Lecture 28(b)

Compiler Optimizations for Thread-Level Speculation


Speculation

\[ T_i \]
store *p
\[ T_{i+1} \]
load *q

Memory

\[ \Phi \]
good when \( p \neq q \)

Synchronization (and Forwarding)

\[ T_i \]
store *p
\[ T_{i+1} \]
load *q

Memory

(\[ \Phi \]
good when \( p = q \))

Synchronizing Scalars

\[ T_i \]
\[ T_{i+1} \]
wait(a)
as…
signal(a)

\[ \Phi \]
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 \(\rightarrow\) less execution time
Reducing the Critical Forwarding Path

Instruction scheduling can reduce critical forwarding path

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

Compiler's Tasks

Identifying Forwarding Scalar
Inserting Wait/Signal
Scheduling Instructions Speculatively

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

\[
\text{a = a*11}
\]

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**

\[
\text{do }
\]

\[
\text{\texttt{\ldots}}
\]

\[
\text{\texttt{while (p)}}
\]

---

**Stack Analysis**
Stack Analysis

1. `wait(p)`
2. `counter = 0`
3. `p->jmp`?
4. `p = p->jmp`
5. `q = p->real?`
6. `counter++`
7. `q > next`
8. `signal p`
9. `p = p->next`
10. `p = p->next`
11. `q = q > next`?
12. `p = p->next`
13. `signal p`
14. `signal p`
15. `signal p`

15-745: Optimizing TLS
Stack Analysis

The Solution is Consistent
**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

- Early
- Not earliest
- Not evaluated

Earliest
Not earliest

Code Transformation

- Early
- Not earliest

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 Visit
- Other
- Busy

Normalized Execution Time

U = No Instruction Scheduling
A = Instruction Scheduling

Improves performance by 18% over no instruction scheduling

Compiler’s Tasks

1. Identifying Forwarding Scalar
2. Inserting Wait/Signal
3. Scheduling Instructions Speculatively

Speculating Beyond a Control Dependence

Speculating Beyond a Potential Data Dependence

Hardware Support Needed
Speculatively Scheduling Instructions
Across Control Dependences

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

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