Motivation

- Industry is delivering multithreaded processors

IBM Power4 processor:
  - 2 processor cores per die
  - 4 dies per module
  - $\rightarrow$ 8 64-bit processors per unit

- Improving throughput is straight forward

How can we use multithreaded processors to improve the performance of a single application?

We need parallel programs

Automatic Parallelization

- Finding independent threads from integer programs is limited by
  - Complex control flow
  - Ambiguous data dependences
  - Runtime inputs
- Fundamental problem:
  - Parallelization is determined at compile time

Thread-Level Speculation

Example

```
while (...) {
  x = hash[index1];
  ... hash[index2] = y;
  ...
}
```

```
while (...) {
  x = hash[index1];
  ... hash[index2] = y;
  ...
}
```
Speculation

\[ T_i \]
\[ \text{store } *p \]
\[ \text{load } *q \]
\[ \text{Memory} \]

\( \text{good when } p \neq q \)

Synchronization (and Forwarding)

\[ T_i \]
\[ \text{store } *p \]
\[ \text{load } *q \]
\[ \text{Memory} \]

(Speculation)

\[ T_{i+1} \]
\[ \text{signal} \]
\[ \text{wait(} \text{stall} \text{)} \]

\( \text{good when } p = q \)

Synchronizing Scalars

\[ T_i \]
\[ \ldots = a \]
\[ a = \ldots \]
\[ \text{wait(a)} \]
\[ \text{signal(a)} \]

Compiler's Tasks

Identifying Forwarding Scalar

Inserting Wait/Signal

Scheduling Instructions

Scheduling Instructions Speculatively
Compiler's Tasks

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

Cost of Synchronization

- 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
Compiler’s Tasks

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

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?
  - We know how to compute the forwarded value
  - We don't know how to compute the forwarded value
  - We have not evaluated this node

- Earliest
  - Find the earliest node to compute the forwarded value?
A Simplified Example from GCC

do {
    ...
} while(p);

Stack Analysis

Stack Analysis

Stack Analysis

Stack Analysis
Stack Analysis

1. `start`  
   - `p=p->jmp`  
   - `p=p->next`  
   - `counter++;q=q->next`;  
   - `q?`  
   - `signal p`  
   - `p=p->next`  
   - `signal p`  
   - `wait(p)`  
   - `counter=0`  
   - `p->jmp?`  
   - `signal p`  
   - `p=p->next`  
   - `signal p`  
   - `p=p->jmp`

2. `start`  
   - `p=p->jmp`  
   - `p=p->next`  
   - `counter++;q=q->next`;  
   - `q?`  
   - `signal p`  
   - `p=p->next`  
   - `signal p`  
   - `wait(p)`  
   - `counter=0`  
   - `p->jmp?`  
   - `signal p`  
   - `p=p->next`  
   - `signal p`  
   - `p=p->jmp`
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

```
start
p = p->jmp
q = p
p->real?
```

```
end
```

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

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

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

```
signal p
```

```
p = p->jmp
```

```
wait(p)
```

### Code Transformation

```
start
wait(p)
q = p
p->real?
```

```
end
```

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

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

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

```
signal p
```

```
p = p->jmp
```

```
wait(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


### Instruction Scheduling

- Improves performance by 18% over no instruction scheduling

```
U = No Instruction Scheduling
A = Instruction Scheduling
```
Benefits from Global Analysis

- Multiscalar instruction scheduling [Vijaykumar, Thesis '98]
  - Uses local analysis to schedule instructions across basic blocks
  - Does not allow scheduling of instructions across inner loops

Instruction Scheduling

Improves performance by 18% over no instruction scheduling

Compiler’s Tasks

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

Speculating Beyond a Control Dependence
Speculating Beyond a Potential Data Dependence

```c
p = load(addr1); signal(p);
store(addr2);
```

Hardware Support Needed

Speculatively Scheduling Instructions Across Control Dependences

A=Instruction Scheduling
C=Speculating Across Control 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

\[ \text{Critical forwarding path can be addressed by the compiler} \]