal } p \]

\[ \text{end} \]
Speculating Beyond a Potential Data Dependence

```
p=load(addr1);
q->next = NULL
store(addr2);
p=p->next
signal p
p = p->next
signal p
end
```

- **Hardware Support Needed**

Todd C. Mowry  
Carnegie Mellon
Speculatively Scheduling Instructions Across Control Dependences

A=Instruction Scheduling
C=Speculating Across Control Dependences

No significant performance gain

Todd C. Mowry
Carnegie Mellon

15-745: Optimizing TLS
Aggressively Scheduling Instructions Across Both Control and Data Dependences

Improves performance significantly for some benchmarks

A=Instruction Scheduling
C=Speculating Across Control Dependences
D=Speculating Across Control & Data Dependences

Todd C. Mowry
Carnegie Mellon

15-745: Optimizing TLS
Hardware Optimization to Reduce Synchronization

Hardware optimization techniques [Steffan et al, HPCA’02]

- Avoid synchronization:
  - use the value from a hardware value predictor

- Reduce synchronization stalls:
  - prioritize computation of forwarded value

Hardware optimization impact [Steffan et al, HPCA’02]

- No compiler optimization: Effective
- With compiler optimization: Negligible
Conclusions

Instruction scheduling for reducing synchronization

- Is effective in reducing critical forwarding path
  - Performance improved by 18%
- Is beneficial to handle complex control flow, such as inner loops
  - Improved GCC by 3%
- Gives additional benefit with speculative instruction scheduling
  - Our robust instruction scheduling algorithm can be easily extended to accommodate this
  - One biggest benefactor is GCC, performance improved by 18%
- Reduces the importance of additional hardware optimization

💰 Critical forwarding path can be addressed by the compiler