Lecture 29(b)

Compiler Optimizations for Thread-Level Speculation


Speculation

\[ T_i \quad T_{i+1} \]

\[ \text{store } *p \quad \text{load } *q \]

Memory

\[ \text{good when } p \neq q \]

Synchronization (and Forwarding)

\[ T_i \quad T_{i+1} \]

\[ \text{store } *p \quad \text{signal} \]

\[ \text{wait (stall)} \quad \text{load } *q \]

Memory

\[ \text{good when } p = q \]

Synchronizing Scalars

\[ T_i \quad T_{i+1} \]

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

\[ a = \ldots \quad a = \ldots \]

\[ \text{good when } p = q \]
Compiler's Tasks

- Identifying Forwarding Scalar
- Inserting Wait/Signal
- Scheduling Instructions
- Scheduling Instructions Speculatively

Cost of Synchronization

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

Reducing the synchronization stall has great potential

The Critical Forwarding Path

Shorter critical forwarding path → 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

Inserting Wait/Signal

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

\[ a = a \times 11 \]

We know how to compute the forwarded value

We don't know how to compute the forwarded value

We have not evaluated this node
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

Earliest
Not earliest
Not evaluated

The Earliest Analysis

Earliest
Not earliest
Not evaluated

The Earliest Analysis

Earliest
Not earliest
Not evaluated

The Earliest Analysis

Earliest
Not earliest
Not evaluated
The Earliest Analysis

```
p = p->next
p = p->next
p = p->jmp
wait(p)
counter = 0
```

Earliest
Not earliest
Not evaluated

Code Transformation

```
p = p->next
signal p
```

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

- Sync
- Failed Viol
- Other
- Busy

Normalized Execution Time

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

Speculating Beyond a Potential Data Dependence

Hardware Support Needed
Speculatively Scheduling Instructions Across Control Dependences

- Sync
- Failed Viol
- Other
- Busy

A=Instruction Scheduling
C=Speculating Across Control Dependences

No significant performance gain

Aggressively Scheduling Instructions Across Both Control and Data Dependences

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

Improves performance significantly for some benchmarks

Hardware Optimization to Reduce Synchronization

- 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