Lecture 24

Thread-Level Speculation

Reference: *"Compiler Optimization of Scalar Value Communication Between Speculative Threads"*, by Antonia Zhai, Christopher B. Colohan, J. Gregory Steffan and Todd C. Mowry. ASPLOS, 2002.

Automatic Parallelization

Proving independence of threads is hard:

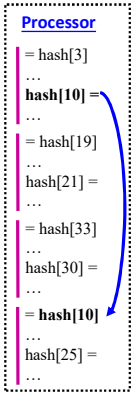
- complex control flow
- complex data structures
- pointers, pointers, pointers
- run-time inputs

How can we make the compiler's job feasible?

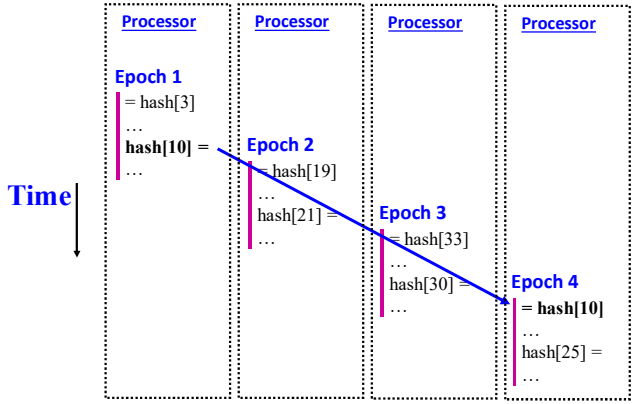
Thread-Level Speculation (TLS)

Example

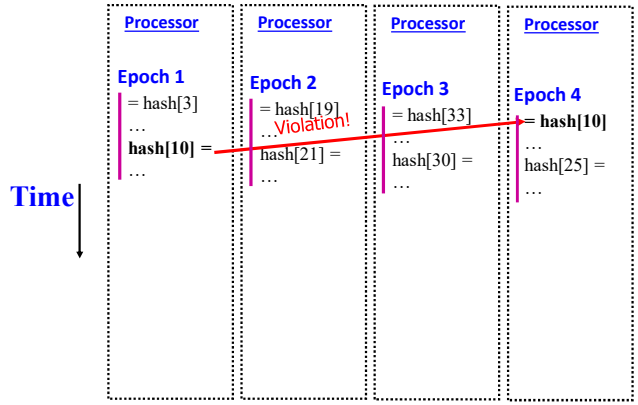
```
while (...){
    x = hash[index1];
    …
    hash[index2] = y;
    …
}
```



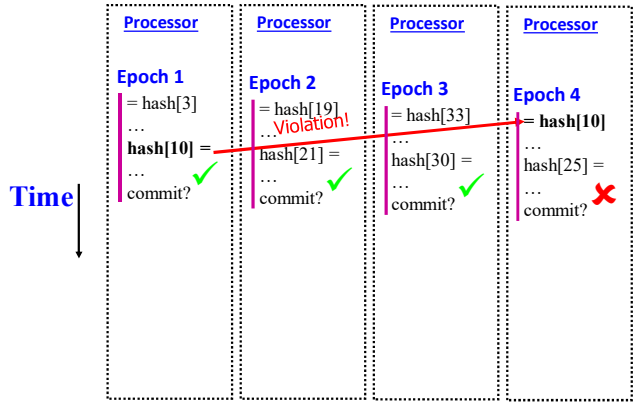
Example of Thread-Level Speculation



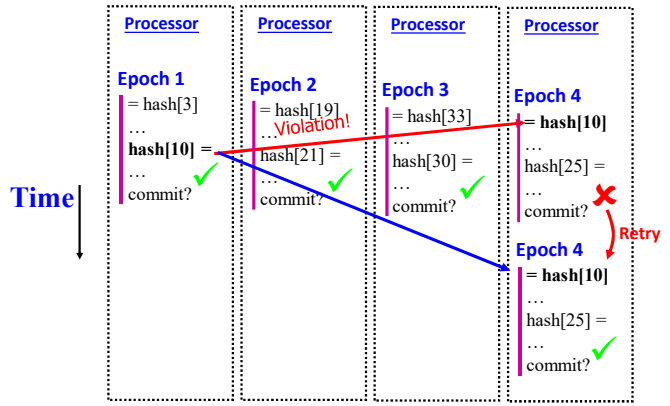
Example of Thread-Level Speculation



Example of Thread-Level Speculation



Example of Thread-Level Speculation



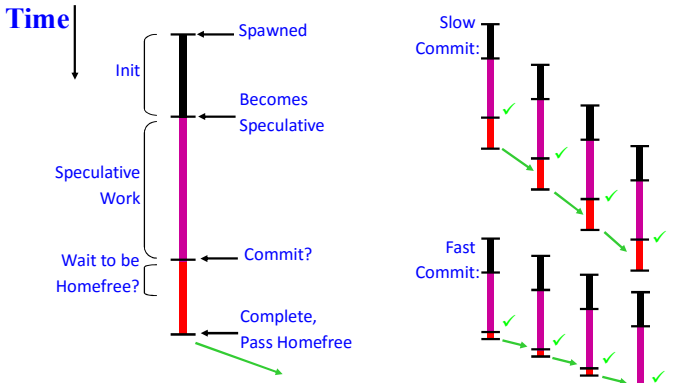
System Requirements for Thread-Level Speculation

System requirements:

1) Detect data dependence violations
   - extend invalidation-based cache coherence

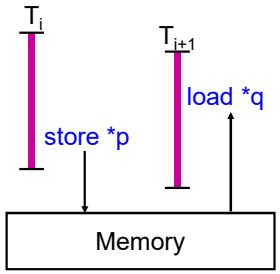
2) Buffer speculative modifications
   - use the caches as speculative buffers

Life Cycle of an Epoch



☞ speed-up depends on commit times

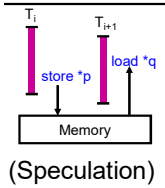
Another Limitation on Speed-up: Possible Dependences



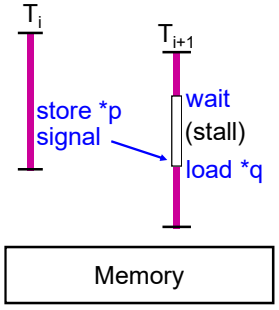
☞ good when p != q

☞ but when p == q, will cause violation

Solution: Add Synchronization and Value Forwarding

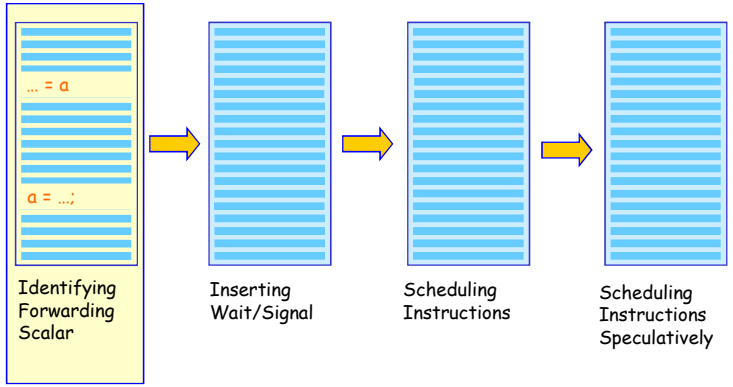


(Speculation)

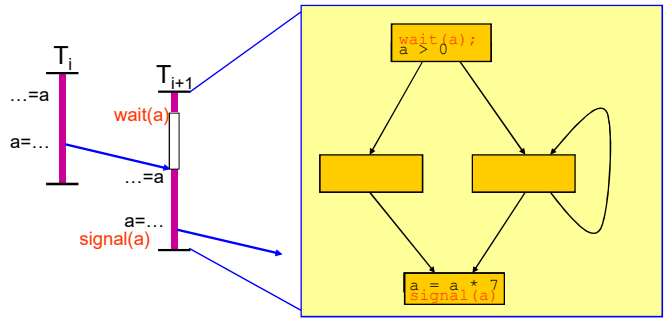


☞ good when p == q

Outline: Compiler Support for TLS



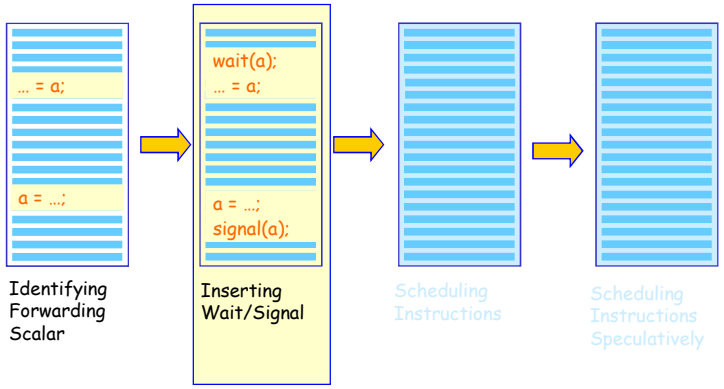
Synchronizing Scalars



Carnegie Mellon

13

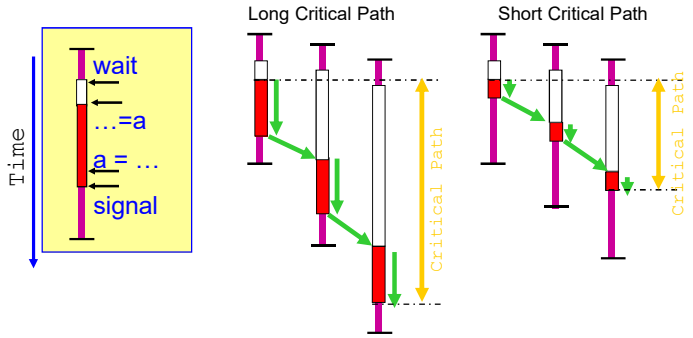
Compiler's Tasks



Carnegie Mellon

14

The Critical Forwarding Path

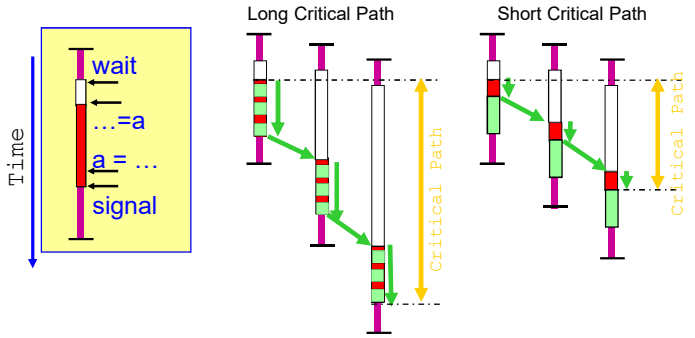


☞ Shorter critical forwarding path → less execution time

Carnegie Mellon

15

Reducing the Critical Forwarding Path

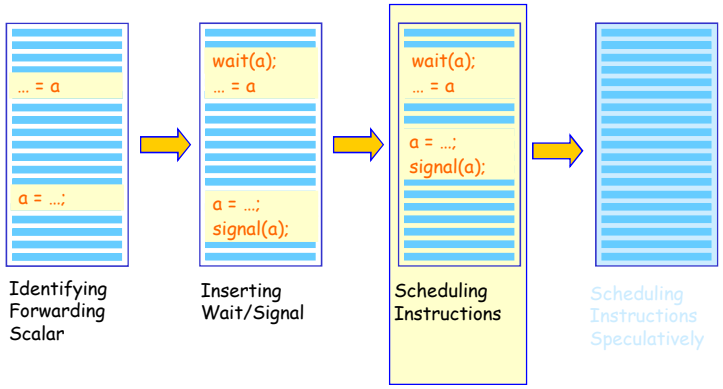


☞ Instruction scheduling can reduce critical forwarding path

Carnegie Mellon

16

Compiler's Tasks



Scheduling Instructions

Dataflow analysis

Handles complex control flow

Define two dataflow analyses

☞ Stack

Find the instructions to compute the forwarded value?

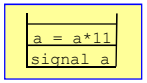
Earliest

Find the earliest node to compute the forwarded value?

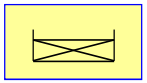
Computation Stack

☞ Stores the instructions to compute a forwarded value

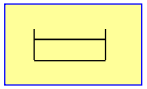
Associating a stack with every node for every forwarded scalar



We know how to compute the forwarded value

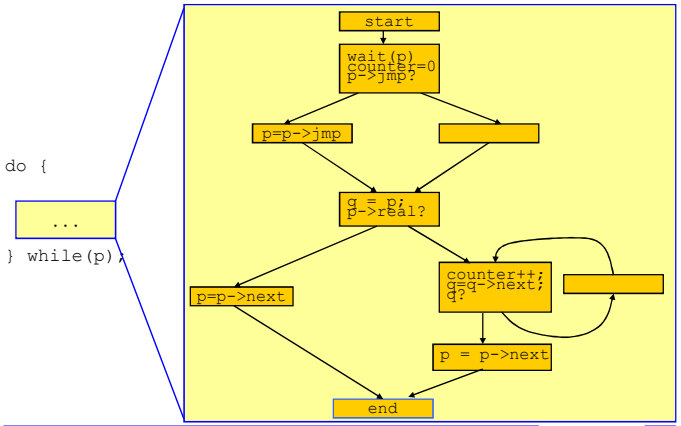


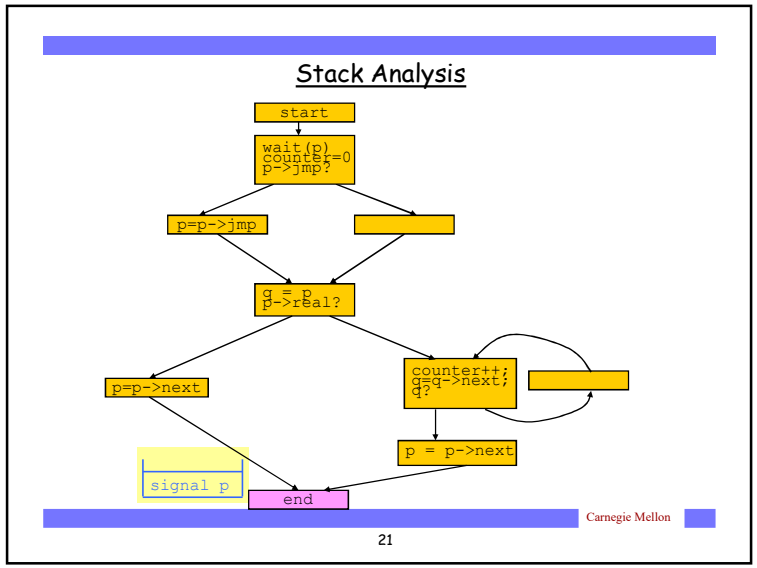
We don't know how to compute the forwarded value

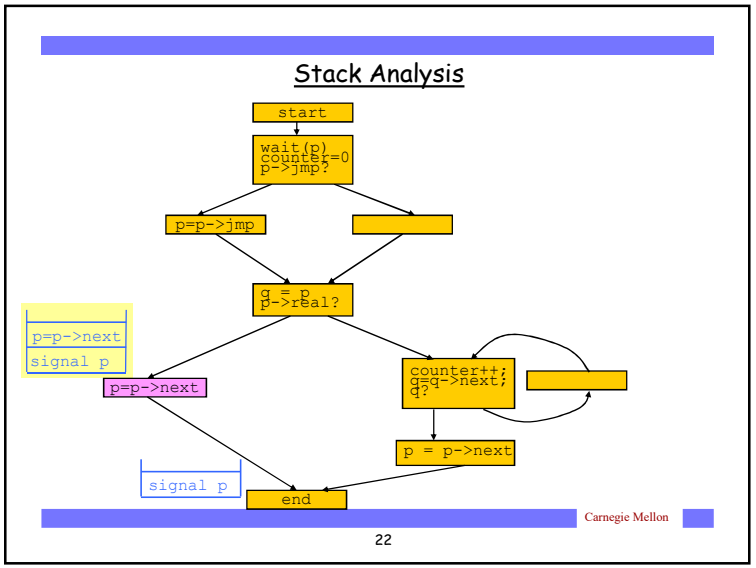


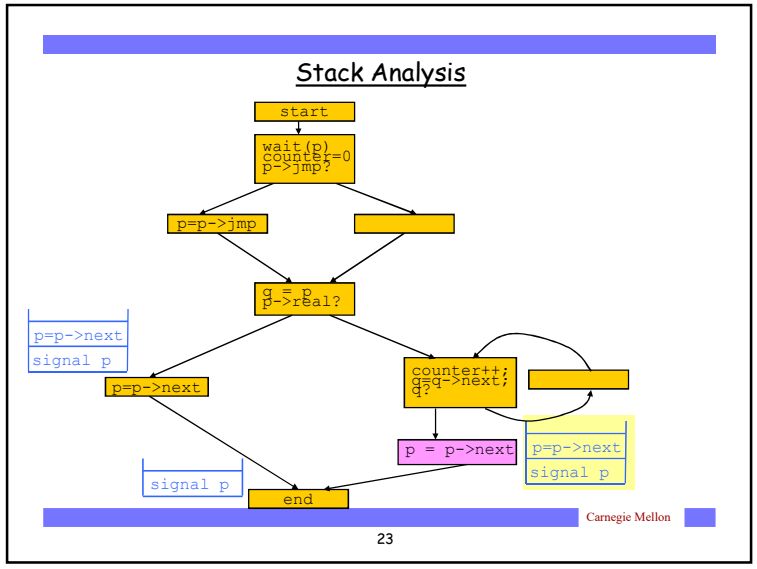
We have not evaluated this node

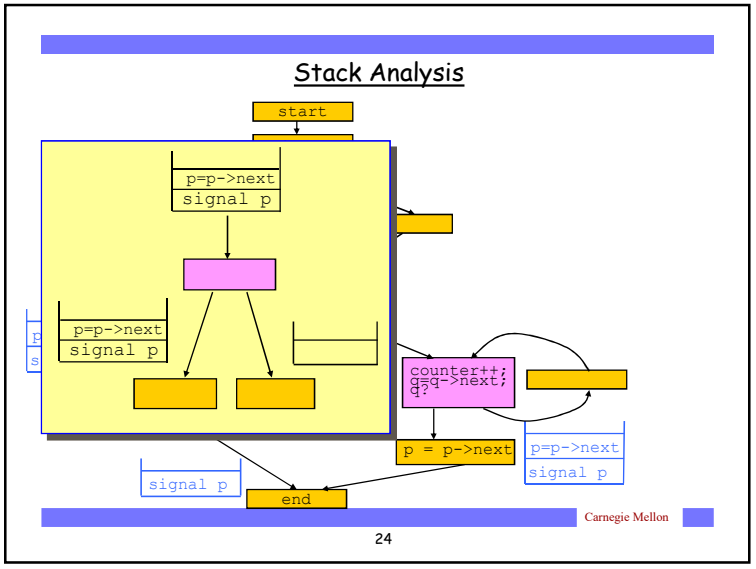
A Simplified Example from GCC

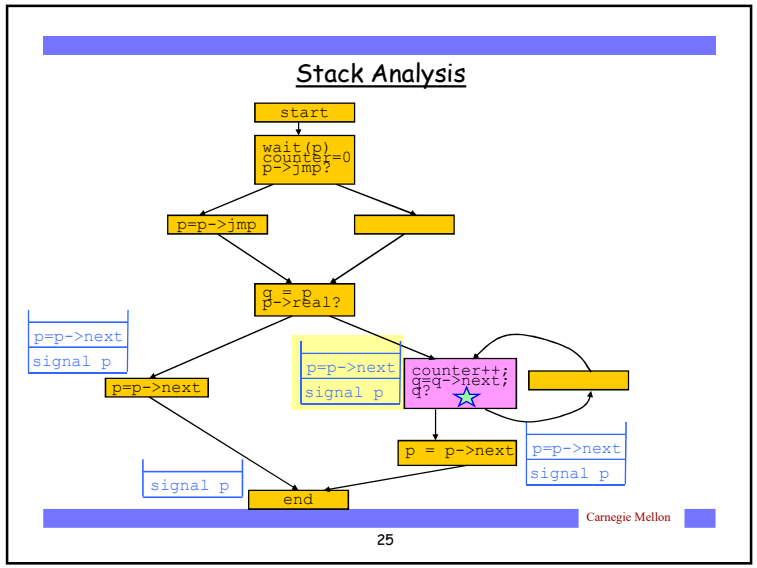


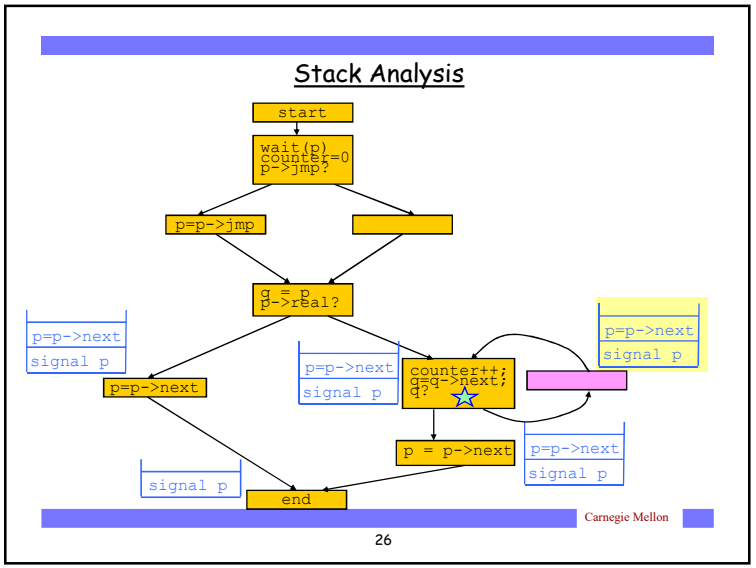

Stack Analysis
start
wait(p)
counter=0
p->jmp?
p=p->jmp
q = p
p->real?
p=p->next
counter++;
q=q->next;
q?
p = p->next
signal p
end
Carnegie Mellon
21

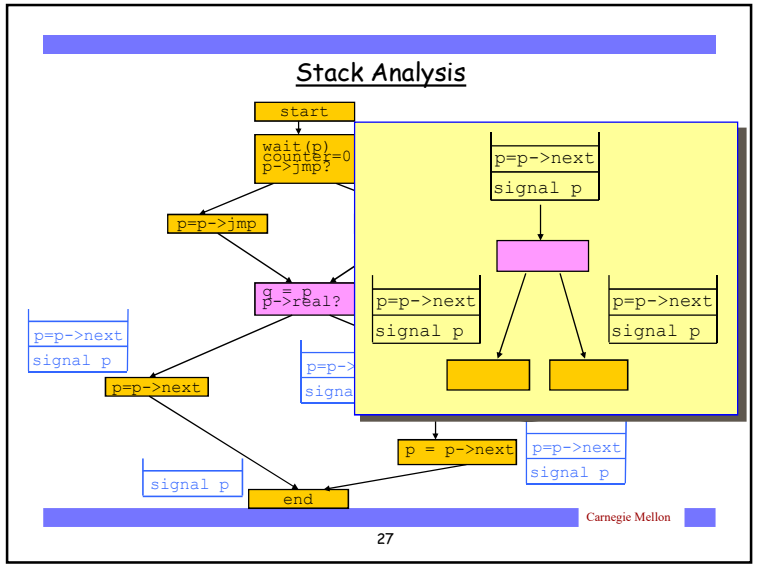

Stack Analysis
start
wait(p)
counter=0
p->jmp?
p=p->jmp
q = p
p->real?
p=p->next
signal p
p=p->next
counter++;
q=q->next;
q?
p = p->next
signal p
end
Carnegie Mellon
22

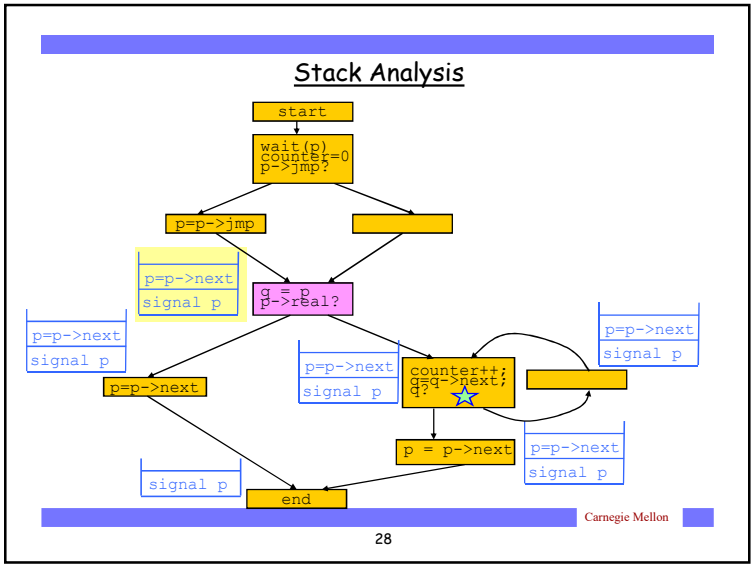

Stack Analysis
start
wait(p)
counter=0
p->jmp?
p=p->jmp
q = p
p->real?
p=p->next
signal p
p=p->next
counter++;
q=q->next;
q?
p = p->next
signal p
p=p->next
signal p
end
Carnegie Mellon
23

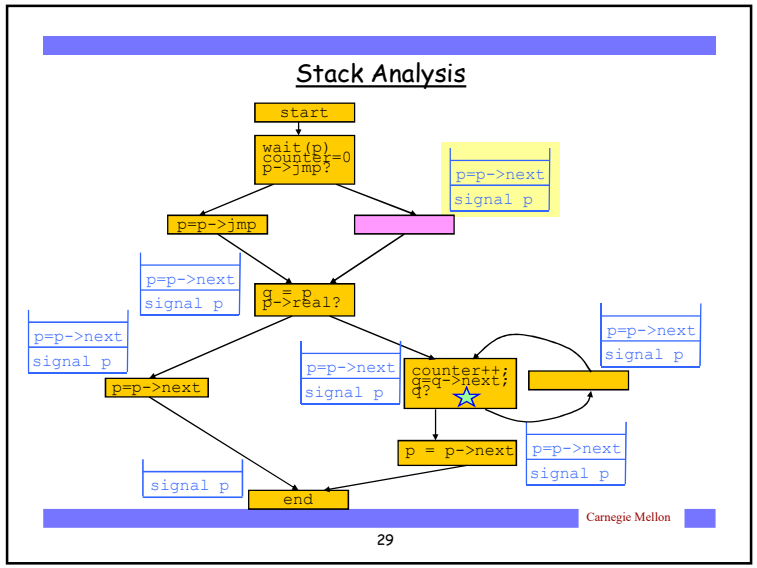

Stack Analysis
start
p=p->next
signal p
p=p->next
signal p
counter++;
q=q->next;
q?
p = p->next
p=p->next
signal p
signal p
end
Carnegie Mellon
24

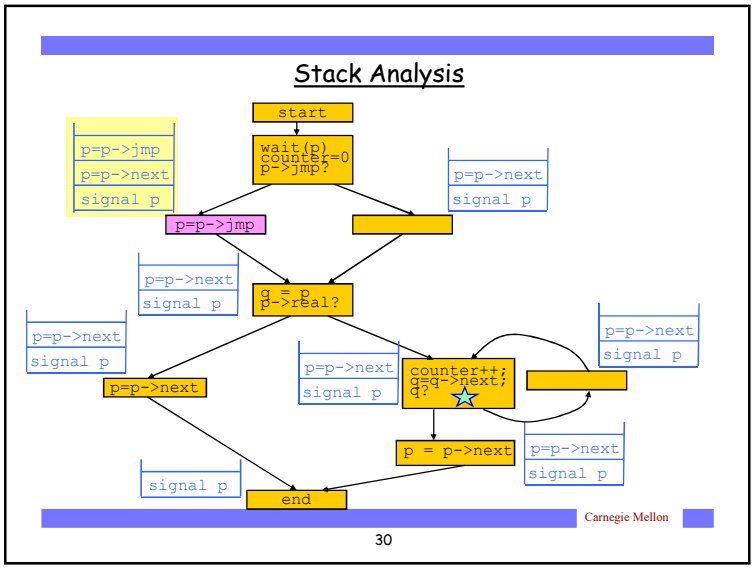

Stack Analysis
start
wait(p)
counter=0
p->jmp?
p=p->jmp
q = p
p->real?
p=p->next
signal p
p=p->next
signal p
counter++;
q=q->next;
q?
p = p->next
p=p->next
signal p
signal p
end
Carnegie Mellon
25

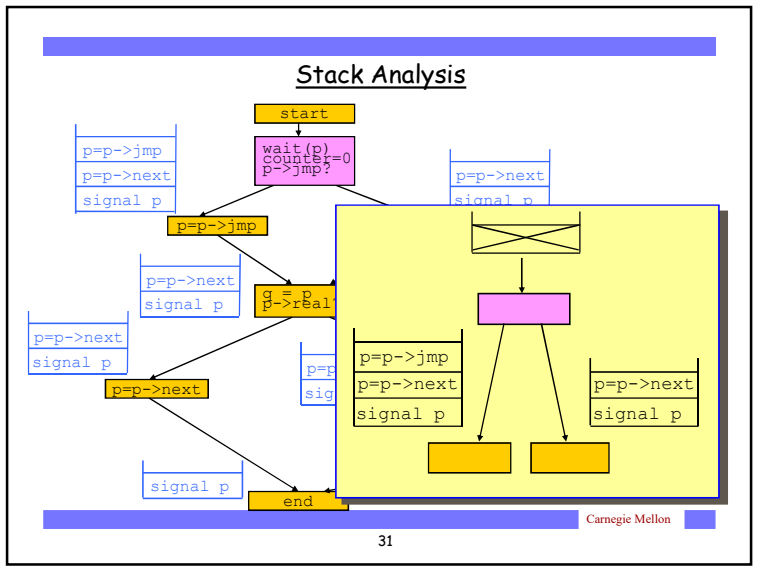

Stack Analysis
start
wait(p)
counter=0
p->jmp?
p=p->jmp
q = p
p->real?
p=p->next
signal p
p=p->next
signal p
p=p->next
signal p
counter++;
q=q->next;
q?
p = p->next
p=p->next
signal p
signal p
end
Carnegie Mellon
26

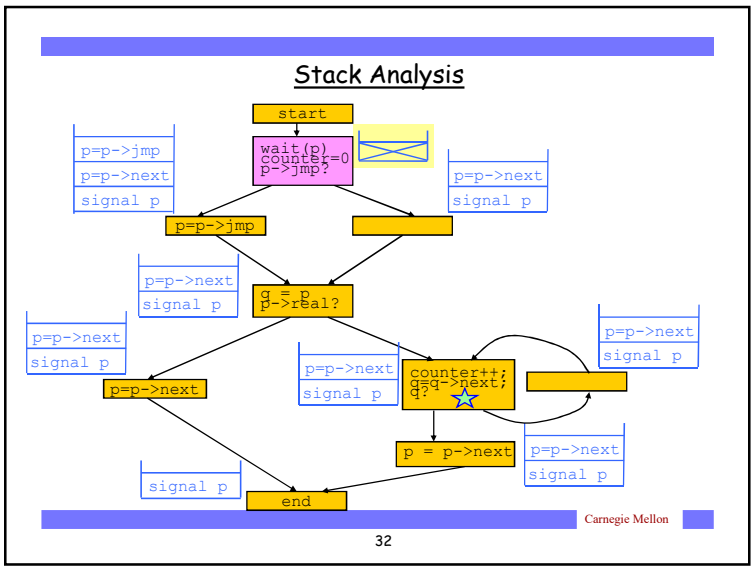

Stack Analysis
start
wait(p)
counter=0
p->jmp?
p=p->jmp
q = p
p->real?
p=p->next
signal p
p=p->next
signal p
p=p->next
signal p
p=p->next
signal p
p = p->next
p=p->next
signal p
signal p
end
Carnegie Mellon
27


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
counter++;
q=q->next;
q?
p = p->next
p=p->next
signal p
signal p
end
Carnegie Mellon
28

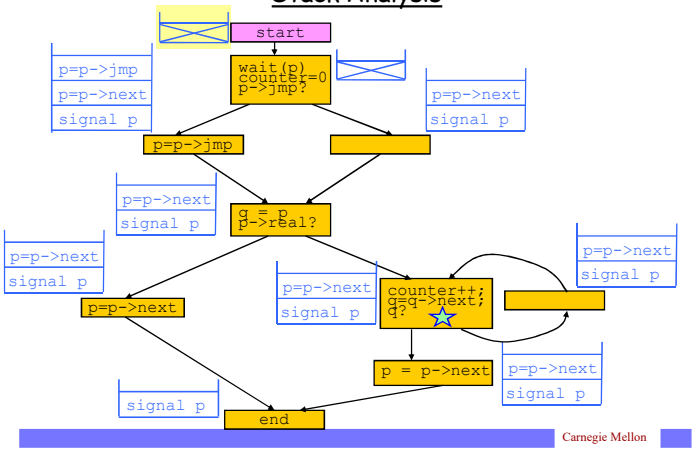

Stack Analysis
start
wait(p)
counter=0
p->jmp?
p=p->jmp
p=p->next
signal p
g = p
p->real?
p=p->next
counter++;
q=q->next;
q?
p = p->next
end
Carnegie Mellon
29


Stack Analysis
start
wait(p)
counter=0
p->jmp?
p=p->jmp
p=p->next
signal p
g = p
p->real?
counter++;
q=q->next;
q?
p = p->next
end
Carnegie Mellon
30

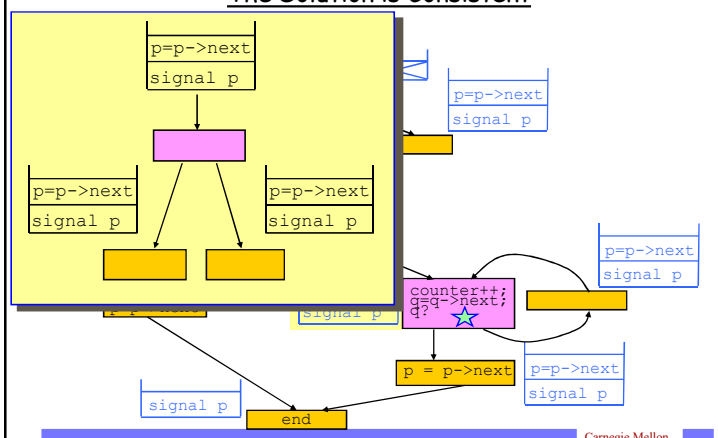

Stack Analysis
start
wait(p)
counter=0
p->jmp?
p=p->jmp
p=p->next
signal p
g = p
p->real
p=p->next
end
Carnegie Mellon
31


Stack Analysis
start
wait(p)
counter=0
p->jmp?
p=p->jmp
p=p->next
signal p
g = p
p->real?
counter++;
q=q->next;
q?
p = p->next
end
Carnegie Mellon
32

Stack Analysis



The Solution is Consistent



Scheduling Instructions

Dataflow analysis

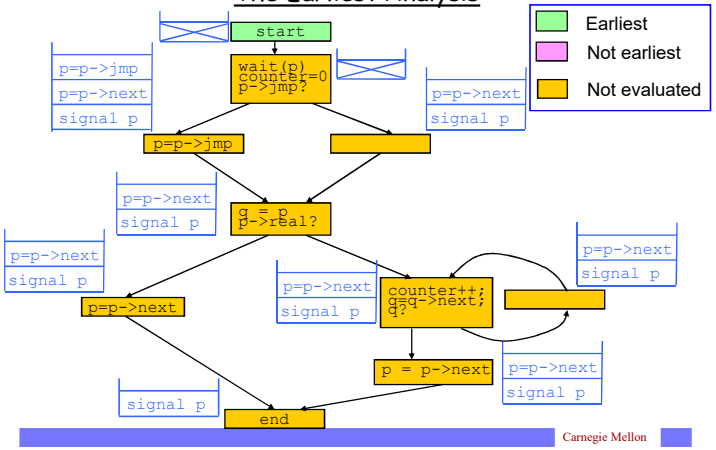
Handles complex control flow

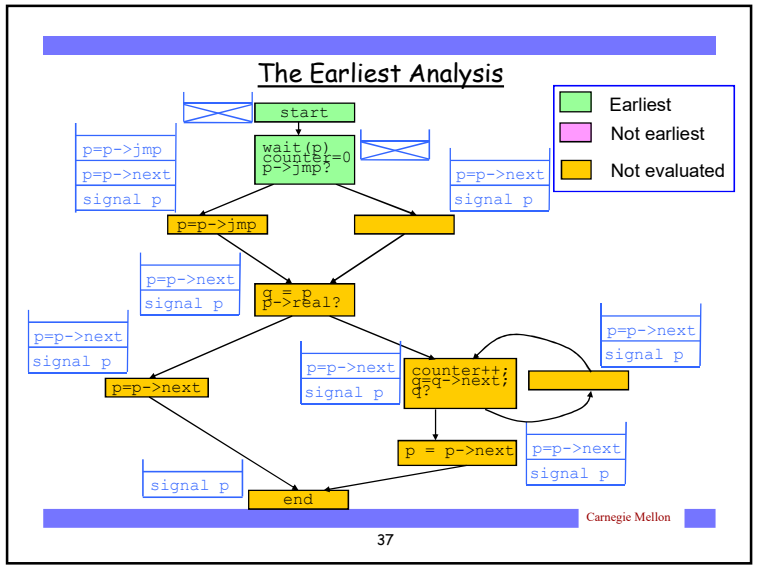
Define two dataflow analyses

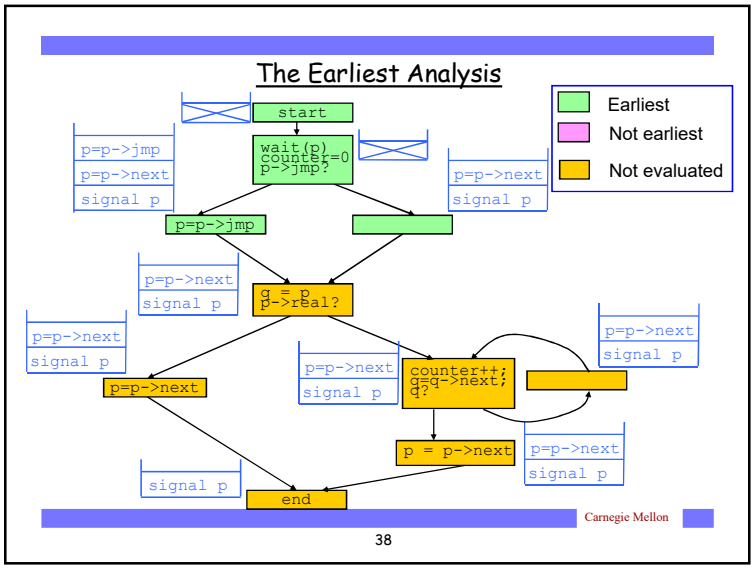
Stack
Find the instructions to compute the forwarded value?

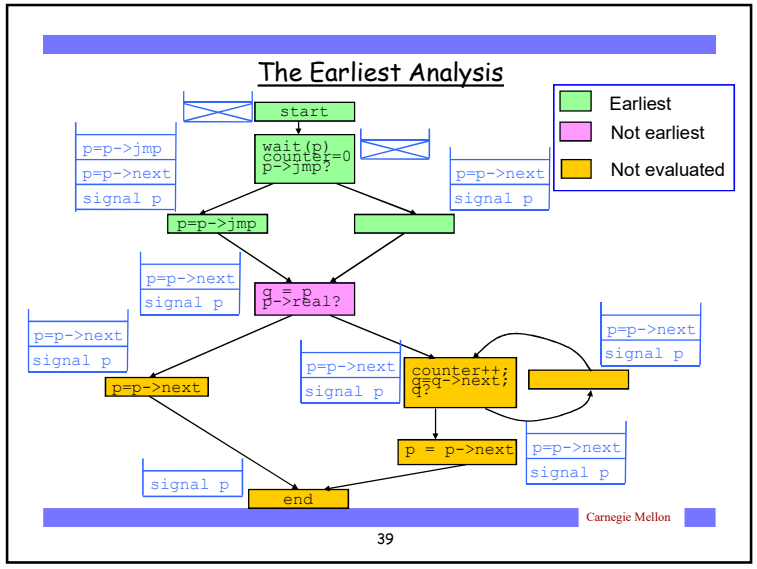
☞ Earliest
Find the earliest node to compute the forwarded value?

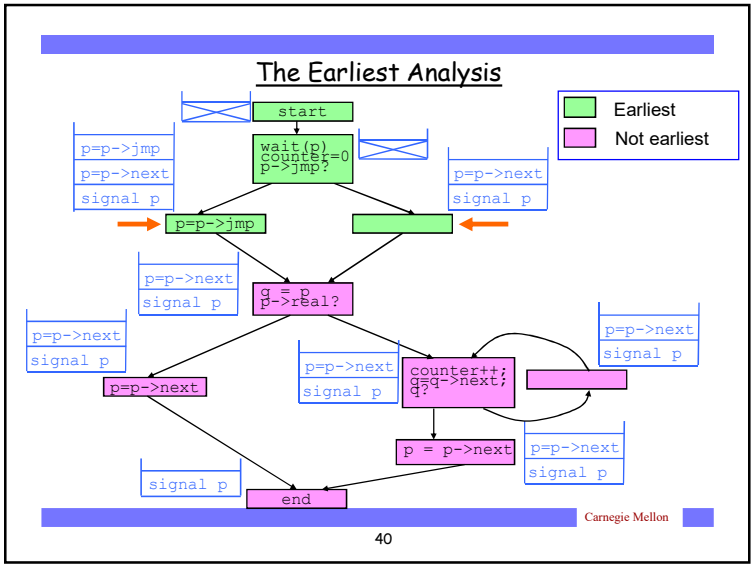
The Earliest Analysis



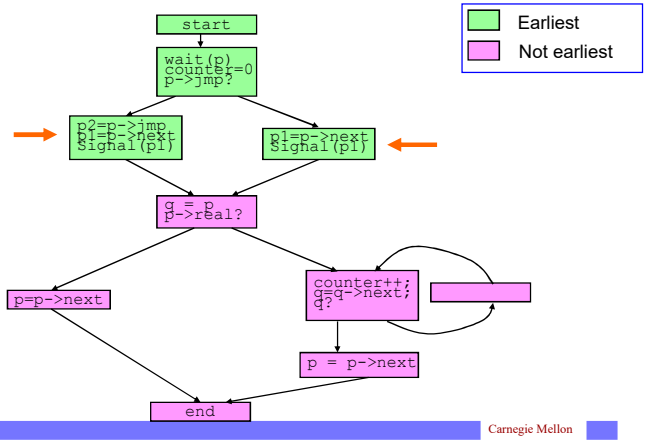

The Earliest Analysis
Earliest
Not earliest
Not evaluated
start
wait(p)
counter=0
p->jmp?
p=p->jmp
p=p->next
signal p
q = p
p->real?
counter++;
q=q->next;
q?
p = p->next
end
Carnegie Mellon
37


The Earliest Analysis
Earliest
Not earliest
Not evaluated
start
wait(p)
counter=0
p->jmp?
p=p->jmp
p=p->next
signal p
q = p
p->real?
counter++;
q=q->next;
q?
p = p->next
end
Carnegie Mellon
38


The Earliest Analysis
Earliest
Not earliest
Not evaluated
start
wait(p)
counter=0
p->jmp?
p=p->jmp
p=p->next
signal p
q = p
p->real?
counter++;
q=q->next;
q?
p = p->next
end
Carnegie Mellon
39


The Earliest Analysis
Earliest
Not earliest
start
wait(p)
counter=0
p->jmp?
p=p->jmp
p=p->next
signal p
q = p
p->real?
counter++;
q=q->next;
q?
p = p->next
end
Carnegie Mellon
40

Code Transformation



Experimental Framework

Benchmarks

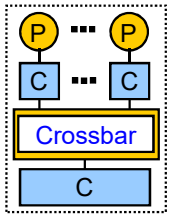
- from SPECint95 and SPECint2000, -O3 optimization

Underlying architecture

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

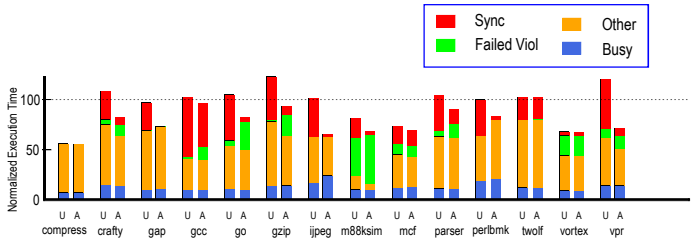
Simulator

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



☞ detailed simulation!

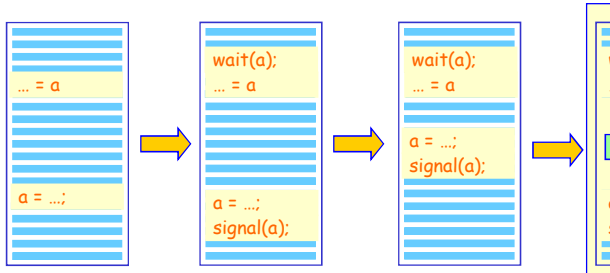
Instruction Scheduling



U=No Instruction Scheduling
A=Instruction Scheduling

☞ Improves performance by 18% over no instruction scheduling

Compiler's Tasks



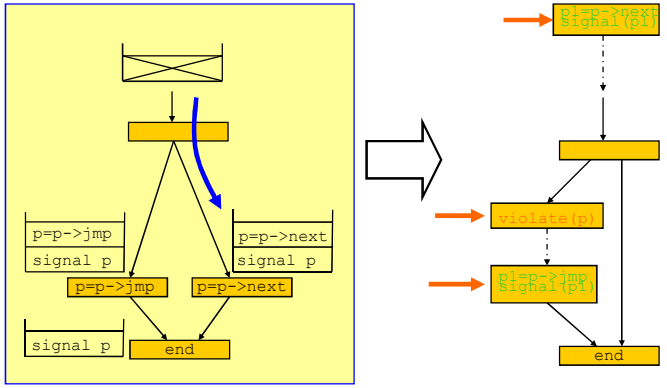
Identifying Forwarding Scalar

Inserting Wait/Signal

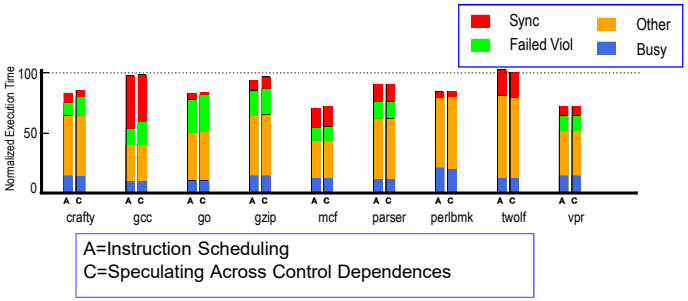
Scheduling Instructions

Scheduling Instructions Speculatively

Speculating Beyond a Control Dependence



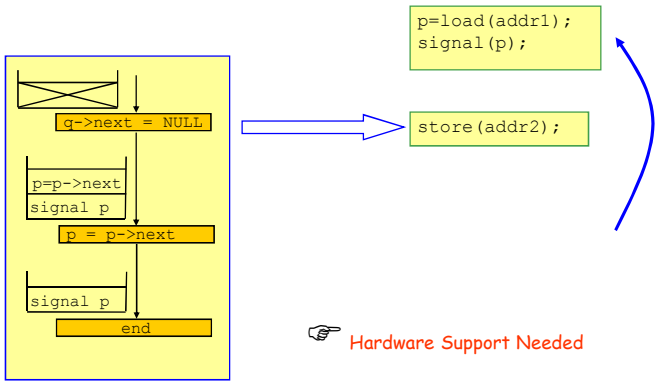
Speculatively Scheduling Instructions Across Control Dependences



A=Instruction Scheduling
C=Speculating Across Control Dependences

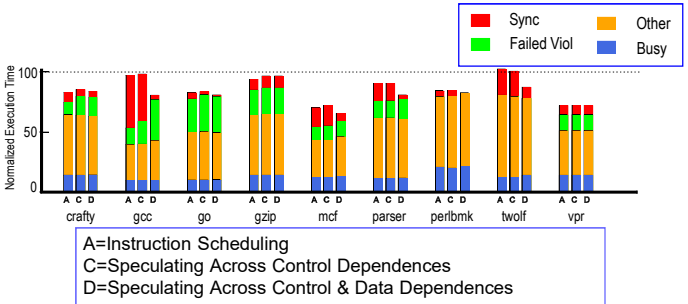
☞ No significant performance gain

Speculating Beyond a Potential Data Dependence



☞ Hardware Support Needed

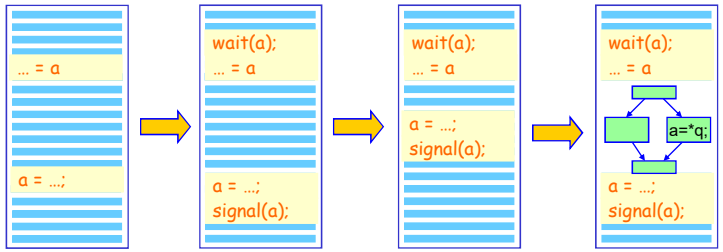
Aggressively Scheduling Instructions Across Both Control and Data Dependences



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

☞ Improves performance significantly for some benchmarks

Summary: Compiler Support for Thread-Level Speculation



Identifying Forwarding Scalar

Inserting Wait/Signal

Scheduling Instructions

Scheduling Instructions Speculatively

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
  - One biggest benefactor is GCC, performance improved by 18%
  - Some hardware support needed

☞ Critical forwarding path can be addressed by the compiler

What's Next

- Project Milestone Reports due midnight Monday
- Exam on Wednesday