Architecture 2024

Chapter 5
Instruction-Level Parallelism and Its Exploitation

Hiroaki Kobayashi, Masayuki Sato

Tohoku University

Ver. Jan 7,2025

Contents and Schedule

1. Trends in Computer Architecture Design
2. Instruction Set Principles and Examples
3. Pipelining
4. Memory Hierarchy (1/7)
5. Instruction Level Parallelism and Its Exploitation(1/7?)
6. Multicores, Multiprocessors, and Clusters (1/14)
7. Data-Level Parallelism: SIMD, Vector, and GPU (1/14)
8. Warehouse-Scale Computers (1/21)

Contents and schedule are subject to change.

Techniques to Avoid Pipeline-Stalls

- Maintain the dependence among the instructions but avoid a hazard, and eliminate a dependence by transforming the code to increase instruction-level parallelism, and then
- Fill stall-slots in the pipeline with independent instructions by their reordering
  - Exploit instruction-Level Parallelism (ILP), parallelism among instructions

Two approaches to exploiting ILP

- Dynamic reordering of instruction execution by hardware
  - Pentium III&4, Athlon, MIPSR10000/12000, UltraSPARC III, PowerPC G3&G4, Alpha 21264
- Static reordering of instruction execution by compiler
  - IA-64 (Itanium 1&2)

Importance of Data Dependences and Hazards

- Determining whether an instruction is dependent on another instruction is critical to …
  - determining how much parallelism exists in a program and
  - determining how that parallelism can be exploited.

☺ If two instructions are *parallel*, they can execute simultaneously in a pipeline without causing any stalls

☹ If two instructions are dependent, they are not parallel and must be executed in order, though they may often be partially overlapped in pipelining.

Data Dependences

- Data dependences

  An instruction *j* is *data dependent* on instruction *i* if either of the following holds

  - instruction *i* produces a result that may be used by instruction *j*, or
  - instruction *j* is data dependent on instruction *k*, and instruction *k* is data dependent on instruction *i*.

```
Loop   LD     F0,0(R1)
       ADD     F4,F0,F2
       SD      F4,0(R1)
       DADDUI  R1,R1,-8
       BNE     R1,R2,LOOP
```

```
LOOP LD     F0,0(R1)
       ADD     F4,F0,F2
       SD      F4,0(R1)
       DADDUI R1,R1,-8
       BNE     R1,R2,LOOP
```

Name Dependences

A name dependence occurs when

- two instructions use the same register or memory location, called *name*,
- but there is no flow of data between the instructions associated with that name.

- An *antidependence* between instructions *i* and *j* occurs
  - when *j* writes a register or memory location that *i* reads.
- An *output dependence* occurs
  - when instructions i and j write the same register or memory location.

Since a name dependence is not a true dependence, it can be solved if the number of registers is enough.

Data Hazards

- A hazard is created whenever
  - there is a dependence between instructions, and
  - they are close enough that the overlap caused by pipelining, or other reordering of instructions, would change the order of access to the operand involved in the dependence.

* The goal of software and hardware techniques is to exploit parallelism by preserving program order *only where it affects the outcome of the program.*

Types of Data Hazards

- RAW (read after write)
  - *j* tries to read a source before *i* writes it, so j incorrectly gets the old value in the case of *i* occurring before *j* in program order
  - True dependence
- WAW (write after write)
  - *i* tries to write an operand before it is written by *i*.
  - Output dependence
- WAR (write after read)
  - *j* tries to write a destination before it is read by *i*, so *i* incorrectly gets the new value.
  - Antidependence

- RAW

```
LD    F0, 0(R2)

ADD   F4,F0,F2
```

- WAW

```
ADD   F4,F0,F2
  ...
DIV   F4, F1,F3
```

- WAR

```
ADD   F4, F0, F2

LD    F0, 0(R2)
```

Exploiting Instruction-Level Parallelism with Hardware Approaches

Overcoming Data Hazards with Dynamic Scheduling

- Dynamic scheduling
  - Hardware rearranges the instruction execution to reduce the stalls while maintaining data flow and exception behavior.

- Advantages
  - It enables handling some cases when dependences are unknown at compile time (e.g., memory references)
  - It simplifies the compiler
  - It allows code that was compiled with one pipeline in mind to run efficiently on a different pipeline.

☹ The advantages of dynamic scheduling are gained at a cost of significant increase in hardware complexity.

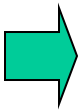
Dynamic Scheduling: The Idea

- Simple pipelining techniques use in-order instruction issue and execution.
  - Instructions are issued in program order, and if an instruction is stalled in the pipeline, no later instructions can proceed.

Example:

```
DIV     F0,F2,F4
ADD     F10,F0,F8
SUB     F12,F8,F14
```

SUB cannot executed because dependence of ADD on DIV causes the pipeline to stall!



If we can both check for any structural hazards and wait for the absence of a data hazard in the ID stage, the pipeline can do *out-of-order execution*.

An instruction can begin execution as soon as its data operand is available

Some Concerns about Dynamic Scheduling

- WAR and WAW hazards

Example

DIV	F0,F2,F4	
ADD	F6,F0,F8	RAW
SUB	F8,F10,F14	WAR
MUL	F6,F10,F8	WAW

If the pipeline executes SUB before ADD, it will violate the antidependence (WAR).

Likewise, to avoid violating output dependences, WAW hazards must be handled.

- Precise exception handling
  - Dynamic scheduling with out-of-order completion must preserve exception behavior in the sense that *exactly* those exceptions that would arise if the program were executed in strict program order *actually* do arise.

Dynamically scheduled processors preserve exception behavior by ensuring that no instruction can generate an exception until the processor knows that the instruction raising the exception will be executed.

New Pipeline Staging for Out-Of-Order Execution

To allow out-of-order execution, the ID pipe stage is split into two stages

- *Issue:* Decode instructions, check for structural hazards
- *Read operands:* Wait until no data hazards are observed, and the read operands are ready for execution.

- An instruction fetch stage precedes the issue stage and may fetch either into an instruction register or into a queue of pending instructions
- Instructions are then issued from the register or queue.
- The EX stage follows the read operands stage, just as in the five-stage pipeline

To take advantage of dynamic scheduling,
- pipelines allows multiple instructions to be in execution at the same time, and
- have multiple functional units, pipelined functional units or both.

Basic Dynamic Scheduling

In a dynamically scheduled pipeline,

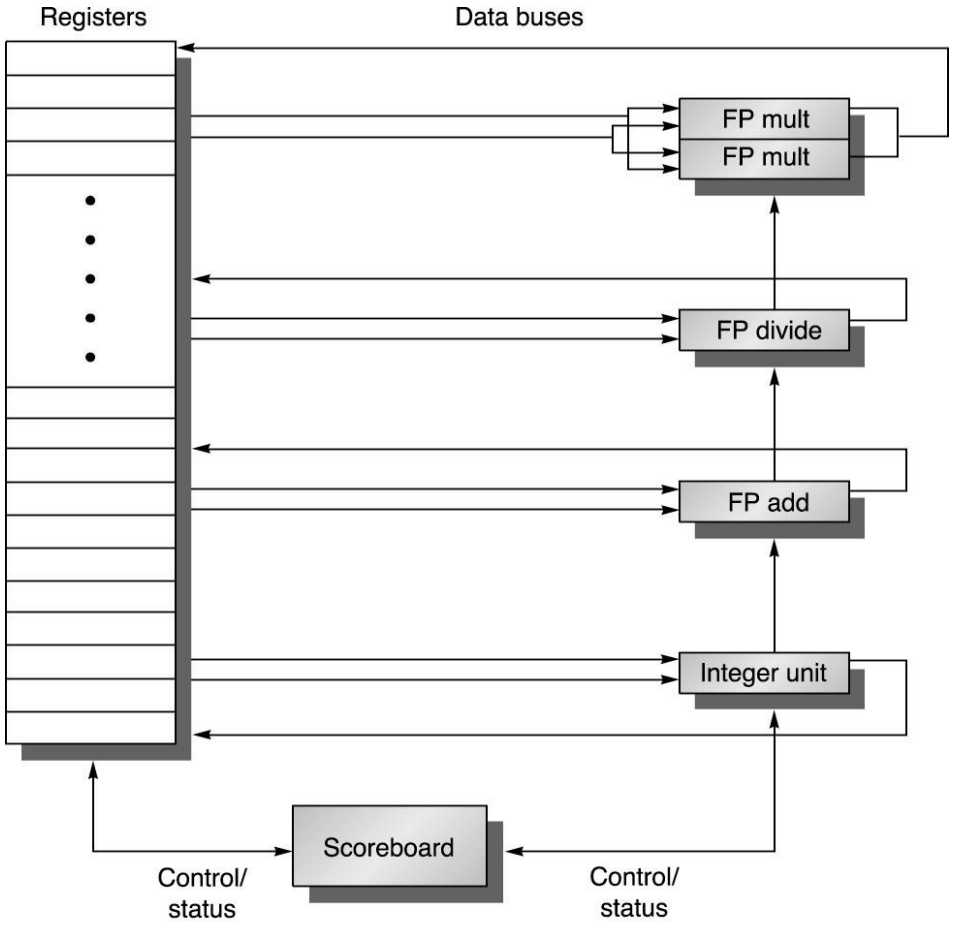
- all instructions pass through the issue stage in order *(in-order issue)*, however,
- they can be *stalled or bypass* each other in second stage (read operand) and
- thus enter execution *out of order*.

- Scoreboarding: The basic idea
  - When the next instruction to execute is stalled, other instructions can be issued and executed if they do not depend on any active or stalled instruction.

DIV	F0,F2,F4	
ADD	F6,F0,F8	RAW
SUB	F8,F10,F14	WAR
MUL	F6,F10,F8	WAW

Scoreboarding (First introduced in CDC6600)

- All hazard detection and resolution are centralized in the scoreboard to control instruction execution.
  - Every instruction goes through the scoreboard, where a record of the data dependences is constructed.
  - The scoreboard determines when the instruction can read its operands and begin execution.
  - If the scoreboard decides the instruction cannot execute immediately, it monitors every change in the hardware and decides when the instruction can be executed.
  - The scoreboard also controls when an instruction can write its result into the destination register.



The basic four steps for scoreboarding

- Issue
  - If *a functional unit for the instruction is free* and *no other active instruction has the same destination register*, the scoreboard issues the instruction to the functional unit and updates it internal data structure
  - Structural and WAW hazard handling
- Read operands
  - The scoreboard *monitors the availability of the source operands.*
  - When the source operands are available, the scoreboard tells the functional unit to proceed to read the operands from the register and begin execution.
  - Dynamic RAW hazard handling
- Execution
  - The functional unit begins execution upon receiving operands
  - When the result is ready, it notifies the scoreboard that it has completed execution
- Write result
  - Once the scoreboard is aware that the functional unit has completed execution, the scoreboard *checks for WAR hazards and stall the completing instruction*, if necessary.
  - If this WAR hazard does not exist, or when it clears, the scoreboard tells the functional unit to store its result to the destination register.
  - WAR hazard handling

Data Structures for Scoreboarding

- Instruction status
  - Indicates which of the four steps the instruction is in.
- Functional unit status
  - Indicates the state of the functional unit (FU).
  - There are nine fields for each functional unit
    - Busy: indicates whether the unit is busy or not
    - Op: operation to perform in the unit
    - Fi: destination register
    - Fj, Fk: source registers
    - Qj, Qk: Functional units producing source registers Fj,Fk
    - Rj,Rk: Flags indicating when Fj, Fk are ready and not yet read. Set to No after operands are read
- Register result status
  - Indicates which functional unit will write each register, if an active instruction has the register as its destination.
  - This field is set to blank whenever there are no pending instructions that will write that register.

Example

- Example code

```
LD   F6,34(R2)
LD   F2,45(R3)
MUL  F0,F2,F4
SUB  F8,F6,F2
DIV  F10,F0,F6
ADD  F6,F8,F2
```

		Instruction Status			
Instruction		Issue	Read Operand	Execution complete	Write result
LD	F6,34(R2)	×	×	×	×
LD	F2,45(R3)	×	×	×	
MUL	F0,F2,F4	×			
SUB	F8,F6,F2	×			
DIV	F10,F0,F6	×			
ADD	F6,F8,F2				

Example (Cont'd)

Functional Unit Status

Name	Busy	Op	Fi	Fj	Fk	Qj	Qk	Rj	Rk
Integer	Yes	Load	F2	R3				No	
Multi1	Yes	Mult	F0	F2	F4	Integer		No	Yes
Multi2	No								
Add	Yes	Sub	F8	F6	F2		Integer	Yes	No
Divide	Yes	Div	F10	F0	F6	Mult1		No	Yes

SUB.D F8, F6, F2

Register Results Status

	F0	F2	F4	F6	F8	F10	F12	…	F30
FU	Mult1	Integer			Add	Divide			

Required Checks and Bookkeeping Actions

Instruction Status	Wait until	Bookkeeping
Issue	Not Busy[FU] and Not Result[D]	Busy[FU]←yes;Op[FU] ←op;Fi[FU] ←D; Fj[FU] ←S1;Fk[FU] ←S2; Qj ←Result[S1]; Qk ←Result[S2]; Rj ←not Qj; Rk ←not Qk; Result[D] ←FU
Read operands	Rj and Rk	Rj ←No; Rk ←No; Qj ←0; Qk ←0;
Execution Complete	Functional unit done	
Write result	$\forall f$((Fj[f]≠Fi[FU] or Rj[f]=No) & (Fk[f] ≠Fi[FU] or Rk[f]=No)	$\forall f$(if Qj[f]=FU then Rj[f] ←Yes); $\forall f$(if Qk[f]=FU then Rk[f] ←Yes); Result [Fi[FU]] ←0; Busy[FU] ←No);

Scoreboarding


1.7x Performance improvement for a Fortran program on CDC6600


Main Cost: Large number of buses

Factors Limiting the Scoreboard Performance

- ✓ The amount of parallelism available among the instruction
  - This determines whether independent instructions can be found to execute.

This is due to…

1. The number of scoreboard entries limited!
   - This determines how far ahead the pipeline can look for independent instructions.
   - The set of instructions examined as candidates for potential execution is called the *window*.
2. The number and types of functional units limited!
   - This determines the importance of structural hazards, which can increase when dynamic scheduling is used
3. The presence of antidependences and output dependences unavoidable!
   - These lead to WAR and WAW stalls

Tomasulo's Approach: Solve the Problems of Scoreboarding!

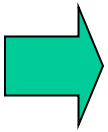
Invented by Robert Tomasulo for IBM 360/91 FP units

- Tracks when operands for instructions are available, to minimize RAW hazards
  - Reservation stations of functional units buffers the operands of instructions waiting to issue
- Introduces register renaming, to minimize WAW and WAR hazards
  - *Register renaming* eliminates WAR and WAW hazards by renaming all destination registers, including those with a pending read or write for an earlier instruction.
- Hazard detection and execution control are distributed

Dynamic Scheduling with Register Renaming

- Example

```
DIV   F0,F2,F4
ADD   F6,F0,F8
ST    F6,0(R1)
SUB   F8,F10,F14
MUL   F6,F10,F8
```



Register Renaming with two temporary registers S and T

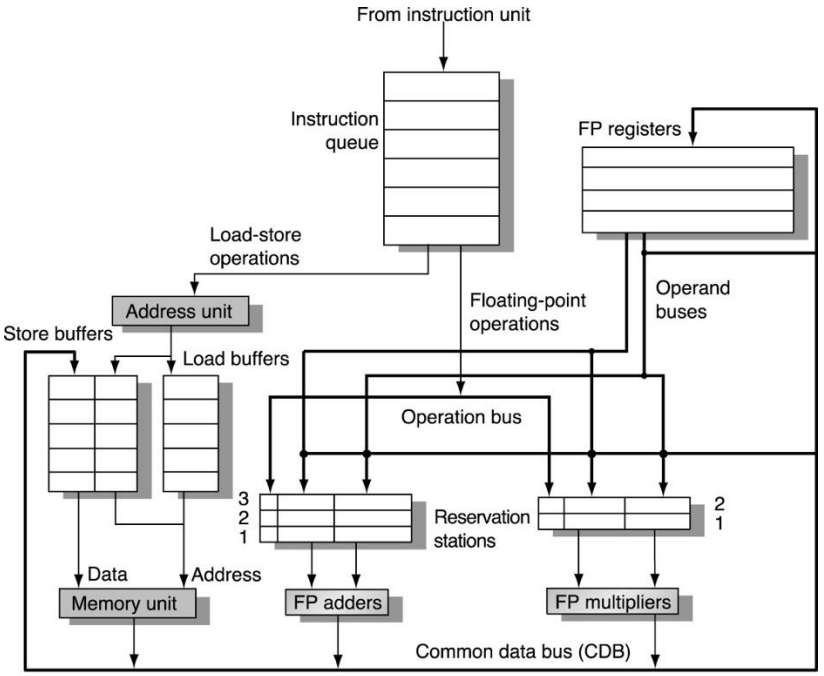
```
DIV   F0,F2,F4
ADD   S, F0,F8
ST    S,0(R1)
SUB   T,F10,F14
MUL   F6,F10,T
```

RAW hazards only!

- True dependences…
  DIV&ADD, SUB&MUL, ADD&ST
- Antidependences…
  ADD&SUB
- Output dependencies…
  ADD&MUL

Reservation Stations for Register Renaming

- As instructions are issued, the register specifiers for pending operands are renamed to the names of the reservation stations
- A reservation station fetches and buffers an operand as soon as it is available, eliminating the need to get the operand from a register.
- Pending instructions designate the reservation station that will provide their input.



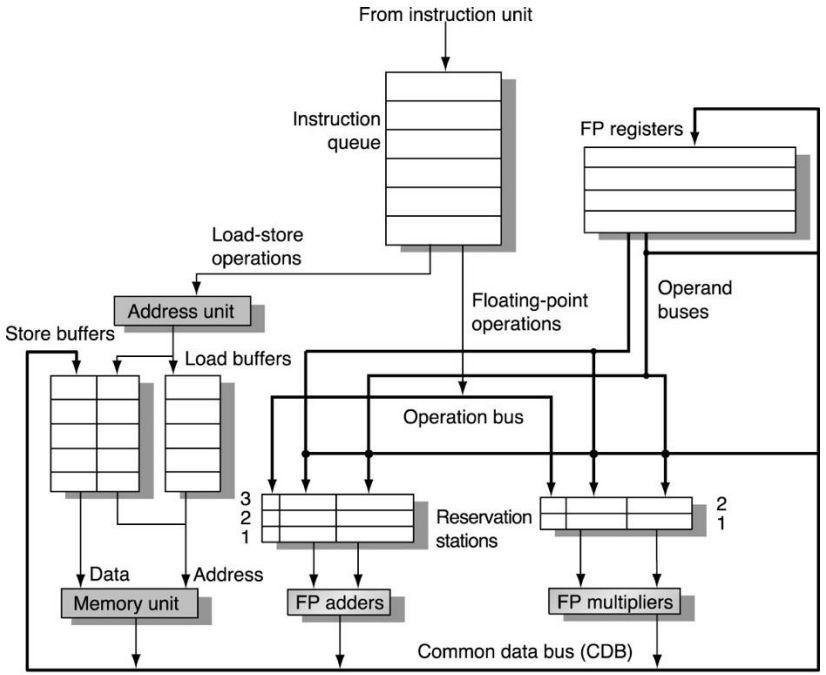
Since there can be more reservation stations than real registers, the technique can even eliminate hazards arising from name dependences.

The Advantages of Tomasulo's Approach

- Hazard detection and execution control are distributed.
  - The information held in the reservation stations at each functional unit determines when an instruction can begin execution at that unit.
- WAW and WAR hazards are eliminated by register renaming
- Results are passed directly to functional units from the reservation stations where they are buffered, rather than going through the registers.
  - This bypassing is done with a common result bus that allows all units waiting for an operand to be loaded simultaneously.

Pipeline Stages for Tomasulo's Approach

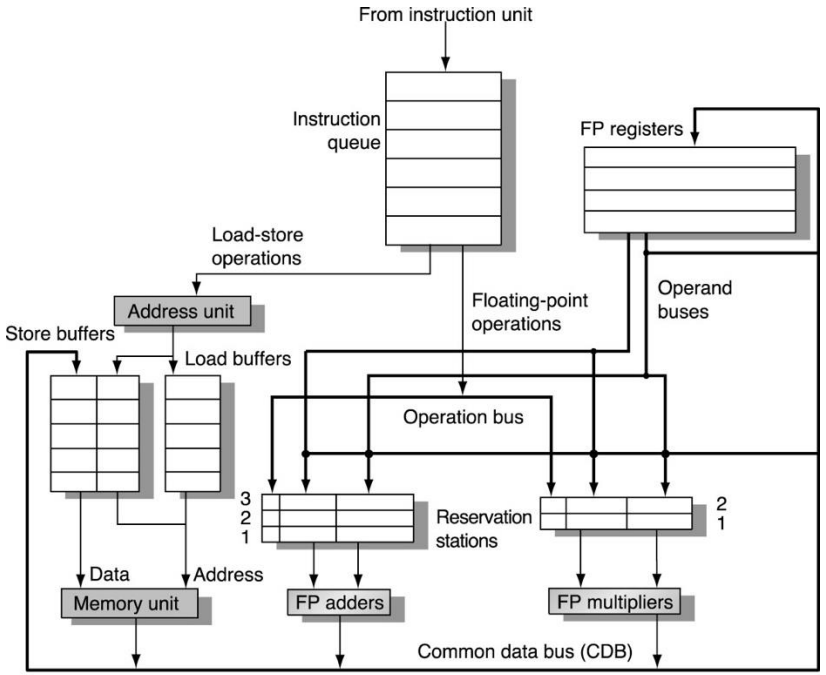
- Issue
  - Get the next instruction from the head of the instruction queue, which is maintained in FIFO order to ensure the maintenance of correct data flow.
  - If there is a matching reservation station that is empty, issue the instruction to the station with the operand values, if they are currently in the registers.
  - If there is not an empty reservation station, then there is a structural hazard and the instruction stalls until a station or buffer is freed
  - If the operands are not in the registers, keep track of the functional units that will produce the operands



This step renames registers, eliminating WAR and WAW hazards.

Pipeline Stages for Tomasulo's Approach

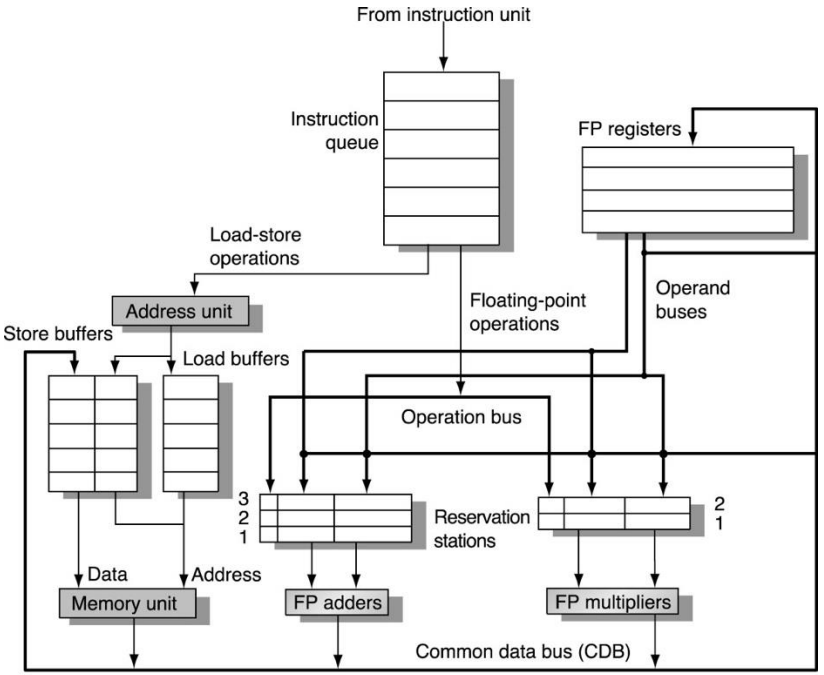
- Execute
  - If one or more of the operands is not yet available, monitor the common data bus while waiting for it to be computed.
  - When an operand becomes available, it is placed into the corresponding reservation station.
  - When all the operands are available, the operation can be executed at the corresponding functional unit.



By delaying instruction execution until the operands are available, RAW hazards are avoided.

Pipeline Stages for Tomasulo's Approach

- Write results
  - When the result is available, write it on the CDB, and from there into the registers and into any reservation stations waiting for this result.
  - Stores also write data to memory during this step: When both the address and data value are available, they are sent to the memory unit and the store completes.



Dynamic Scheduling Example by Tomasulo's Approach

Example code

```
LD      F6,34(R2)
LD      F2,45(R3)
MUL     F0,F2,F4
SUB     F8,F6,F2
DIV     F10,F0,F6
ADD     F6,F8,F2
```

What the information tables look like for this code sequence when only the first load has completed and written its results?

Instruction Status

Instruction		Issue	Execute	Write result
LD	F6,34(R2)	×	×	×
LD	F2,45(R3)	×	×	
MUL	F0,F2,F4	×		
SUB	F8,F6,F2	×		
DIV	F10,F0,F6	×		
ADD	F6,F8,F2	×		

Reservation Stations

Reservation Stations							
Name	Busy	Op	Vj	Vk	Qj	Qk	A
Load1	no						
Load2	yes	Load					45+Regs[R3]
Add1	yes	SUB		Mem[34+Regs[R2]]	Load2		
Add2	yes	ADD			ADD1	Load2	
Add3	no						
Mult1	yes	MUL		Regs[F4]	Load2		
Mult2	yes	DIV		Mem[34+Regs[R2]]	Mult1		

Busy: indicates that this reservation station and its accompanying functional unit are occupied.
Op: the operation to perform on source operands S1 and S2
Vj,Vk: The value of the source operands.
Qj,Qk: the reservation stations that will produce the corresponding source operand.
A: Used to hold information for the memory address calculation for a load and store

Register Status Table

	Register Status								
	F0	F2	F4	F6	F8	F10	F12	…	F30
Qi	Mult1	Load2		Add2	Add1	Mult2			

Qi: The number of the reservation station that contains the operation whose result should be stored into this register.

Example

- Using the same code segment as in the previous example, show what the status tables look like when the MUL is ready to write its result.

		Instruction Status		
Instruction		Issue	Execute	Write result
LD	F6,34(R2)	×	×	×
LD	F2,45(R3)	×	×	×
MUL	F0,F2,F4	×	×	
SUB	F8,F6,F2	×	×	×
DIV	F10,F0,F6	×	×	
ADD	F6,F8,F2	×	×	×

Example (cont'd)

Reservation Stations

Name	Busy	Op	Vj	Vk	Qj	Qk	A
Load1	no						
Load2	no						
Add1	no						
Add2	no						
Add3	no						
Mult1	yes	MUL	Mem[45+Reg[R3]]	Regs[F4]			
Mult2	yes	DIV		Mem[34+Regs[R2]]	Mult1		

Register Status

	F0	F2	F4	F6	F8	F10	F12	…	F30
Qi	Mult1					Mult2			

Tomasulo's Algorithm: A Loop-Based Example

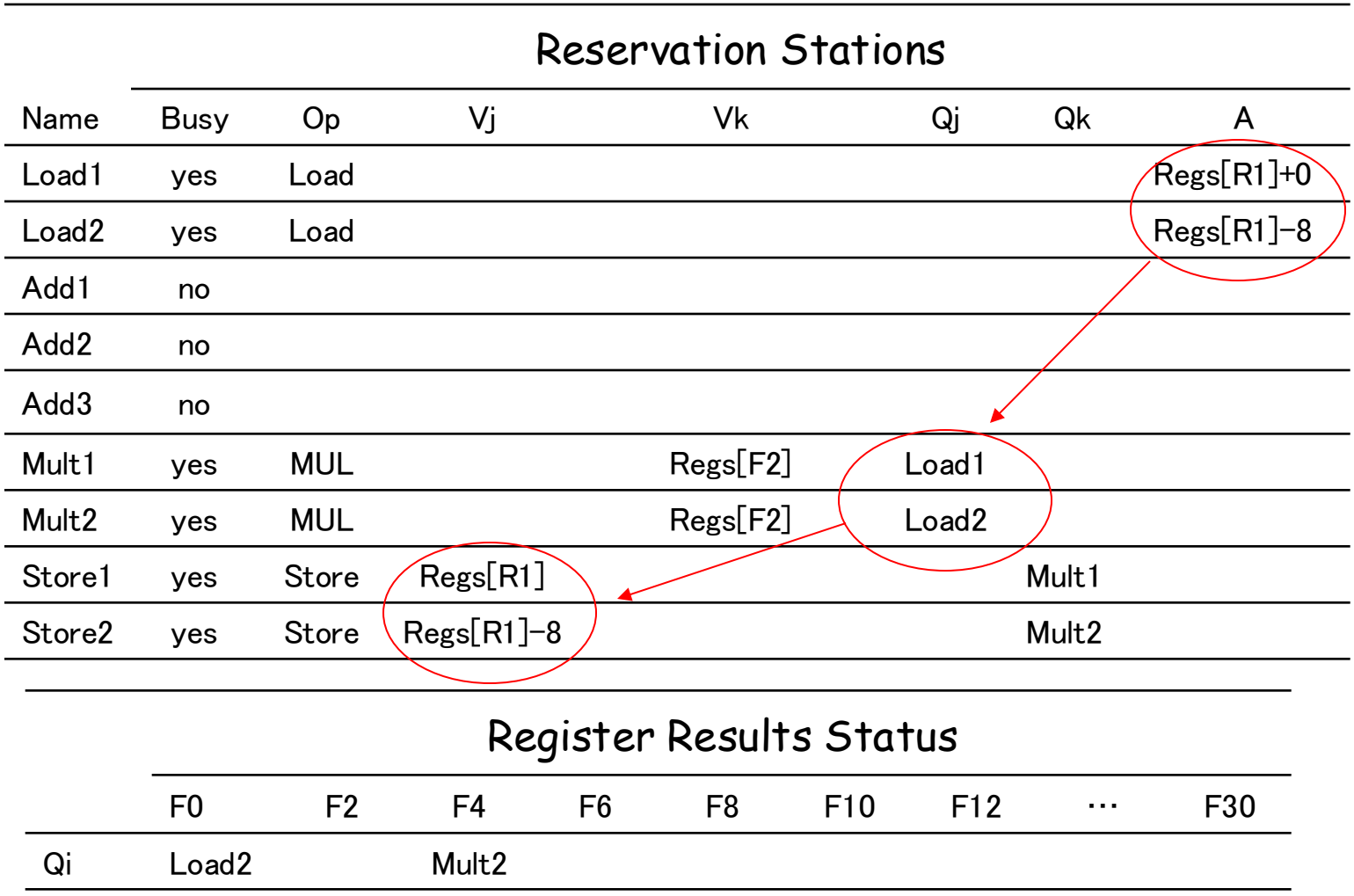
- Example

```
Loop:   LD    F0,0(R1)
        MUL   F4,F0,F2
        ST    F4,0(R1)
        ADD   R1,R1,-8
        BNE   R1,R2,Loop
```

		Instruction Status			
Instruction		From iteration	Issue	Execute	Write result
LD	F0,0(R1)	1	×	×	
MUL	F4,F0,F2	1	×		
ST	F4,0(R1)	1	×		
LD	F0,0(R1)	2	×	×	
MUL	F4,F0,F2	2	×		
ST	F4,0(R1)	2	×		

Two active iterations of the loop with no instruction yet completed.

Tomasulo's Algorithm: A Loop-Based Example



Reservation Stations

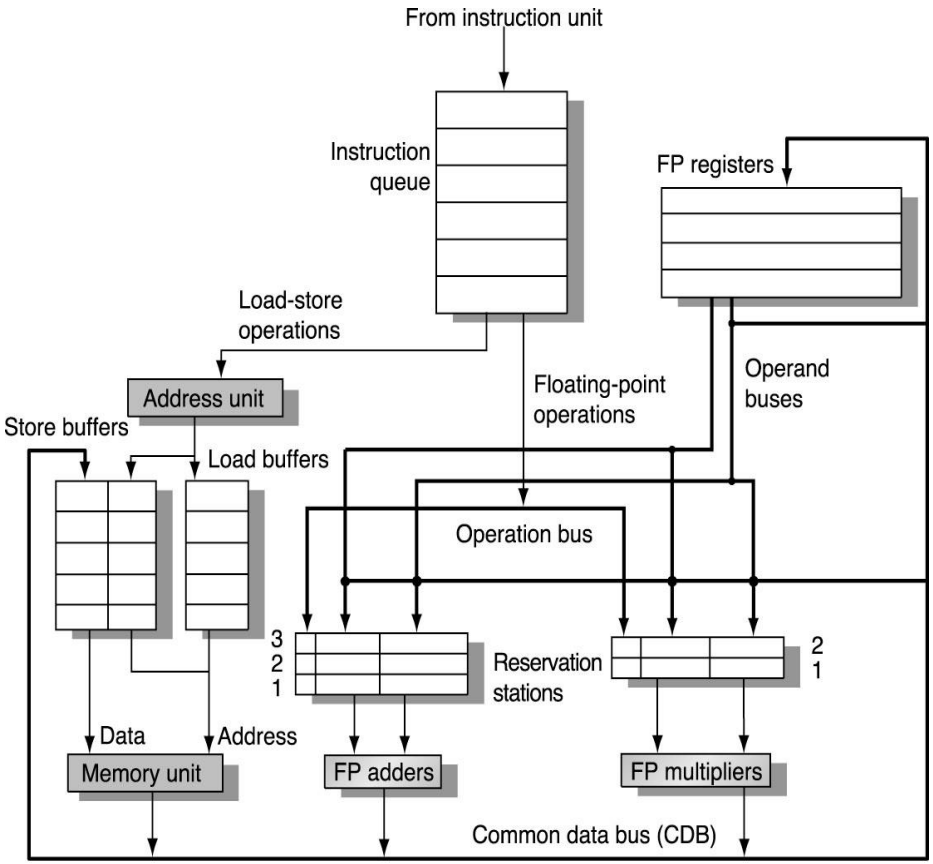
Name	Busy	Op	Vj	Vk	Qj	Qk	A
Load1	yes	Load					Regs[R1]+0
Load2	yes	Load					Regs[R1]−8
Add1	no						
Add2	no						
Add3	no						
Mult1	yes	MUL		Regs[F2]	Load1		
Mult2	yes	MUL		Regs[F2]	Load2		
Store1	yes	Store	Regs[R1]			Mult1	
Store2	yes	Store	Regs[R1]−8			Mult2	

Register Results Status

	F0	F2	F4	F6	F8	F10	F12	…	F30
Qi	Load2		Mult2						

Dynamic Disambiguation of Memory Addresses

- In a load operation, if the load address matches the address of any active entries in the store buffer, the load instruction is not sent to the load buffer until the conflicting store completes.
  - RAW hazards handling for load operations
- In a store operation, processor must check for conflicts in both the load buffers and store buffers.
  - WAW and WAR hazards handling for store operations



Summary of the Tomasulo's Approach

- A dynamically scheduled pipeline by the Tomasulo's algorithm can yield very high performance
  - The key components for enhancing ILP in Tomasulo's algorithm
    - dynamic scheduling
    - register renaming
    - dynamic memory disambiguation
  - ✓ The role of dynamic scheduling as a basis for hardware speculation has made this approach very popular in modern microprocessors!

- The major drawback of this approach is its hardware complexity
  - ➢ Each reservation station must contain an associative buffer, which must run at high speed, as well as complex control logic.
  - The performance can be limited by the single CDB.
    - Although additional CDBs can be added, each CDB must interact with each reservation station, and the associative tag-matching hardware would need to be duplicated at each station for each CDB
  - The size of the reservation stations limits the ILP exploited.

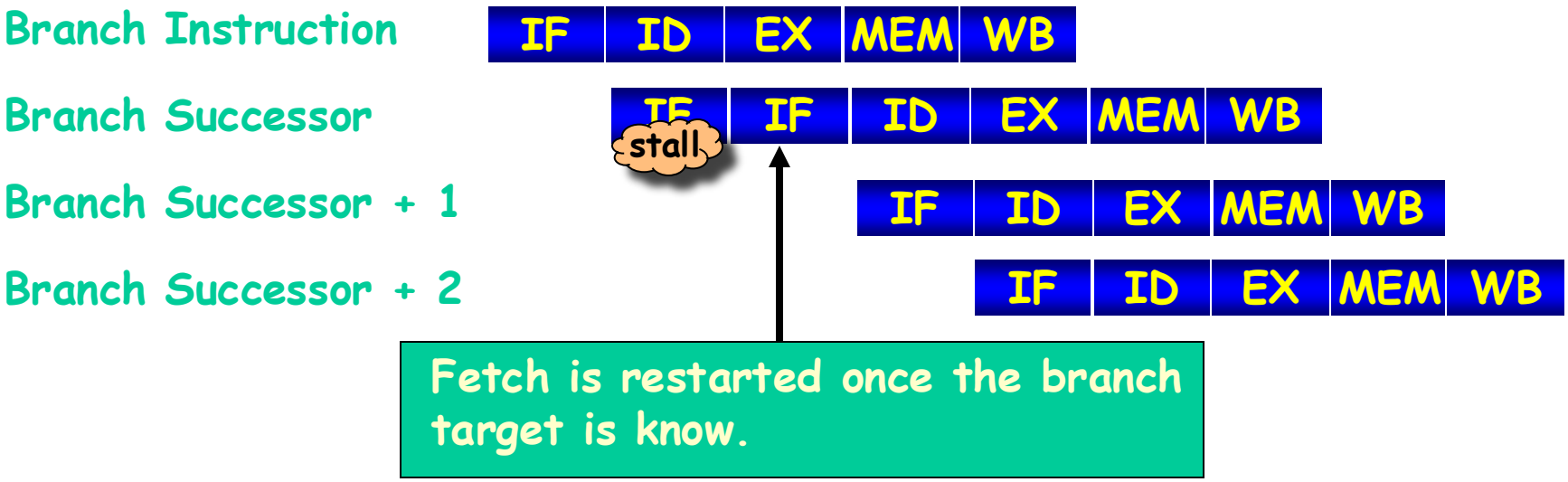
Reducing Branch Costs with Dynamic Hardware Prediction

- As the amount of ILP to exploit grows, control dependences rapidly become the limiting factor.
  - Branches will arrive up to *n* times faster in an *n*-issue processor and providing an instruction stream to the processor will require that we predict the outcome of branches.
  - Amdahl's Law reminds us that relative impact of the control stalls will be larger with the low potential CPI in such machines.
- Branch Prediction Schemes
  - Static branch prediction by compiler
    - Predict-taken/predict-not taken schemes, delayed branch scheme
  - Dynamic branch prediction by hardware
    - Branch-prediction buffer, branch history table, branch target buffer

The goal of these mechanisms is to allow the processor to resolve the outcome of a branch early

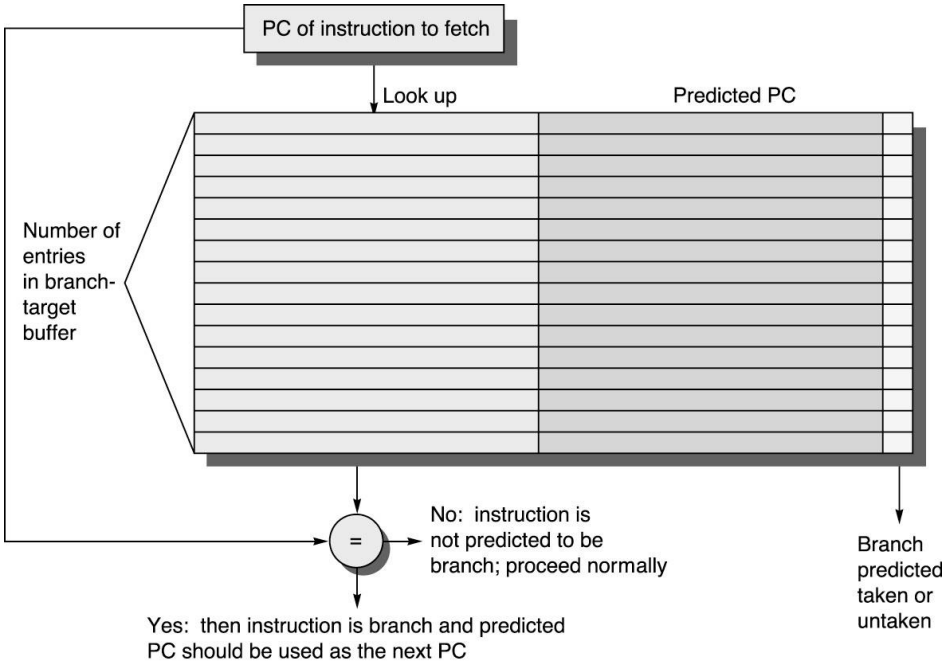
Control (Branch) Hazards

- Execution of branch instructions may or may not change the PC to something other than its current execution sequence.



Basic Branch Prediction and Branch-Prediction Buffers

- Branch Target Buffer
  - A small memory indexed by the lower portion of the address of the branch instruction.
  - The memory contains prediction bits that say whether the branch was recently taken or not.
  - If the prediction turns out to be wrong, the prediction bits are updated.



Prediction accuracy depends on the prediction bits/prediction schemes!

How many bits are required for accurate prediction?

- 1-bit prediction scheme
  - 1-bit holds the information about last branch direction.
  - If the prediction turns out to be wrong, the prediction bit is inverted and stored back.

  ☺ Simple and low cost prediction scheme

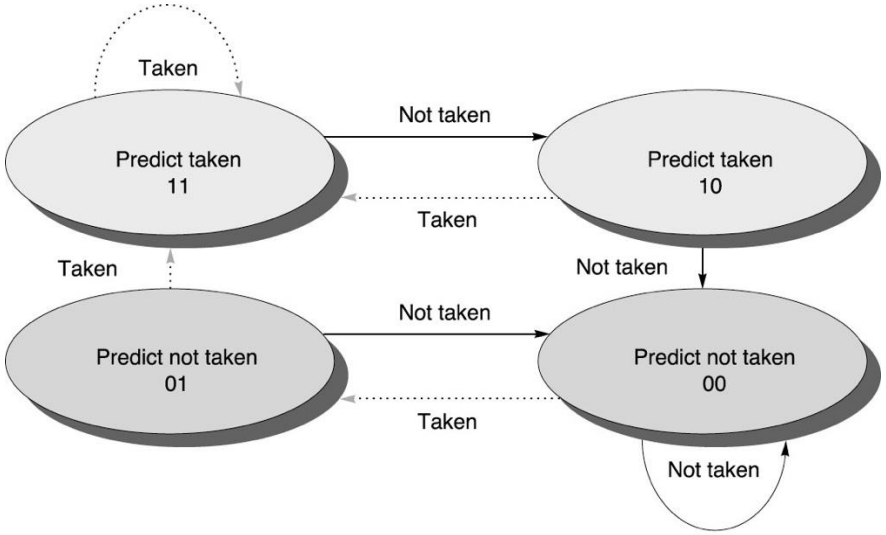
  ☹ Low prediction accuracy
    - Even if a branch is almost always taken, it will likely predict incorrectly twice, rather than twice, when it is not taken.

- Example
  - Consider a loop branch whose behavior is taken nine times in a row, then not taken once. What is the prediction accuracy for this branch?
    - Miss-predictions on the first and last loop iterations.

The prediction accuracy for this branch that is taken 90% of the time is *only 80%!*

2-bit Prediction Scheme

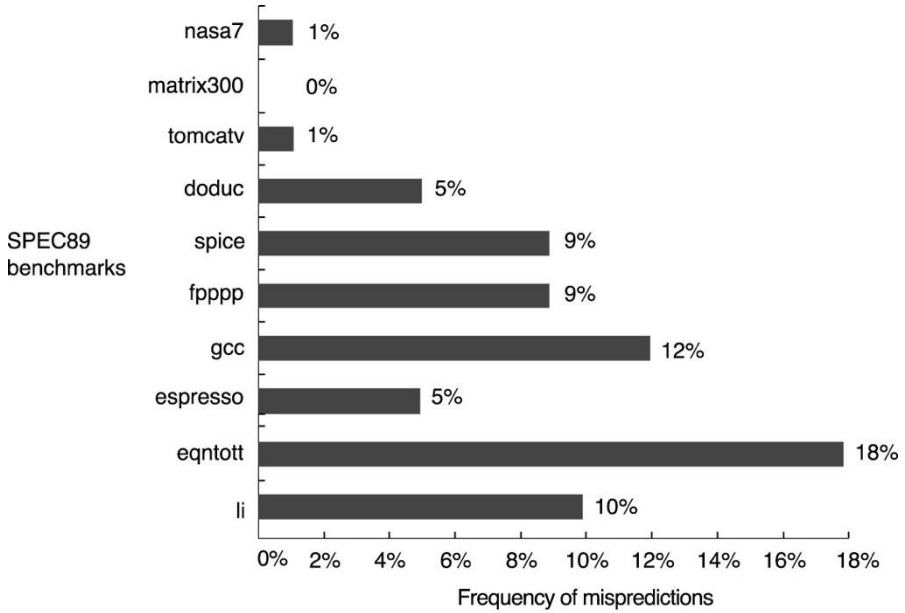
- A prediction must miss twice before it is changed.
- A 2-bit saturating counter hold the information about the recent branch behavior.
- The counter is incremented on a taken branch and decremented on an untaken branch.
- *n*-bit extension is possible, but studies of *n*-bit predictors have shown that the 2-bit predictors do almost as well.



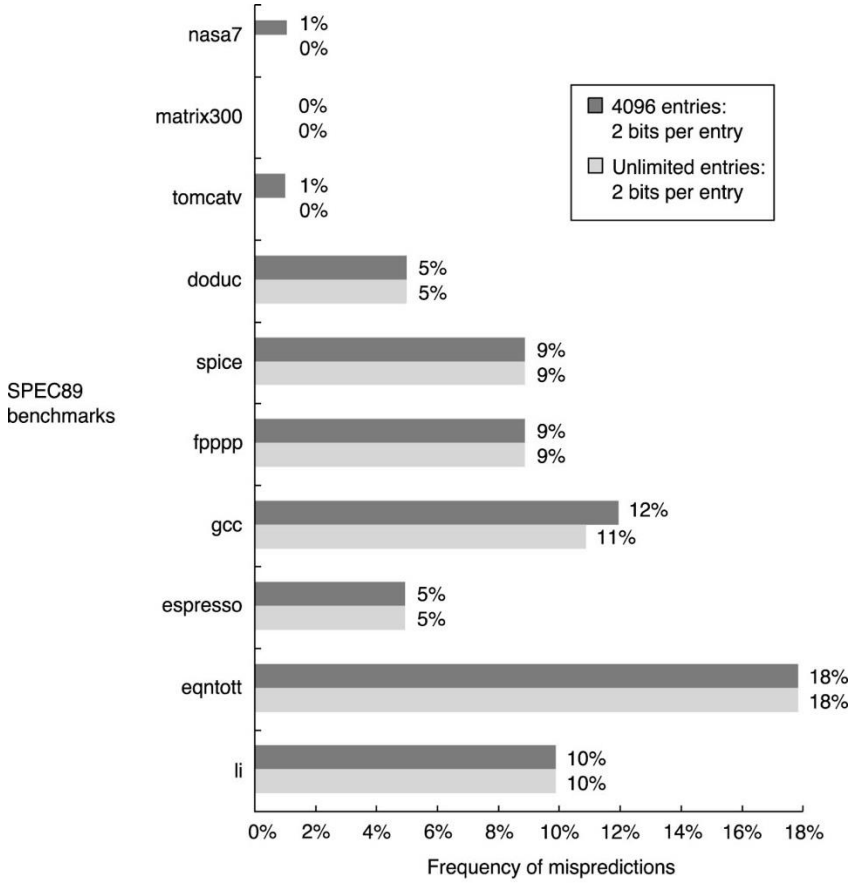
The most systems rely on 2-bit branch predictors rather than the more general *n*-bit predictors.

What Kind of Accuracy Can Be Expected From A 2-bit Branch Prediction

Performance of a branch prediction buffer with 4K entries on SPEC89



Because integer programs, *li*, *eqntott*, *espresso*, and *gcc*, have higher branch frequency, accuracy of the prediction gives a larger impact on the performance.



Prediction accuracy of a 4K-entries 2-bit prediction buffer versus an infinite buffer

How Can We Improve the Accuracy of Branch Prediction?

- Neither the number of entries, nor the size of prediction bits!
- Need to consider the correlation among branches for more accurate branch prediction!
  - 2-bit predictor schemes use only the recent behavior of a single branch to predict the future behavior of that branch

Example:

```
if (aa==2)
        aa=0;
if (bb==2)
        bb=0;
if (aa!=bb) {
```

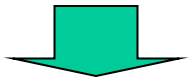
```
        DSUBUI R3,R1,#2
        BNEZ   R3,L1        ;branch b1 (aa!=2)
        DADD   R1,R0,R0     ;aa=0
L1:     DSUBI  R3,R2,#2
        BNEZ   R3,L2        ;branch b2 (bb!=2)
        DADD   R1,R0,R0     ;bb=0
L2:     DSUBI  R3,R1,R2     ;R3=aa-bb
        BNEZ   R3,L3        ;branch b3 (aa==bb)
```


TTT, TNN, NTN, NNT may occur, but the others Never occur!

1-Bit Predictors are Useless for Prediction of Correlating Branches

- Example

```
if (d==0)          d=1;
if (d==1)          …
```



```
    BNEZ    R1,L1      ;branch b1 (d!=0)
    DADDIU  R1,R0,#1   ;d==0, so d=1
L1: DADDIU  R3,R1,#-1
    BNEZ    R3,L2      ;branch b2 (d!=1)
    …
L2:
```

Possible execution sequences for a code fragment

Initial value of d	d==0?	b1	Value of d before b2	d==1?	b2
0	Yes	Not taken	1	Yes	Not taken
1	No	Taken	1	Yes	Not taken
2	No	Taken	2	No	taken

Behavior of a 1-bit predictor initialized to not taken

d=?	b1 prediction	b1 action	New b1 prediction	b2 prediction	b2 action	New b2 prediction
2	NT	T	T	NT	T	T
0	T	NT	NT	T	NT	NT
2	NT	T	T	NT	T	T
0	T	NT	NT	T	NT	NT

All the branches are mispredicted!

Correlating/Two-Level Predictors: Basic Idea

- Prediction bits are provided for each case of branch correlating patterns

Prediction bits	Prediction if last branch not taken	Prediction if last branch taken
NT/NT	NT	NT
NT/T	NT	T
T/NT	T	NT
T/T	T	T

The action of the 1bit predictor with 1 bit of correlation, initialized to not-taken/not-taken.

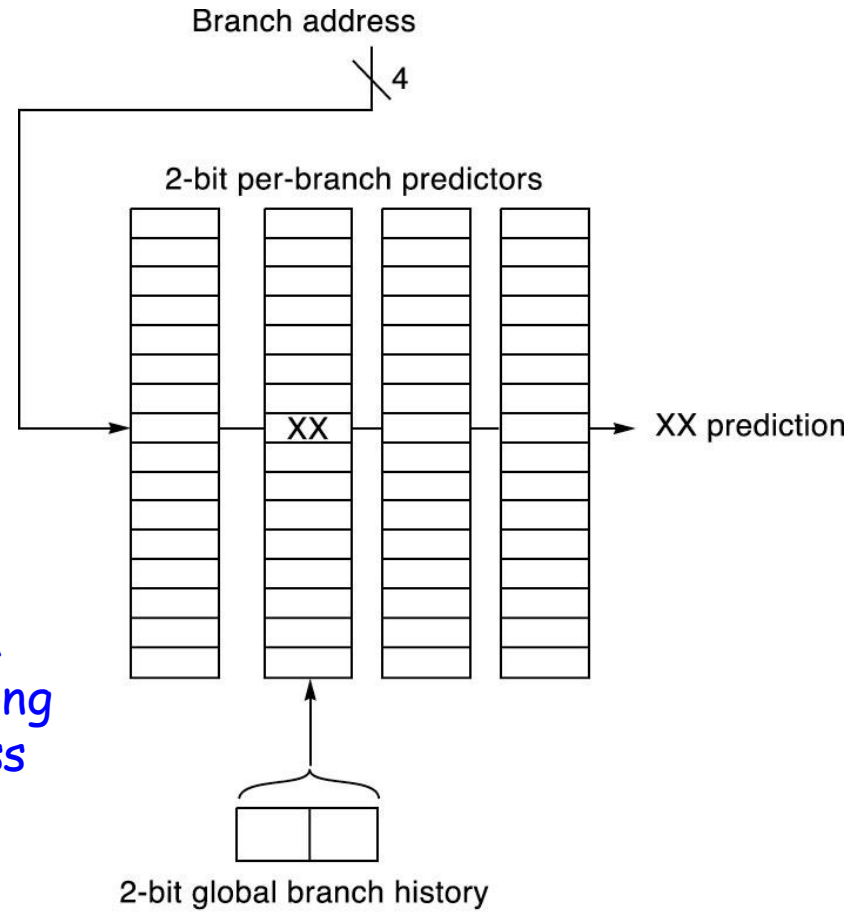
d=?	b1 prediction	b1 action	New b1 prediction	b2 prediction	b2 action	New b2 prediction
2	NT/NT	T	T/NT	NT/NT	T	NT/T
0	T/NT	NT	T/NT	NT/T	NT	NT/T
2	T/NT	T	T/NT	NT/T	T	NT/T
0	T/NT	NT	T/NT	NT/T	NT	NT/T

The only misprediction is on the first iteration!

(1,1) predictor: it uses the behavior of the last branch to choose from among a pair of 1-bit branch predictors.

(m, n) Predictors

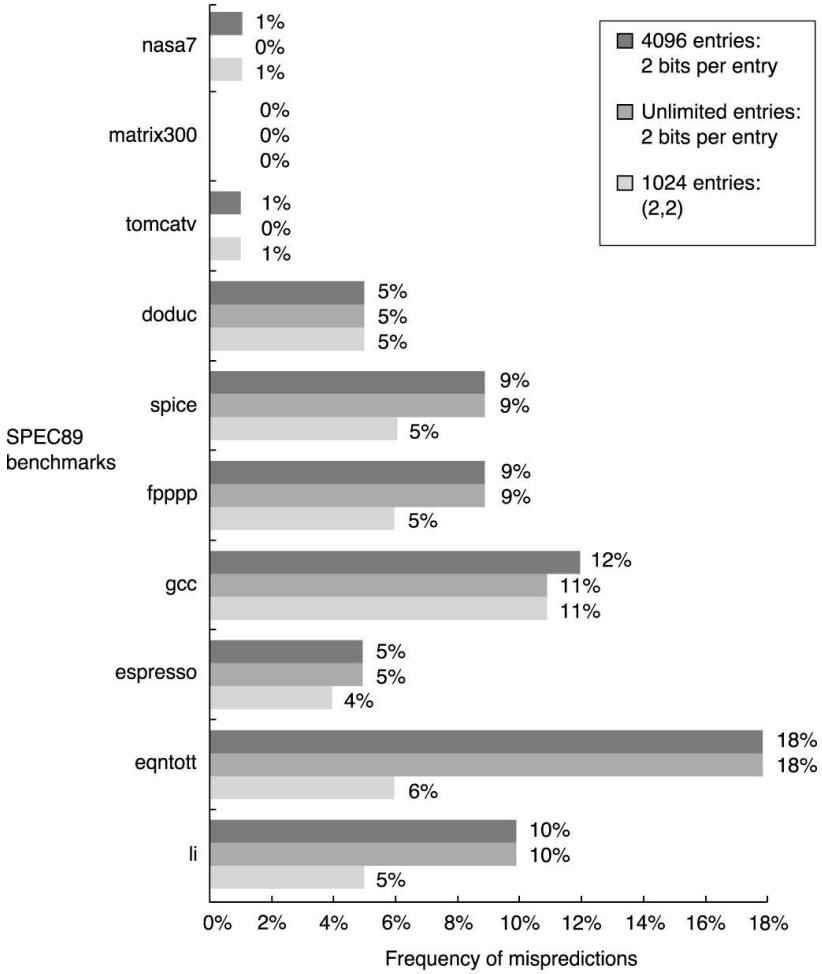
- Use the behavior of the last *m* branches to choose from 2^m branch predictors, each of which is an *n*-bit predictor for a single branch.



A (2,2) branch-prediction buffer uses a 2-bit global history to choose from among four predictors for each branch address

- A 2-bit predictor with no global history is simply a (0,2) predictor.

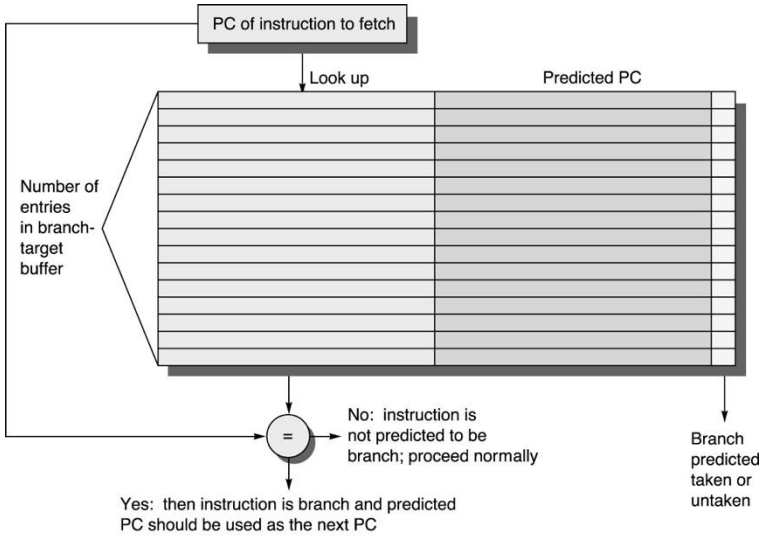
Comparison of 2-bit predictors



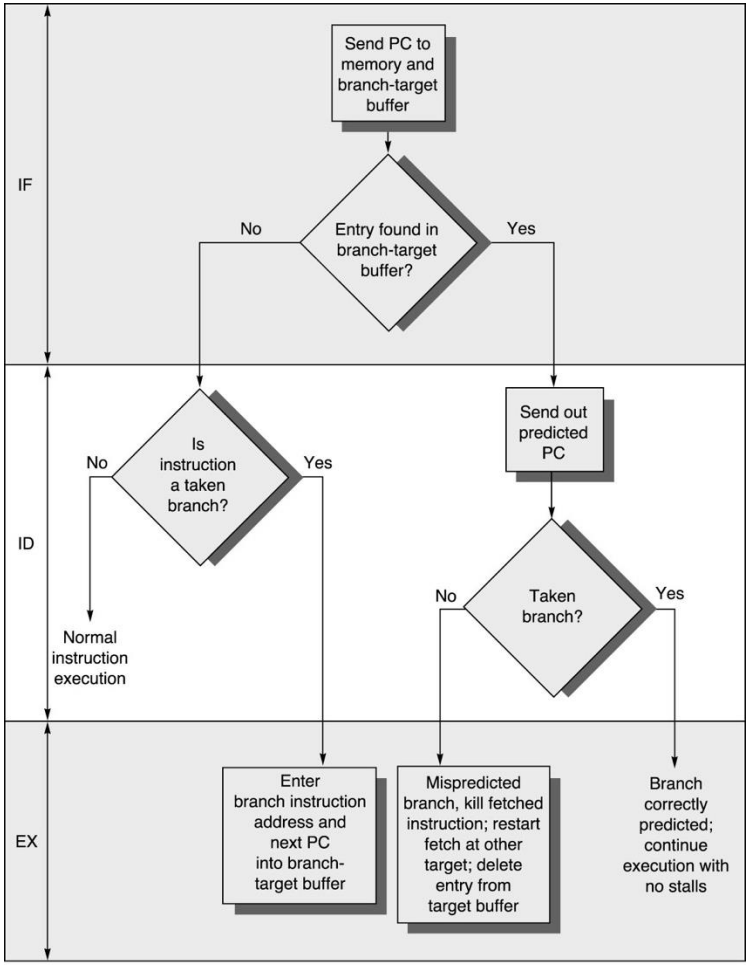
Pipelining with Branch Prediction

- To reduce the branch penalty for pipelines, need to know what address to fetch by the end of IF, *not in ID*.

Get a hint from instruction address



Branch-Target Buffer



Penalties of Branch Misprediction

Penalties for all possible combinations of whether the branch is in the buffer and what it actually does, assuming we store only taken branches in the buffer.

Instruction in buffer	Prediction	Actual branch	Penalty cycles
Yes	Taken	Taken	0
Yes	Taken	Not taken	2
No		Taken	2
No		Not taken	0

- Example

  Determine the total branch penalty for a branch-target buffer assuming the penalty cycles for individual mispredictions from the left table. Make the following assumptions about the prediction accuracy and hit rate:

  - Prediction accuracy is 90% (for instructions in buffer).
  - Hit rate in the buffer is 90% (for branches predicted taken).

  Assume that 60% of the branches are taken.

- Hints: calculate
  - Probability (branch in buffer, but actually not taken)
  - Probability (branch not in buffer, but actually taken)

  ∴ Branch penalty = (the probability of two events) x (penalty)

Taking Advantage of More ILP with Multiple Issue

- Goal: Decrease the CPI to less than one!

Allow multiple instructions to issue in a clock cycle

- Superscalar processors and
- VLIW (very long instruction word) processors

Common name	Issue structure	Hazard detection	Scheduling	Distinguishing characteristic	Example
Superscalar (static)	Dynamic	Hardware	Static	In-order execution	Sun UltraSPARC II/III
Superscalar (dynamic)	Dynamic	Hardware	Dynamic	Some out-of-order execution	IBM Power2
Superscalar (speculative)	Dynamic	Hardware	Dynamic with speculation	Out-of-order execution with speculation	Pentium III/4, MIPS R10K, Alpha 21264, HP PA 8500, IBM RS64III
VLIW/LIW	Static	Software	Static	No hazards between issue packets	Trimedia, i860
EPIC*	Mostly static	Mostly software	Mostly static	Explicit dependences marked by compiler	Itanium

Characterization of Superscalar Processors

- Number of instructions per clock (*issue width*)
  - Necessary hazard checks among up to maximum issue width of instructions must complete in one clock cycle!

- Instruction issue mechanisms
  - *Statically scheduled* using compiler techniques or
    - In-order execution: if some instruction in the instruction stream is dependent or doesn't meet the issue criteria, only the instructions preceding that one in the instruction sequence will be issued.
  - *Dynamically scheduled* using techniques based on Tomasulo's algorithm
    - Out-of-order execution: Instructions will be issued as long as any hazards do not occur.

Statically Scheduled Superscalar Processors

- Instructions issue *in order* and all pipeline hazards are checked for at issue time.

Instruction type	Pipe stages							
Integer instruction	IF	ID	EX	MEM	WB			
FP instruction	IF	ID	EX	EX	EX	WB		
Integer instruction		IF	ID	EX	MEM	WB		
FP instruction		Stall	Stall	Stall	Stall	Stall		
Integer instruction			IF	ID	EX	MEM	WB	
FP instruction			IF	ID	EX	EX	EX	WB
Integer instruction				Stall	Stall	Stall	Stall	Stall
FP instruction				Stall	Stall	Stall	Stall	Stall

Multiple Instruction Issue with Dynamic Scheduling

Goal: Instructions issue at least until the hardware runs out of reservation stations.

Example: Implementation of a two-issue dynamically scheduled processor

- Consider the execution of the following simple loop, which adds a scalar in F2 to each element of a vector in memory.

```
Loop:  LD      F0,0(R1)      ;F0=array element
       Add     F4,F0,F2      ;add scalar in F2
       SD      F4,0(R1)      ;store result
       DADDIU  R1,R1,-8      ;decrement pointer
                             ; 8 bytes
       BNE     R1,R2,LOOP    ; branch R1!=R2
```

Assumptions

- It can issue two instructions on every clock cycle: one integer operation and one FP operation
- Latencies
  - One cycle for integer ALU
  - Two cycles for loads
  - Three cycles for FP add

Clock Cycle of Issue, Execution, and Writing Results for a dual-issue Pipeline

Iteration Number	Instructions	Issue at	Executes	Memory access at	Write CDB at	Comment
1	LD F0,0(R1)	1	2	3	4	First Issue
1	ADD F4,F0,F2	1	5		8	Wait for LD
1	SD F4,0(R1)	2	3	9		Wait for ADD
1	DADDIU R1,R1,-8	2	4		5	Wait for ALU
1	BNE R1,R2,LOOP	3	6			Wait for DADDIU
2	LD F0,0(R1)	4	7	8	9	Wait for BNE complete
2	ADD F4,F0,F2	4	10		13	Wait for LD
2	SD F4,0(R1)	5	8	14		Wait for ADD
2	DADDIU R1,R1,-8	5	9		10	Wait for ALU
2	BNE R1,R2,LOOP	6	11			Wait for DADDIU
3	LD F0,0(R1)	7	12	13	14	Wait for BNE complete
3	ADD F4,F0,F2	7	15		18	Wait for LD
3	SD F4,0(R1)	8	13	19		Wait for ADD
3	DADDIU R1,R1,-8	8	14		15	Wait for ALU
3	BNE R1,R2,LOOP	9	16			Wait for DADDIU

execution completed

Resource Usage Table

Clock #	Int ALU	FP ALU	D-Cache	CDB
2	1/LD			
3	1/SD		1/LD	
4	1/DADDIU			1/LD
5		1/ADD		1/DADDIU
6				
7	2/LD			
8	2/SD		2/LD	1/ADD
9	2/DADDIU		1/SD	2/LD
10		2/ADD		2/DADDIU

Clock #	Int ALU	FP ALU	D-Cache	CDB
11				
12	3/LD			
13	3/SD		3/LD	2/ADD
14	3/DADDIU		2/SD	3/LD
15		3/ADD		3/DADDIU
16				
17				
18				3/ADD
19			3/SD	

Issue Rate =5/3=1.67, but
Instruction Complete Rate = 15/16=0.94

Integer ALU becomes a bottleneck!

3 Integer Ops & 1FP per iteration

If We Could Add One More Integer ALU…

Iteration Number	Instructions	Issue at	Executes	Memory access at	Write CDB at	Comment
1	LD F0,0(R1)	1	2	3	4	First Issue
1	ADD F4,F0,F2	1	5		8	Wait for LD
1	SD F4,0(R1)	2	3	9		Wait for ADD
1	DADDIU R1,R1,-8	2	3		4	Executes earlier
1	BNE R1,R2,LOOP	3	5			Wait for DADDIU
2	LD F0,0(R1)	4	6	7	8	Wait for BNE complete
2	ADD F4,F0,F2	4	9		12	Wait for LD
2	SD F4,0(R1)	5	7	13		Wait for ADD
2	DADDIU R1,R1,-8	5	6		7	Executes earlier
2	BNE R1,R2,LOOP	6	8			Wait for DADDIU
3	LD F0,0(R1)	7	9	10	11	Wait for BNE complete
3	ADD F4,F0,F2	7	12		15	Wait for LD
3	SD F4,0(R1)	8	10	16		Wait for ADD
3	DADDIU R1,R1,-8	8	9		10	Executes earlier
3	BNE R1,R2,LOOP	9	11			Wait for DADDIU

execution completed

Resource Usage Table

Clock #	Integer ALU	Address Adder	FP ALU	D-Cache	CDB#1	CDB#2
2		1/LD				
3	1/DADDIU	1/SD		1/LD		
4					1/LD	1/DADDIU
5			1/ADD			
6	2/DADDIU	2/LD				
7		2/SD		2/LD	2/DADDIU	
8					1/ADD	2/LD
9	3/DADDIU	3/LD	2/ADD	1/SD		
10		3/SD		3/LD	3/DADDIU	
11					3/LD	
12			3/ADD		2/ADD	
13				2/SD		
14						
15						3/ADD
16				3/SD		

Improved Instruction Complete Rate = 15/11=1.36

Factors Limiting Performance of the Two-Issue Dynamically Scheduled Pipeline

- There is an imbalance between the functional unit structure of the pipeline and example loop.
  - This imbalance means that it is impossible to fully use the FP units. To remedy this, we would need fewer dependent integer operations per loop.
- The amount of overhead per loop interaction is very high
  - Two out of five instructions (DADDIU and BNE) are overhead.
- The control hazard, which prevents us from starting the next *LD* before we know whether the branch was correctly predicted, causes a one-cycle penalty on every loop iteration.

Speculation

- Efficient handling of branch instructions is key for wide-issue superscalar processors

- Overcoming control dependence is done by *speculating* on the outcome of branches and *executing* the program as if our guesses were correct
  - ✓ Need mechanisms to handle the situation where the speculation is incorrect.

Hardware-based Speculation: Three Key Ideas

- Dynamic branch prediction to choose which instructions to execute
- Speculation to allow the execution of instructions before the control dependences are resolved (with the ability to undo the effects of an incorrectly speculated sequence), and
- Dynamic scheduling to deal with the scheduling of different combinations of basic blocks.

Hardware-based speculation follows the predicted flow of data values to choose when to execute instructions ➢ Data flow execution

Extension of Tomasulo's Algorithm for Speculation: Basic Idea

- Modern Microprocessors employ speculative execution mechanisms based on Tomasulo's Algorithm
  - PowerPC603/604/G3/G4, MIPS R10K/R12K, PentiumII/III/4, Alpha21264, AMD K5/K6/Athron

We cannot use the results of speculatively executed instructions until the instructions are no longer speculative!

A special care for speculative execution is to separate the bypassing of results among instructions, *which is needed to execute an instruction speculatively*, from the actual completion of an instruction.

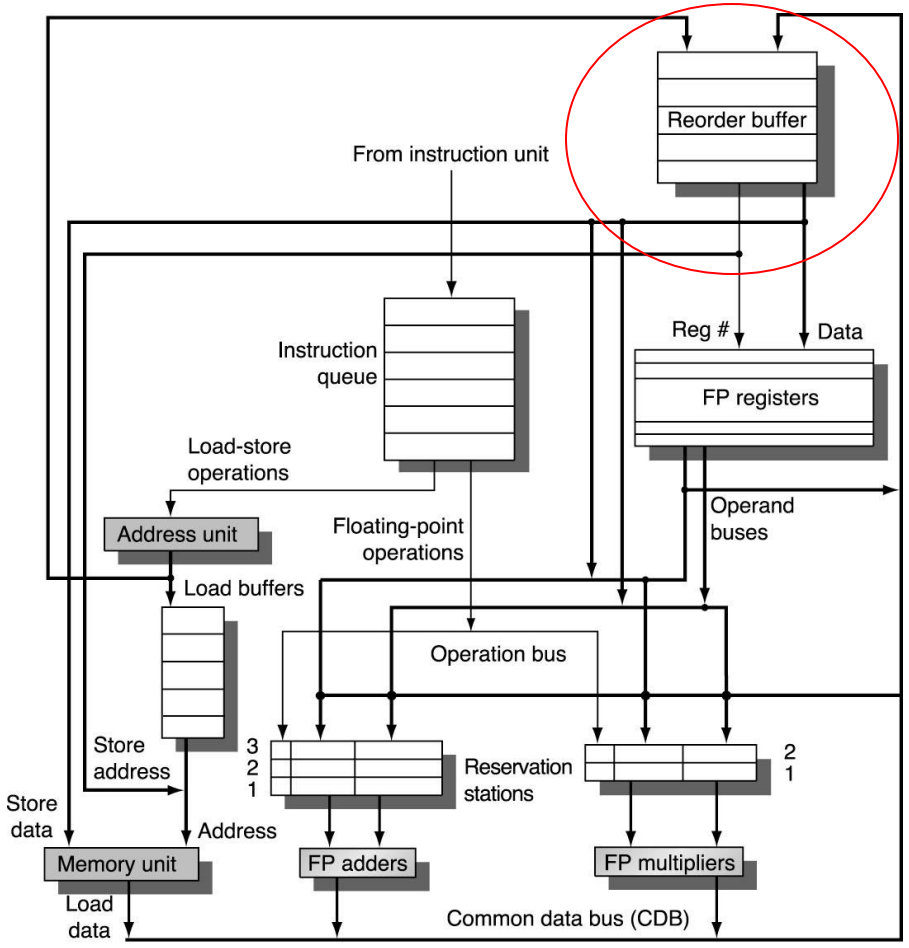
When an instruction is no longer speculative, we allow it to update the register file or memory

Instruction Commit

Implementation

- Allow instructions to *execute out-of-order* but to force them to *commit in-order,* and
- Prevent any irrevocable action* until an instruction commits.
  - ** such as updating state or taking an exception*

- Reorder Buffer: an additional set of hardware buffers that hold the results of instructions that have *finished execution but have not committed*.



The ROB supplies operands/results in the interval between completion of instruction execution and instruction commit.

Reorder Buffer: FIFO to Keep the Execution Order When the Speculation Cleared

Code segment

```
LD      F6,34(R2)
LD      F2,45(R3)
MUL     F0,F2,F4
SUB     F8,F6,F2
DIV     F10,F0,F6
ADD     F6,F8,F2
```

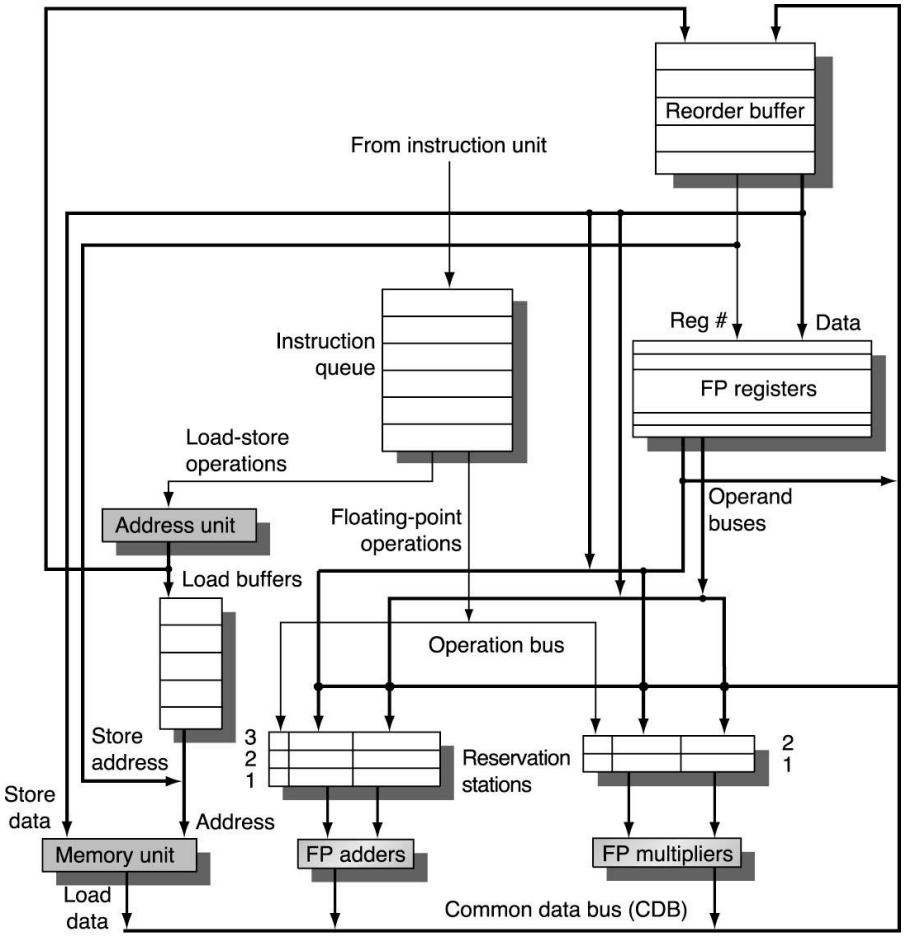
Reorder Buffer

Entry	Busy		Instruction	State	Dest	Value
1	No	LD	F6,34(R2)	Commit	F6	Mem[34+Regs[R2]]
2	No	LD	F2,45(R3)	Commit	F2	Mem[45+Reg[R3]]
3	Yes	MUL	F0,F2,F4	Write result	F0	#2 × Regs[F4]
4	Yes	SUB	F8,F6,F2	Write result	F8	#1 − #2
5	Yes	DIV	F10,F0,F6	Execute	F10	
6	Yes	ADD	F6,F8,F2	Write result	F6	#4 + #2

Pipe Stages for Speculation

Issue

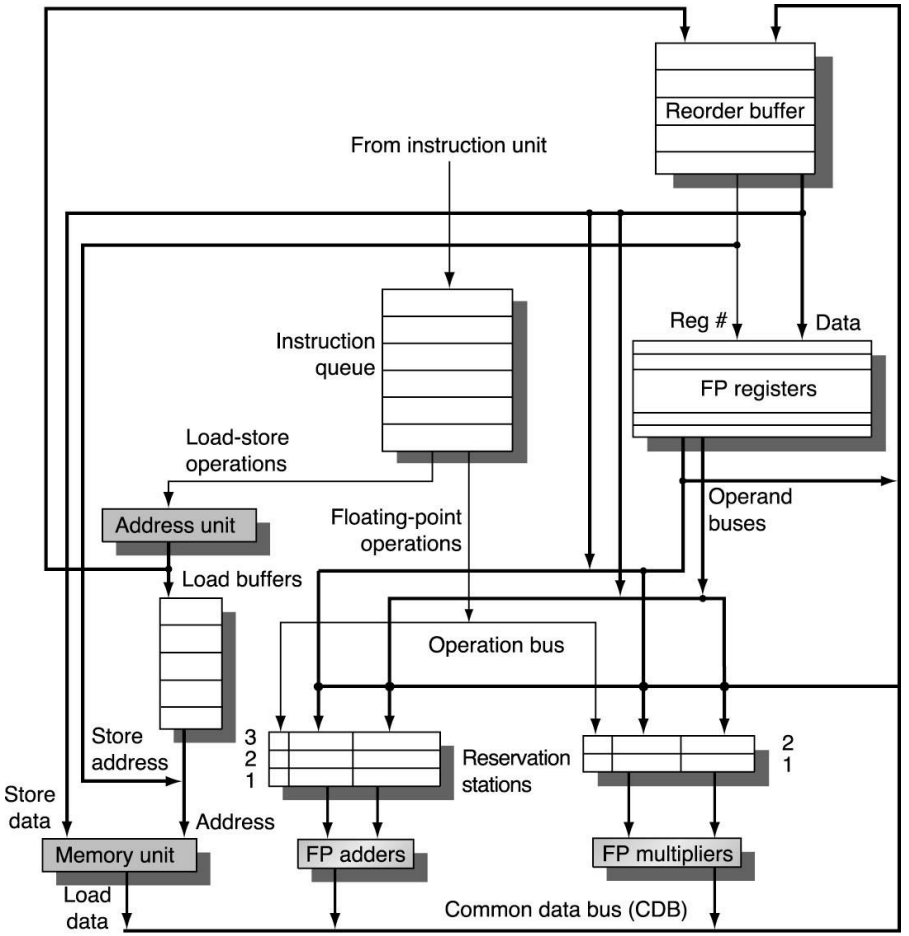
- Get an instruction from the instruction queue
- Issue the instruction if there is *an empty reservation station* and *an empty slot in the ROB*
- Send the operands to the reservation station if they are available in either the registers or the ROB
- Update the control entries to indicate the buffers are in use.
- The number of the ROB allocated for the result is also sent to the reservation station, so that the number can be used to tag the result when it is placed on the CDB
- If either all reservations are full or the ROB is full, then instruction issue is stalled until both have available entries.
- This stage is sometimes called *dispatch* in a dynamically scheduled processor



Pipe Stages for Speculation

Execute

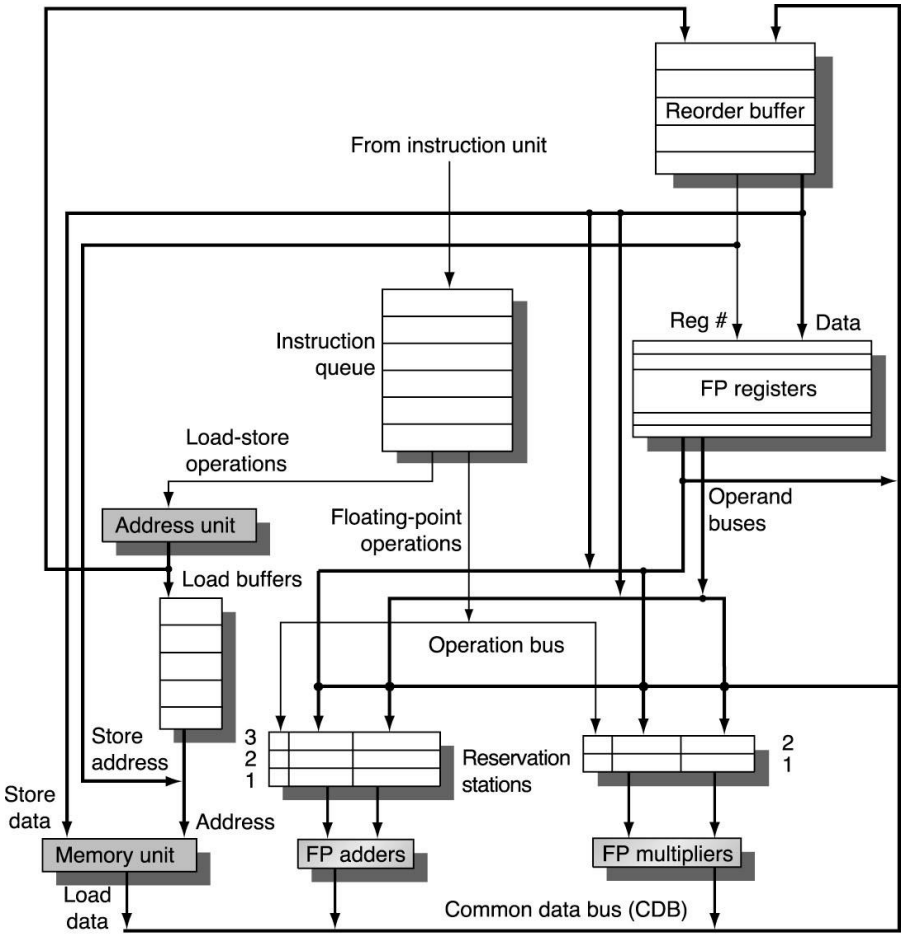
- If one or more of the operands is not yet available, monitor the CDB while waiting for the register to be computed (RAW hazards check).
- When both operands are available at a reservation station, execute the operation.
- Instructions may take multiple clock cycles in this sage, and loads still require two steps in this stage.
- Stores need only to have the base register available at this step, since execution for a store at this point is only effective address calculation.
  - *The store buffer is replaced with ROB.*



Pipe Stages for Speculation

Write result

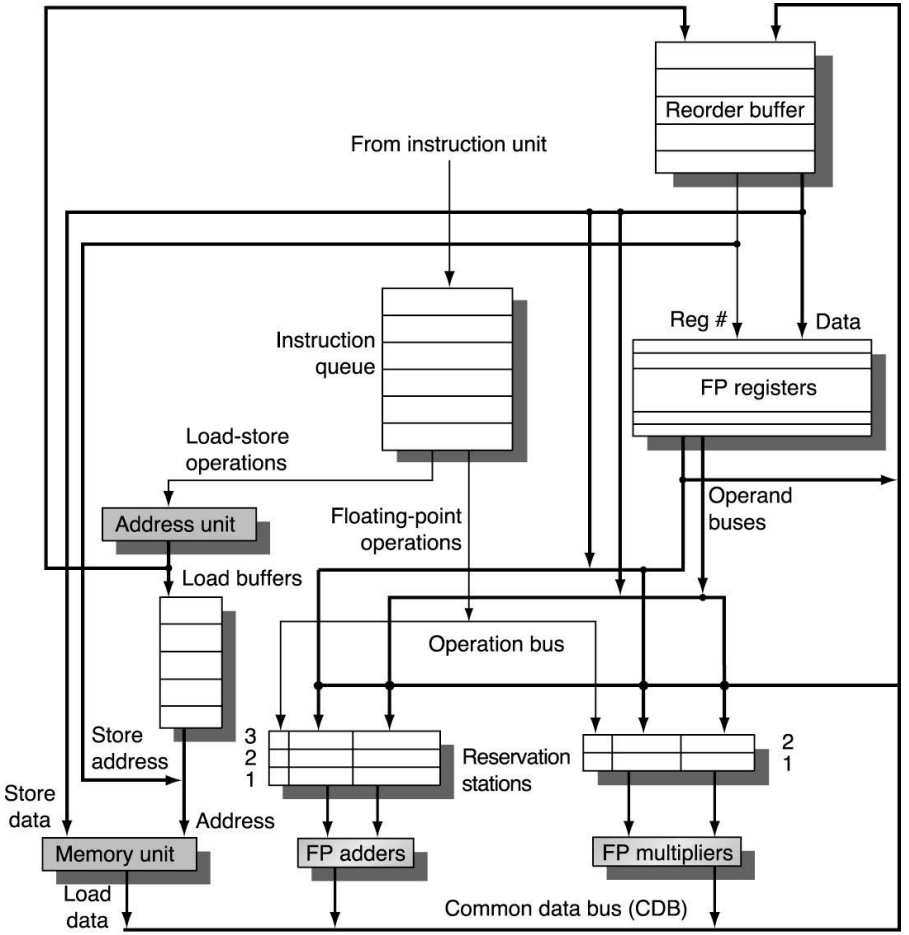
- When the result is available, write it on the CDB (with the ROB tag sent when the instruction issued) and from the CDB into the ROB, as well as to any reservation stations waiting for this result.
- Mark the reservation station as available.
- Special actions are required for store instructions.
  - If the value to be stored is available, it is written into the Value field of the ROB entry for the store.
  - If the value to be stored is not available yet, the CDB must be monitored until that value is broadcast, at which time the Value field of the ROB entry of the store is updated.



Pipe Stages for Speculation

Commit

- The normal commit case occurs when an instruction reaches the head of the ROB and its result is present in the buffer
  - At this point, the processor updates the register with the result and removes the instruction from the ROB
- Committing a store is similar except that memory is updated rather than a result register.
- When a branch with incorrect prediction reaches the head of the ROB, it indicates that the speculation was wrong. The ROB is flushed and execution is restarted at the correct successor of the branch.
- If the branch was correctly predicted, the branch is finished.
- ✓ Some machines call this commit phase "*completion*" or "*graduation*."



Once an instruction commits, its entry in the ROB is reclaimed and the register or memory destination is updated, eliminating the need for the ROB entry.

Example

Assumptions

- 2 clock cycles for add
- 10 clock cycles for multiply
- 40 clock cycles for division

Code segment

```
LD      F6,34(R2)
LD      F2,45(R3)
MUL     F0,F2,F4
SUB     F8,F6,F2
DIV     F10,F0,F6
ADD     F6,F8,F2
```

Question:

- What does the status table look like when MUL, SUB and ADD have completed, and the MUL is ready to go to commit?

Reservation Stations

Reservation Stations								
Name	Busy	Op	Vj	Vk	Qj	Qk	Dest	A
Load1	No							
Load	No							
Add1	No							
Add2	No							
Add3	No							
Multi1	No	MUL	Mem[45+Regs[R3]]	Regs[F4]			#3	
Mult2	Yes	DIV		Mem[34+Reg[R2]]	#3		#5	

Register renaming is performed in ROB instead of RS

Reorder Buffer & FP Register Status

Reorder Buffer						
Entry	Busy		Instruction	State	Dest	Value
1	No	LD	F6,34(R2)	Commit	F6	Mem[34+Regs[R2]]
2	No	LD	F2,45(R3)	Commit	F2	Mem[45+Reg[R3]]
3	Yes	MUL	F0,F2,F4	Write result	F0	#2 × Regs[F4]
4	Yes	SUB	F8,F6,F2	Write result	F8	#1 − #2
5	Yes	DIV	F10,F0,F6	Execute	F10	
6	Yes	ADD	F6,F8,F2	Write result	F6	#4 + #2

FP Register Status										
Field	F0	F1	F2	F3	F4	F5	F6	F7	F8	F10
Reorder #	3						6		4	5
Busy	Yes	No	No	No	No	No	Yes	···	Yes	Yes

Loop Unrolling with Speculation

Example

```
Loop:   LD      F0,0(R1)
        MUL     F4,F0,F2
        SD      F4,0(R1)
        DADDIU  R1,R1,-8
        BNE     R1,R2,Loop
                ;branches if R1!=R2
```

Assume

- we have issued all the instructions in the loop twice
- LD and MUL from the first iteration have committed and all other instructions have completed execution
- The store would wait in the ROB for both the effective address operand (R1) and the value (F4).
- Since we are only considering the floating point pipeline, assume the effective address for the store is computed by the time the instruction is issued.

Reorder Buffer & FP Register Status

	Reorder Buffer					
Entry	Busy	Instruction		State	Dest	Value
1	No	LD	F0,0(R1)	Commit	F0	Mem[0+Regs[R1]]
2	No	MUL	F4,F0,F2	Commit	F4	#1 × Regs[F2]
3	Yes	SD	F4,0(R1)	Write result	0+Regs[R1]	#2
4	Yes	DADDIU	R1,R1,-8	Write result	F1	Regs[R1]-8
5	Yes	BNE	R1,R2,Loop	Write result		
6	Yes	LD	F0,0(R1)	Write result	F0	Mem[#4]
7	Yes	MUL	F4,F0,F2	Write result	F4	#6 × Regs[F2]
8	Yes	SD	F4,0(R1)	Write result	0+#4	#7
9	Yes	DADDIU	R1,R1,-8	Write result	R1	#4－8
10	Yes	BNE	R1,R2,Loop	Write result		

	FP Register Status								
Field	F0	F1	F2	F3	F4	F5	F6	F7	F8
Reorder #	6				7				
Busy	Yes	No	No	No	Yes	No	No	・・・	No

Multiple Issue with Speculation

Process multiple instructions per clock assigning reservation stations and reorder buffers to the instructions.

- Instructions issue and monitoring the CDBs for instruction completion
- Multiple commits per clock cycle

become the major challenges for multiple issue with speculation.

Example

```
Loop:   LD      R2,0(R1)    ;R2=array element
        DADDIU  R2,R2,#1    ;increment R2
        SD      R2,0(R1)    ;store result
        DADDIU  R1,R1,#4    ;increment pointer
        BNE     R2,R3,Loop  ;branch if not last element
```

Assume that

- *there are separate integer FUs for effective address calculation, for ALU operations, and for branch condition evaluation, and*
- *up to two instructions of any type can commit per clock.*

Create a table for the first three iterations of this loop for machines with and without speculation.

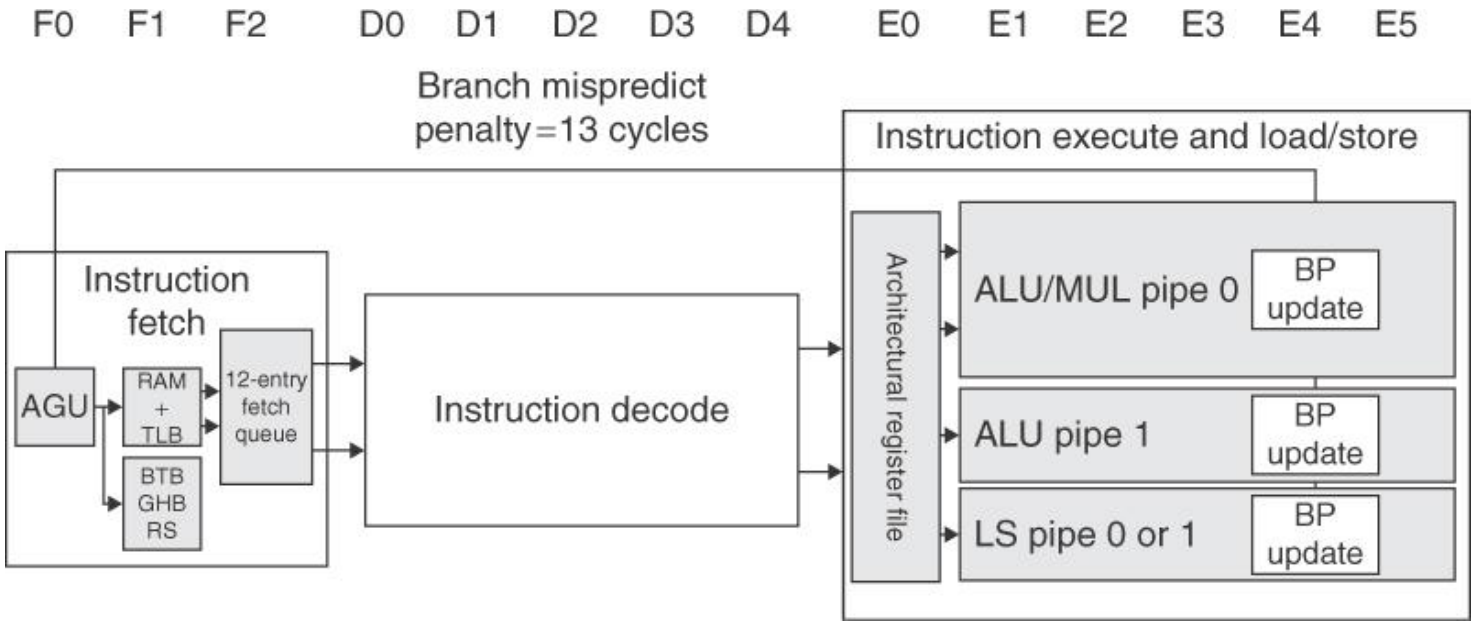
Time of Issue, Execution and Writing Result for a Dual-Issue Pipeline without Speculation

Iteration #	Instructions		Issues at clock cycle #	Executes at clock cycle #	Memory access at clock cycle #	Write CDB at clock cycle #	Comment
1	LD	R2,0(R1)	1	2	3	4	First Issue
1	DADDIU	R2,R2,#1	1	5		6	Wait for LW
1	ST	R2,0(R1)	2	3	7		Wait for DADDIU
1	DADDIU	R1,R1,#4	2	3		4	Execute directly
1	BNE	R2,R3,Loop	3	7			Wait for DADDIU
2	LD	R2,0(R1)	4	8	9	10	Wait for BNE
2	DADDIU	R2,R2,#1	4	11		12	Wait for LW
2	ST	R2,0(R1)	5	9	13		Wait for DADDIU
2	DADDIU	R1,R1,#4	5	8		9	Wait for BNE
2	BNE	R2,R3,Loop	6	13			Wait for DADDIU
3	LD	R2,0(R1)	7	14	15	16	Wait for BNE
3	DADDIU	R2,R2,#1	7	17		18	Wait for LW
3	ST	R2,0(R1)	8	15	19		Wait for DADDIU
3	DADDIU	R1,R1,#4	8	14		15	Wait for BNE
3	BNE	R2,R3,Loop	9	19			Wait for DADDIU

Time of Issue, Execution and Writing Result for a Dual-Issue Pipeline with Speculation

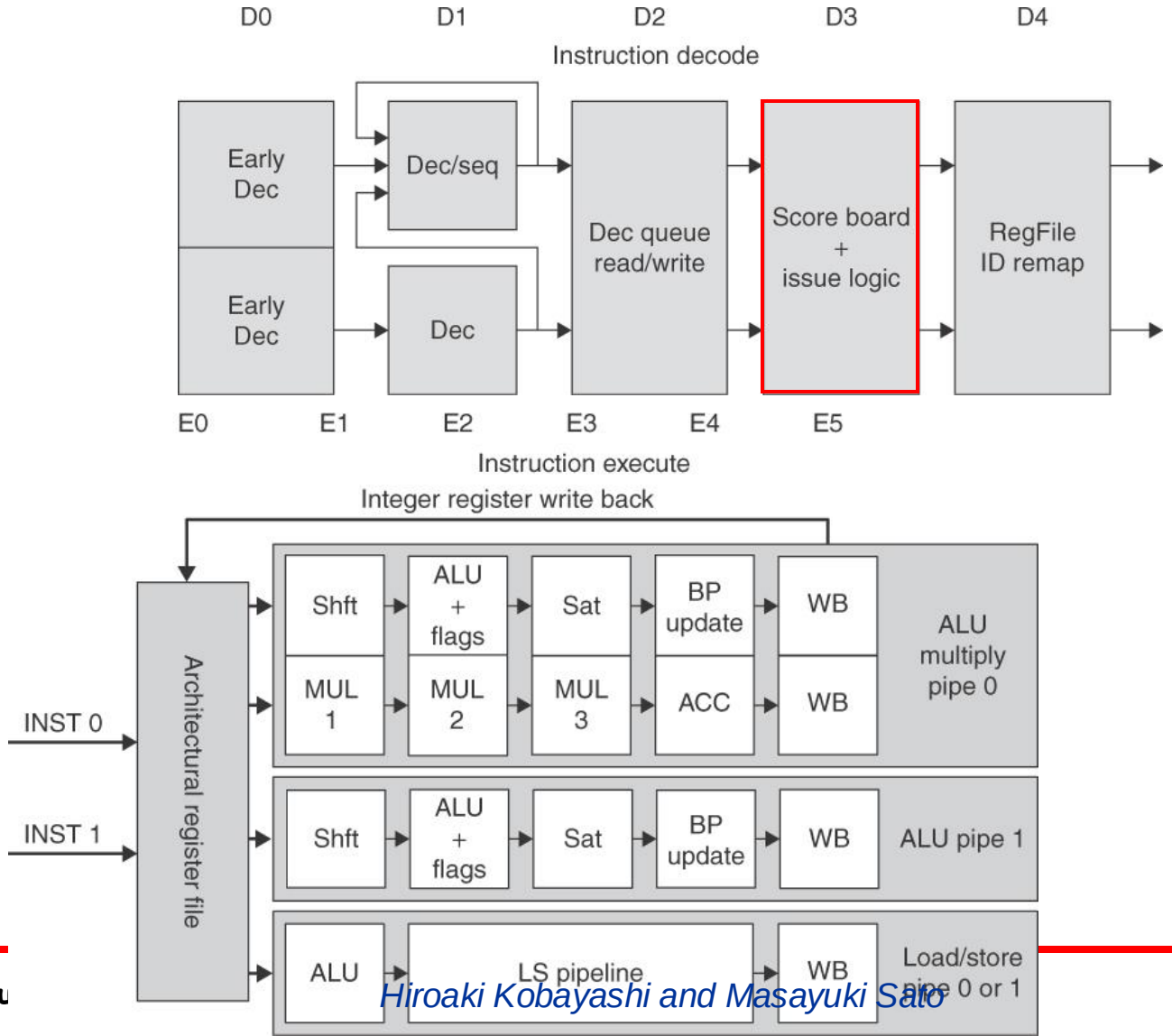
Iteration #	Instructions		Issues at clock cycle #	Executes at clock cycle #	Memory access at clock #	Write CDB at clock #	Commits at clock #	Comments
1	LD	R2,0(R1)	1	2	3	4	5	First issue
1	DADDIU	R2,R2,#1	1	5		6	7	Wait for LW
1	ST	R2,0(R1)	2	3			7	Wait for DADDIU
1	DADDIU	R1,R1,#4	2	3		4	8	Commit in order
1	BNE	R2,R3,Loop	3	7			8	Wait for DADDIU
2	LD	R2,0(R1)	4	5	6	7	9	No execute delay
2	DADDIU	R2,R2,#1	4	8		9	10	Wait for LW
2	ST	R2,0(R1)	5	6			10	Wait for DADDIU
2	DADDIU	R1,R1,#4	5	6		7	11	Commit in order
2	BNE	R2,R3,Loop	6	10			11	Wait for DADDIU
3	LD	R2,0(R1)	7	8	9	10	12	Earliest possible
3	DADDIU	R2,R2,#1	7	11		12	13	Wait for LW
3	ST	R2,0(R1)	8	9			13	Wait for DADDIU
3	DADDIU	R1,R1,#4	8	9		10	14	Execute earlier
3	BNE	R2,R3,Loop	9	13			14	Wait for DADDIU

The ARM Cortex-A8



- Dual-issue, statically scheduled superscalar
- 512-entry two-way set-associative BTB & 4K-entry GHB
- Instruction issue controlled by scoreboard
- 3 types of execution pipes: ALU/MUL, ALU, and L/S
- 13-cycle branch misspredict penalty

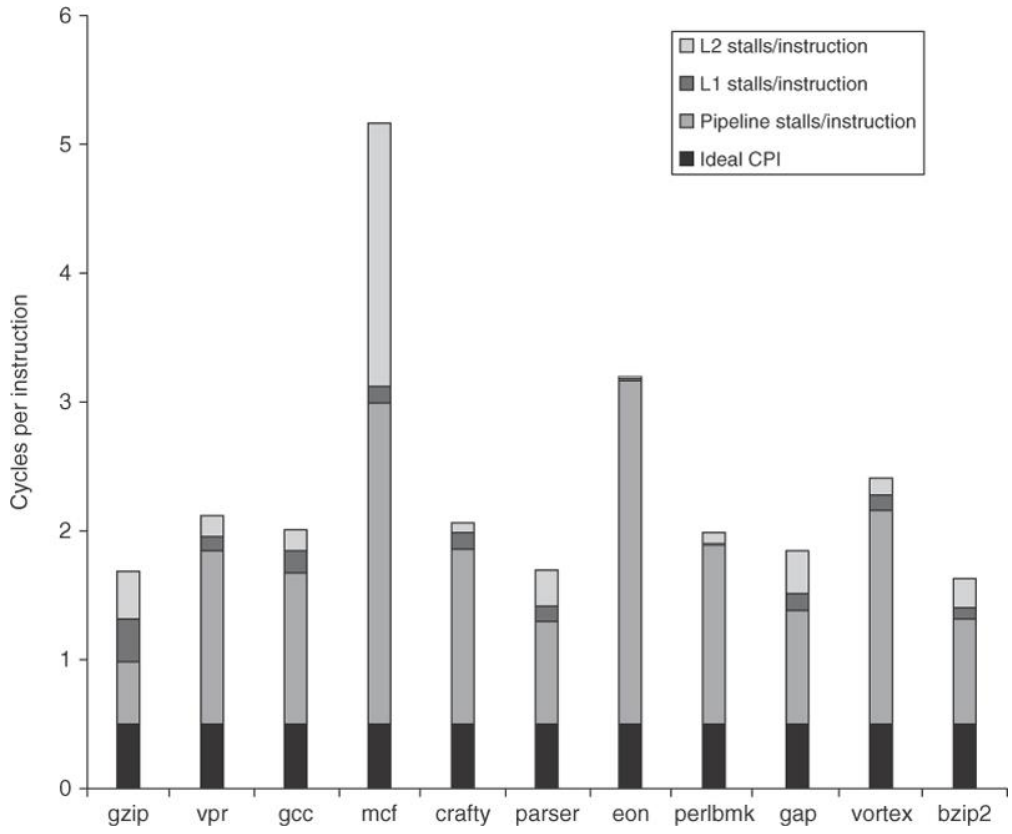
Decode and Execution



Performance of A8

Pipeline Stalls are the major contributor to the CPI, which arise from

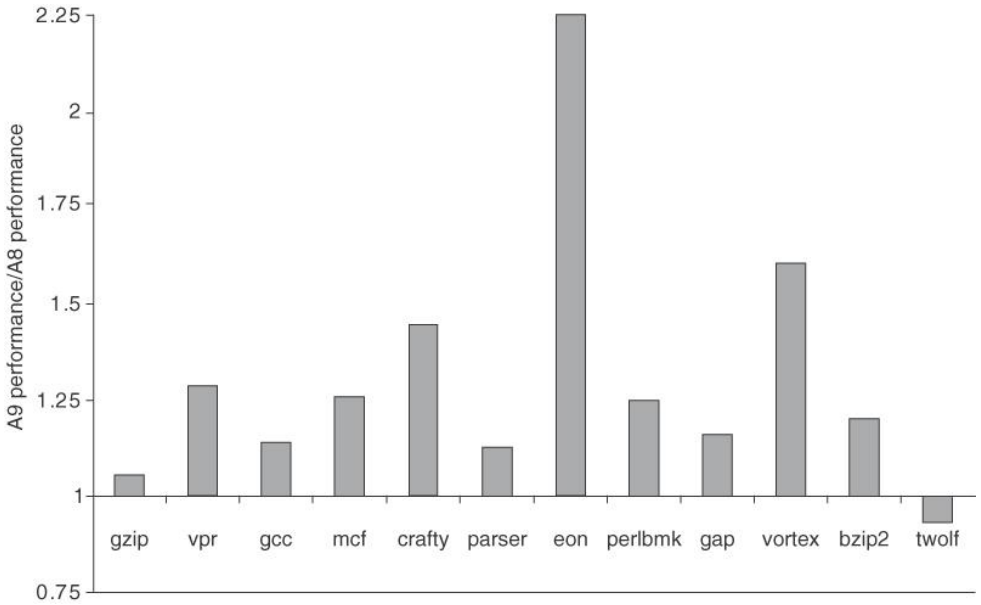
- Functional hazards
  - Two adjacent instructions selected for simultaneous execution use the same functional pipeline
- Data hazards
  - Stall both insts. or second of a pair
- Control hazards
  - Branch mispredicted



A8 vs. A9

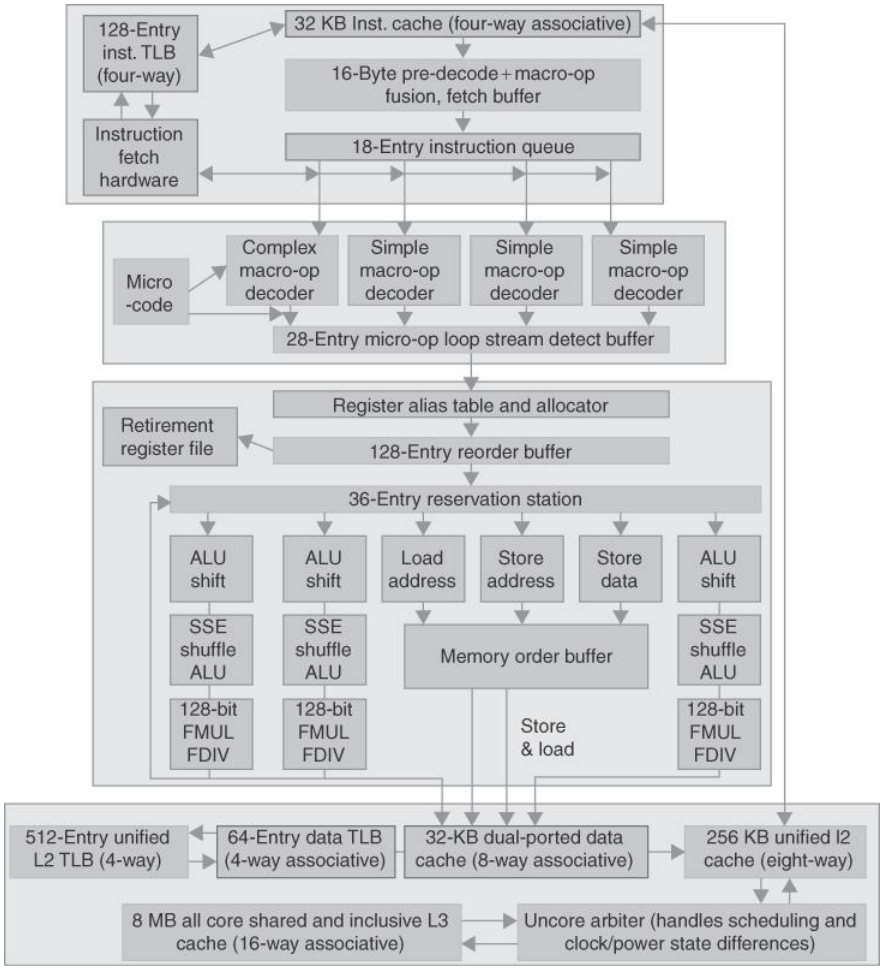
A9 :

- Dynamically scheduled superscalar
  - Two instructions per clock
- Up to four pending instructions can begin execution in a clock cycle
  - Two ALU, one L/S or FP/multimedia, and one branch
  - More powerful branch predictor, instruction cache prefetch and non-blocking L1 data cache

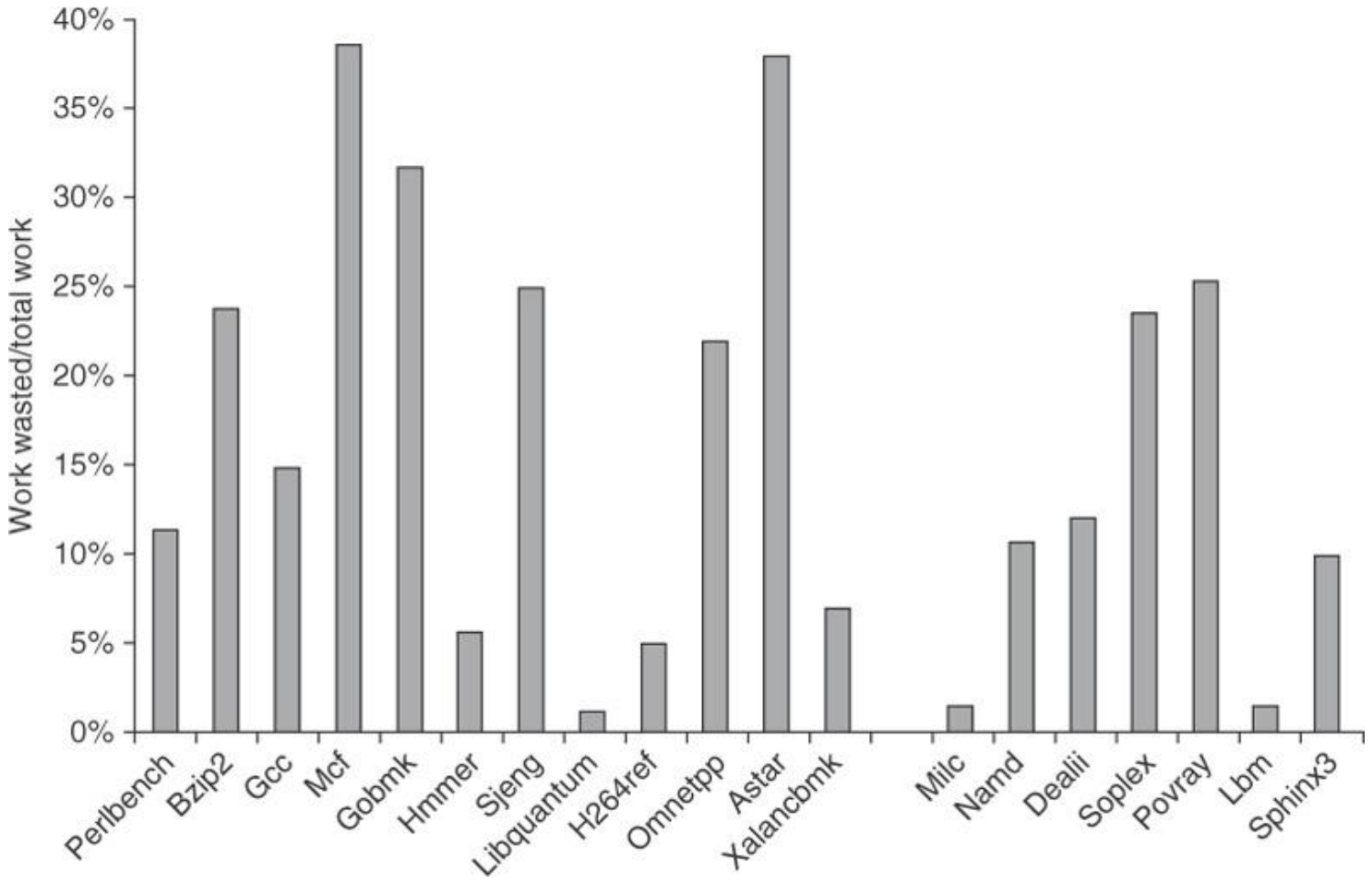


The Intel Core i7

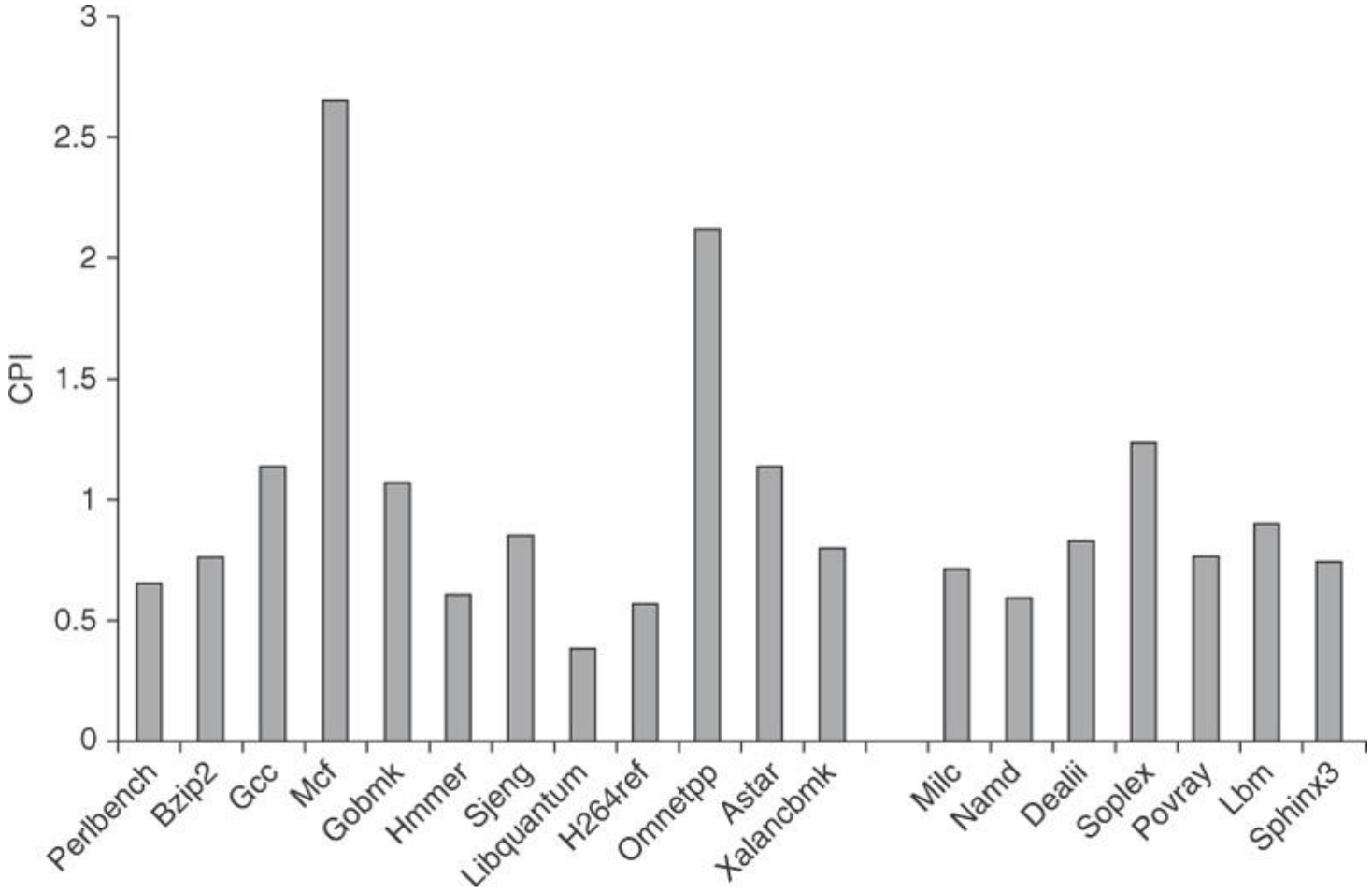
- High throughput by combining multiple issue and high clock rates
  - Out of order speculative microarchitecture
  - Deep pipeline
    - Branch misprediction penalty of 15 cycles
- Micro-op engine
  - X86 instructions are translated into micro-ops
    - MIPS-like instructions
- Centralized reservation station



Ratio of Wasted Work in Speculation



CPI for SPECCPU2006 on Core i7



Exploiting Instruction-Level Parallelism with Software Approaches

Compiler Techniques for Exposing ILP

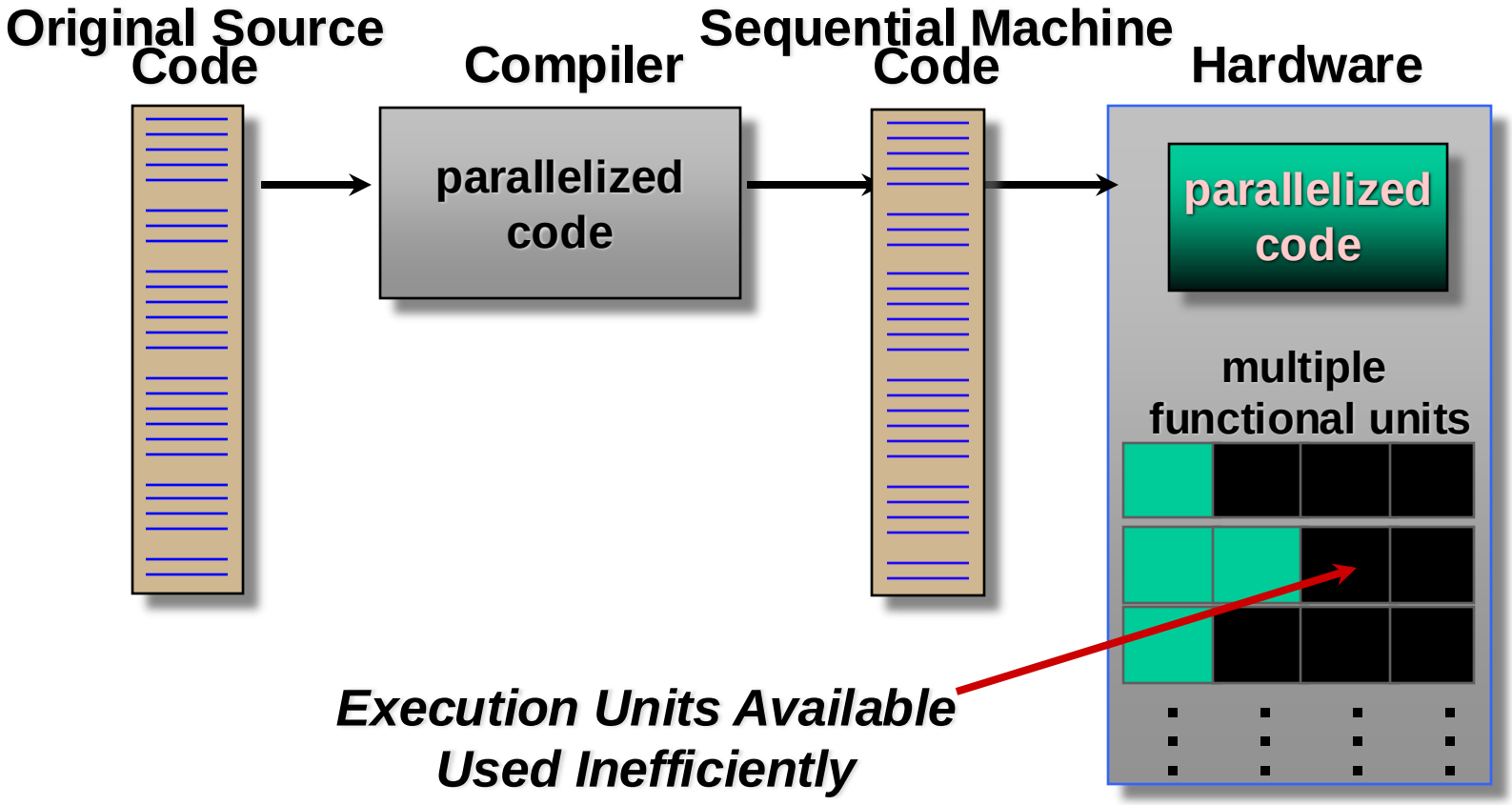
- Static Multiple Issue: The VLIW Approach
- Software Techniques for Increasing ILP
  - Loop Unrolling
  - Static Branch Prediction
  - Software Pipelining
  - Global Code Scheduling
    - Trace Scheduling
    - Superblock Scheduling
  - Conditional or Predicated Instructions

Static Multiple Issue: The VLIW Approach

- VLIW (Very Long Instruction Word) processors
  - Rely on compiler technology not only to minimize the potential data hazard stalls, but to actually format the instructions in a potential issue packet so that the hardware need not check explicitly for dependences.
    - ⇔ *Statically scheduled superscalar processors*

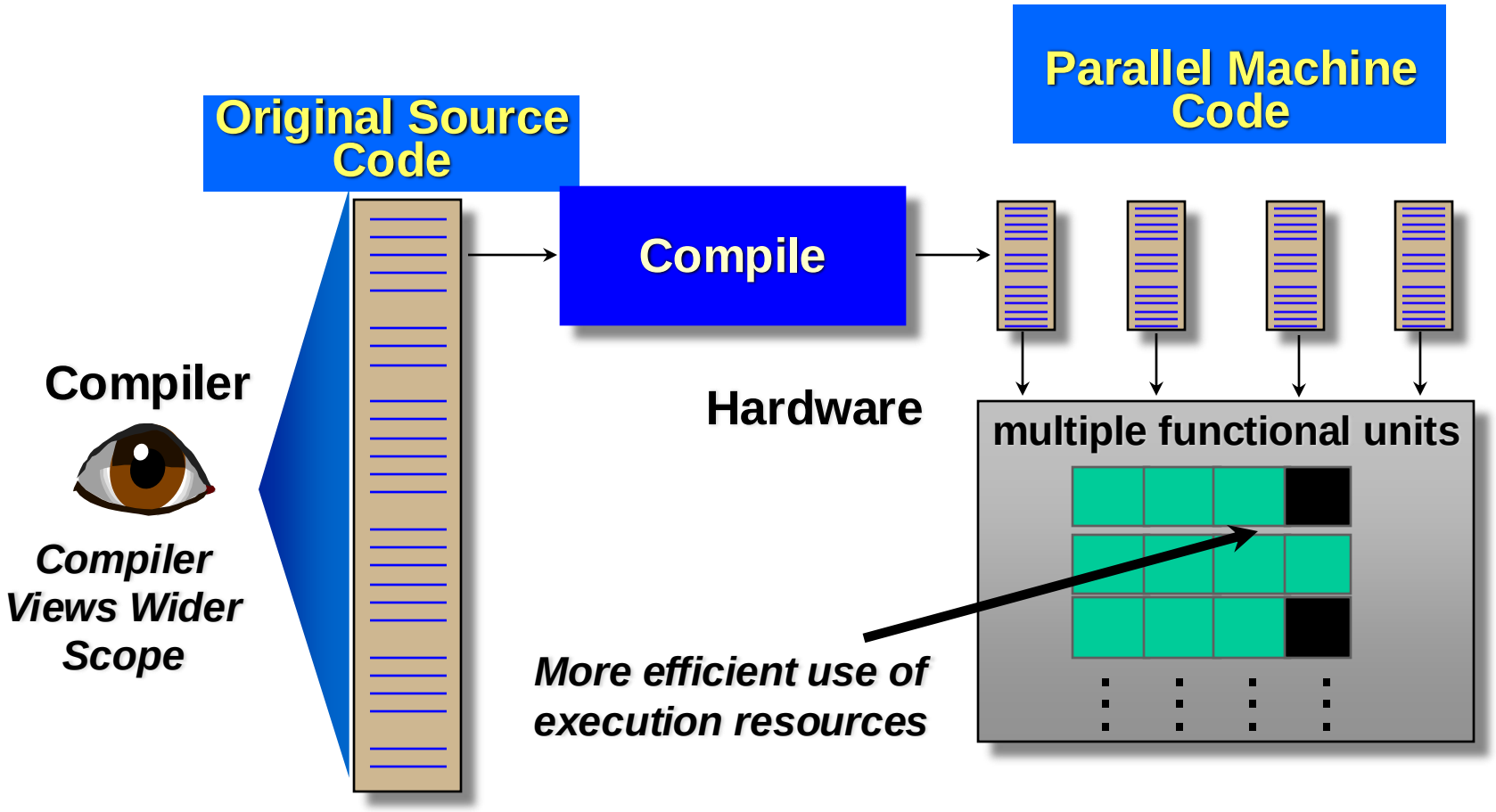
  - ✓ Simpler hardware for instruction issue mechanisms, while still exhibiting good performance through extensive compiler optimization

Traditional Architectures: Limited Parallelism



Today's Processors often 60% Idle

VLIW Architecture: Explicit Parallelism



Increases Parallel Execution

The Basic VLIW Approach

- VLIW processors
  - Multiple, independent functional units are employed, but multiple independent instructions are not issued. Instead,
  - A VLIW packages the multiple independent operations into one very long instruction, and a single long instruction is issued.
    - An instruction length of between 112 and 168 bits, 16 to 24 bits each operation; 5 to 6 operations packed into a VLIW instruction
  - Since the burden for choosing the instructions to be issued simultaneously falls on the compiler, the hardware in a superscalar to make these issue decisions is unneeded.
    - Simple hardware
    - Wide scope for exploiting the ILP

VLIW Code Example

- Suppose a VLIW processor with 5 FUs for two memory references, two FP operations, and one integer operation or branch in every clock cycle.
- Show an unrolled version of the loop x[i]=x[i]+s

Memory reference 1	Memory reference 2	FP operation 1	FP operation 2	Integer operation/branch
LD F0,0(R1)	LD F6,-8(R1)			
LD F10,-16(R1)	LD F14, -24(R1)			
LD F18,-32(R1)	LD F22, -40(R1)	ADD F4,F0,F2	ADD F8,F6,F2	
LD F26,-48(R1)		ADD F12,F10,F2	ADD F16,F14,F2	
		ADD F20,F18,F2	ADD F24,F22,F2	
ST F4,0(R1)	ST F8,-8(R1)	ADD F28,F26,F2		
ST F12,-16(R1)	ST F16,-24(R1)			DADDUI R1,R1,#-56
ST F20,24(R1)	ST F24,16(R1)			
ST F28,8(R1)				BNE R1, R2, Loop

Empty slot

Unrolled 7 times: 9 cycles for seven iterations (23 operations), 1.29 cycles per iteration (2.5 operations per cycle)

Technical and Logistical Problems of the VLIW Model

- Increase in code size
  - Ambitiously unrolling loops is required to fill operation slots with effective operations as many as possible, but also increases code size
  - Inefficient use of FUs
    - 60% of the FUs in the example used

- Compressed format of VLIWs
  - Instructions are stored in a compressed form without no-operations, and expanded when they are read into the cache or are decoded

- Operations in lockstep
  - No hazard detection hardware at all, and a stall in any functional unit pipeline must cause the entire processor to stall.
  - Recent VLIW processors employ functional units that can operate independently with hardware mechanisms for hazard detections
- Binary code compatibility
  - Different numbers of functional units and unit latencies require different versions of the code, and it makes migration between successive different implementations very difficult.
  - Binary translation (code morphing), emulation

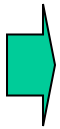
Exploiting ILP by Compiler: Basic Pipeline Scheduling

- Fill stall slots of a pipeline with independent instructions to keep the pipeline full.

Example:

Source

```
for (i=1000; i>0; i=i-1)
     x[i] = x[i] + s;
```



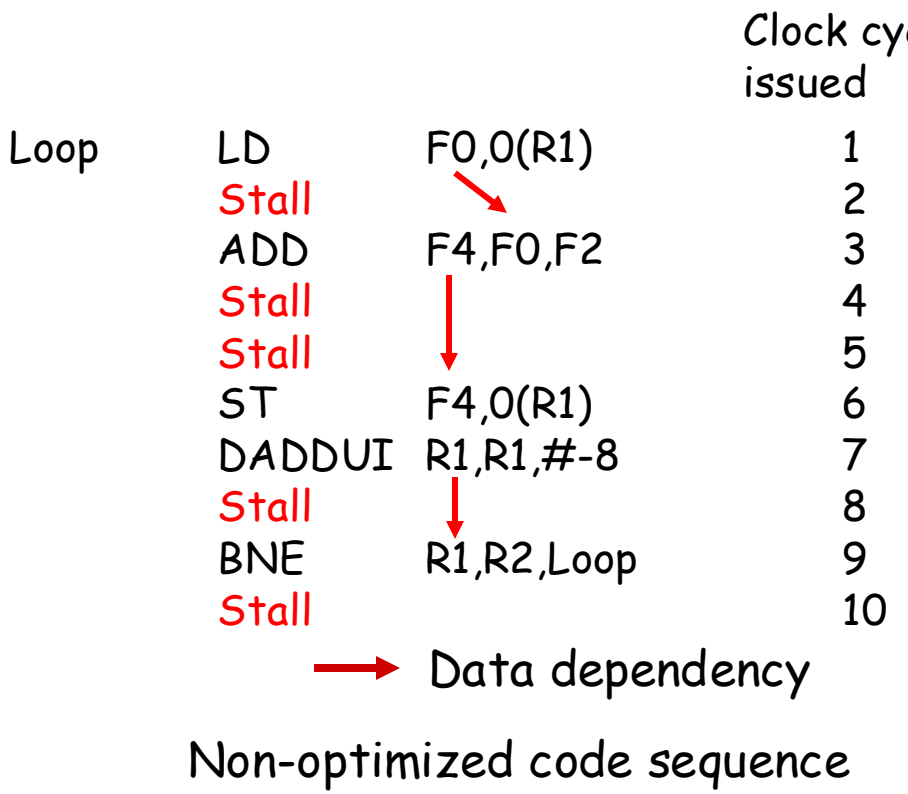
MIPS object code

```
Loop    LD      F0,0(R1)    ;F0=array element
        ADD     F4,F0,F2    ;add scalar in F2
        ST      F4,0(R1)    ;store result
        DADDUI  R1,R1,#-8   ;decrement pointer
                            ;8 bytes (per DW)
        BNE     R1,R2,Loop  ;branch R1!=R2
```

Basic Pipeline Scheduling (Cont'd)

Instruction Producing result	Instruction using result	Latency in clock cycle
FP ALU op	Another FP ALU op	+3
FP ALU op	Store double	+2
Load double	FP ALU op	+1
Load double	Store double	+0

Latencies of FP operations assumed



Non-optimized code sequence

Basic Pipeline Scheduling (Cont'd)

```
Loop    LD      F0,0(R1)
        Stall
        ADD     F4,F0,F2
        Stall
        Stall
        ST      F4,0(R1)
        DADDUI  R1,R1,#-8
        Stall
        BNE     R1,R2,Loop
        Stall
```

Floating point operations

Loop counter calculations

Adding a scalar to a vector in each iteration can be processed during calculation for the next loop counter calculation

```
Loop    LD      F0,0(R1)
        DADDUI  R1,R1,#-8
        ADD     F4,F0,F2
        Stall
        BNE     R1,R2,Loop
        ST      F4,8(R1)
```

Reduction of 10 cycles to 6!

Loop Unrolling

- In the example, a critical path of a LD-ADD-Store chain needs at least 6 cycles because of dependencies and pipeline latencies.

- But the actual work of operating on the array element takes just 3 of those 6 clock cycles.
  - Remaining 3 clock cycles consists of loop overhead and stall!

- Loop Unrolling
  - Enhance ILP and reduce the penalties of the control hazard
  - Simply replicates the loop body multiple times, adjusting the loop termination code
  - Register-reallocation is required to solve register conflicts among unrolled iteration bodies

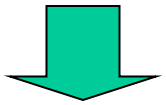
Loop Unrolling Example

- Unrolling 4 times

Loop	LD	F0,0(R1)	**Iteration 1**
	ADD	F4,F0,F2	
	ST	F4,0(R1)	;drop DADDUI&BNE
	LD	F0,-8(R1)	**Iteration 2**
	ADD	F4,F0,F2	
	ST	F4,-8(R1)	;drop DADDUI&BNE
	LD	F0,-16(R1)	**Iteration 3**
	ADD	F4,F0,F2	
	ST	F4,-16(R1)	;drop DADDUI&BNE
	LD	F0,-24(R1)	**Iteration 4**
	ADD	F4,F0,F2	
	ST	F4,-24(R1)	
	DADDUI	R1,R1,#-32	**Loop overhead**
	BNE	R1,R2,Loop	

Three branches and three decrement of R1 eliminated,

But 28(=6x4+2+2) cycles required if no scheduling adopted!



Rescheduling by using independent instructions in unrolled iterations

Loop Unrolling and Scheduling Example

```
Loop   LD      F0,0(R1)
       LD      F6,-8(R1)
       LD      F10,-16(R1)
       LD      F14,-24(R1)
       ADD     F4,F0,F2
       ADD     F8,F6,F2
       ADD     F12,F10,F2
       ADD     F16,F14,F2
       ST      F4,0(R1)
       ST      F8,-8(R1)
       DADDUI  R1,R1,#-32      ;R1=R1-32
       ST      F12,16(R1)      ;16-32=-16
       BNE     R1,R2,Loop
       ST      F16,8(R1)       ;8-32=-24
```

Enough intervals for avoiding stalls!

Rescheduling by using independent instructions in unrolled iterations

- Register renaming to avoid WAR and WAW hazards
- Index adjustment for data references and loop counting

Reduction of 28 cycles to 14, 3.5 cycles per element!

Summary of the Loop Unrolling and Scheduling Example

- The key to most of the techniques that allow us to take advantage of ILP to fully utilize the potential of the function units in a processor is *to know when and how the ordering among instructions may be changed.*

In the case of the example, we had to make the following decisions and transformation

1. Determine that it was legal to move the ST after the DADDUI and BNE, and find the amount to adjust the ST offset.
2. Determine that unrolling the loop would be useful by finding that the loop iterations were independent, except for the loop maintenance code.
3. Use different registers to avoid unnecessary constraints that would be forced by using the same register for different computations
4. Eliminate the extra test and branch instructions and adjust the loop termination and iteration code.
5. Determine that the loads and stores in the unrolled loop can be interchanged by observing that the loads and stores from different iterations are independent. This transformation requires analyzing the memory addresses and finding that they do not refer to the same address
6. Schedule the code, preserving any dependences needed to yield the same result as the original code.

Delayed Branches and Loads

- Delayed Branches and Loads
  - Its effectiveness partly depends on whether we correctly guess which way a branch will go.

Case 1: The branch was almost always taken and the value of R7 was no needed on the fall-through path,

Example

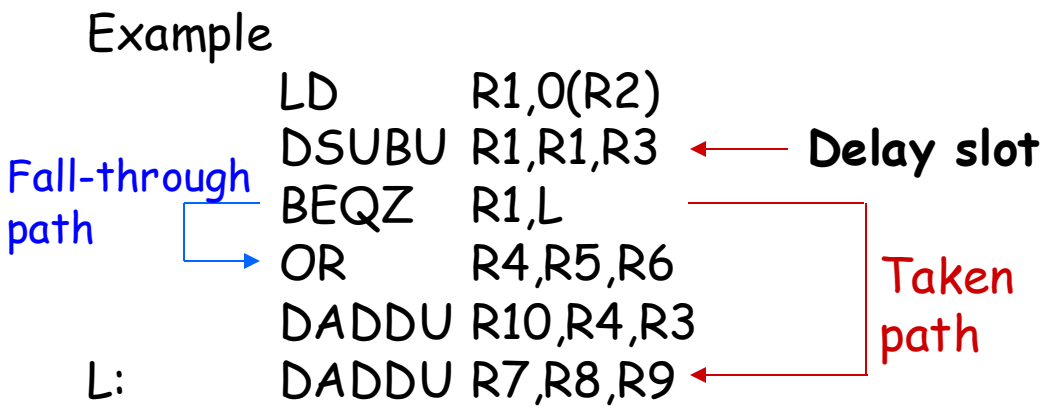
```
        LD      R1,0(R2)
        DSUBU   R1,R1,R3      <-- Delay slot
        BEQZ    R1,L
        OR      R4,R5,R6
        DADDU   R10,R4,R3
L:      DADDU   R7,R8,R9
```

Fall-through path

Taken path

```
LD      R1,0(R2)
DADDU   R7,R8,R9
DSUBU   R1,R1,R3
BEQZ    R1,L
OR      R4,R5,R6
DADDU   R10,R4,R3
L:
```

Delayed Branches and Loads (Cont'd)



Case2: The branch was rarely taken and the value of R4 was not needed on the taken path,

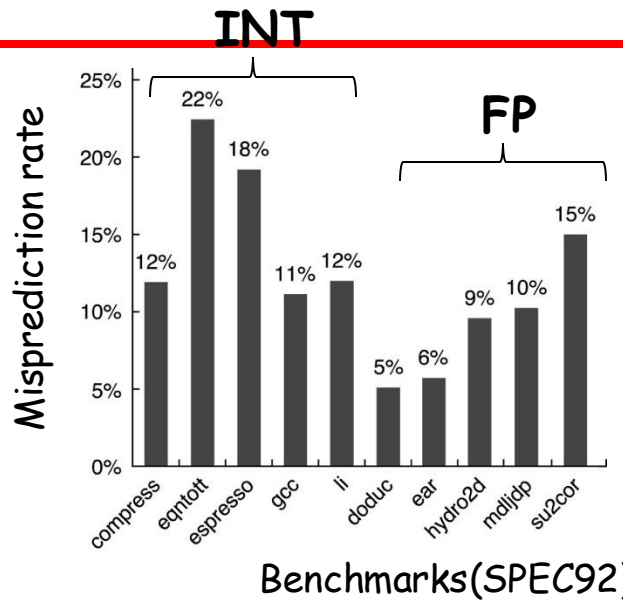
```
        LD    R1,0(R2)
        OR    R4,R5,R6
        DSUBU R1,R1,R3
        BEQZ  R1,L
        DADDU R10,R4,R3
L:      DADDU R7,R8,R9
```

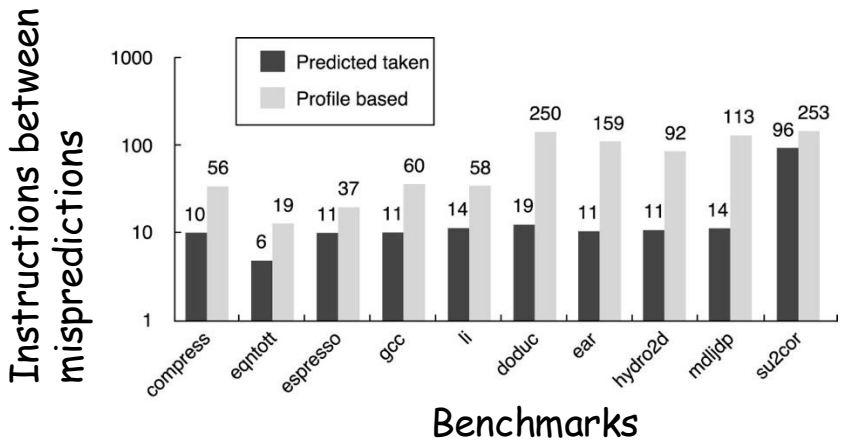
How Can We Predict the Branch Behavior Statically?

- Always predict a branch as taken
  - The misprediction rate ranges from not very accurate(59%) to highly accurate (9%), depending the individual SPEC programs (average 34%)
- Predict on the basis of branch direction
  - Choosing backward-going branches to be taken and forward-going branches to be not taken.
    - For some programs, the frequency of forward taken branches may be significantly less than 50%, but
    - Direction-based prediction is unlikely to generate an overall misprediction rate of less than 30% to 40%:more than half of the forward-going branches are taken in the SPEC benchmarks.
- Predict branches on the basis of profile information collected from earlier runs.
  - An individual branch is often highly biased toward taken or untaken.

Misprediction Rate on SPEC92 for a Profile-Based Predictor

- Better prediction rate for the FP programs than integer ones
  - FP Prgms: Average misprediction rate of 9% with a standard deviation of 4%
  - INT Prgms: Average misprediction rate of 15% with a standard deviation of 5%
- Profile-based predictor gives a longer span of instructions between mispredictions
  - 20 instructions in a predict-taken strategy and 110 instruction in the profile-based strategy.
    - Exploit higher ILP





Advanced Compiler Support for Exposing and Exploiting ILP

- Detecting and Enhancing Loop-Level Parallelism
  - Analyzed at the source level or close to it, instead of the instruction level
  - Involves determining what dependences exist among the operand in a loop across the iteration of that loop.
  - ✓ Loop-carried dependence
    - Data accesses in later iterations are dependent on data values produced in earlier iterations

Example: Consider a loop like this one:

```
For (i=1; i <= 100; i=i+1) {
        A[i+1] = A[i] + C[i];     /* S1 */
        B[i+1] = B[i] + A[i+1];   /* S2 */
}
```

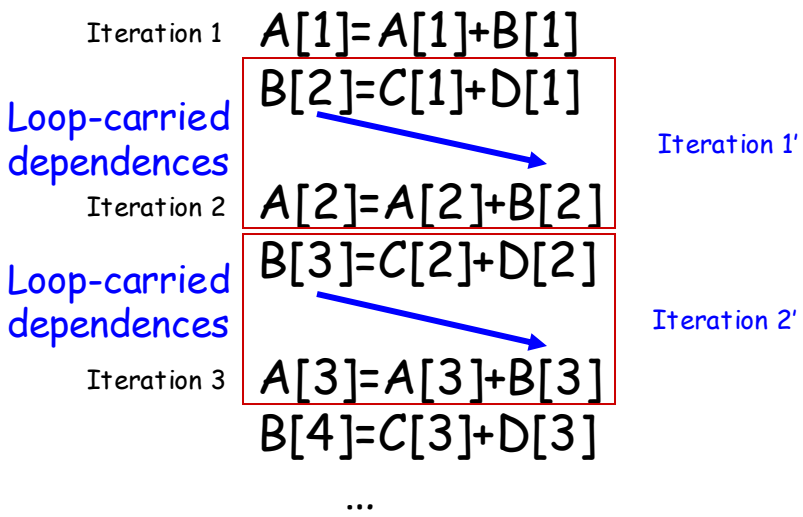
Loop-carried dependence

Intra-iteration dependence

What are the data dependences among the statements S1 and S2 in the loop?

Loop-Carried Dependence with Loop-Level Parallelism

```
For (i=1; i<=100; i=i+1){
    A[i]=A[i]+B[i];          /* S1 */
    B[i+1]=C[i]+D[i];        /* S2 */
}
```



- A loop-carried dependence between S1 and S2, but
- Neither statement depends on itself, and S2 does not depend on S1, although S1 depends on S2.

The absence of a cycle in the dependences means that dependences give a partial ordering on the statements and a loop is parallel.

Loop-Carried Dependence with Loop-Level Parallelism (Cont'd)

- There is no dependence from S1 to S2. If there were, then there would be a cycle in the dependences and the loop would not be parallel. Since this other dependence is absent, interchanging the two statements will not affect the execution of S2
- On the first iteration of the loop, statement S1 depends on the value of B[1] computed prior to initiating the loop

```
A[1] = A[1] + B[1]
for (i=1; i<=99; i=i+1) {
        B[i+1] = C[i] + D[i];
        A[i+1] = A[i+1] + B[i+1];
}
B[101] = C[100] + D[100];
```

The dependence between the two statements is no longer loop carried, so that iterations of the loop may be overlapped, provided the statements in each iteration are kept in order.

Recurrence

- Loop-carried dependences in the form of a recurrence.

```
for (i=2; i<=100; i=i+1) {
    Y[i] = Y[i-1] + Y[i];
}
```

- Dependence distance *k*
  - On the iteration *i*, the loop references element *i-k*
    - A recurrence with a dependence distance of 5
  - For (i=6; i<=100:i=i+1) {
    Y[i] = Y[i-5] + Y[i];
    }

The larger distance, the more potential parallelism can be obtained by unrolling the loop.

- If we unroll the loop with a dependence distance of 5, there is a sequence of five statements that have no dependences.

Finding Dependences

- Finding the dependences in a program is an important part of three tasks:
  1. Good scheduling of code,
  2. Determining which loops might contain parallelism, and
  3. Eliminating name dependences.

- How does the compiler detect dependences in general?
  - Assume that array indices are *affine,* where an indices can be written in the form $a \times i + b$
  - Determining whether there is a dependence between two references to the same array in a loop is thus equivalent to determining whether two affine functions can have the same value for different indices between the bounds of the loop.

Finding Dependences (Cont'd)

- A dependence exists if two conditions hold:

1. There are two iteration indices, j and k, both within the limits of the for loop.
   - $m \leq j \leq n$, $m \leq k \leq n$
2. The loop stores into an array element indexed by $a \times j + b$ and later fetches from that same array element when it is indexed by $c \times k + d$.
   - $a \times j + b = c \times k + d$

- As many programs contain primarily simple indices where a, b, c, and d are all constants, it is possible to devise reasonable compile time tests for dependence.

➢ GCD test for finding dependences
   - If a loop-carried dependence exists, then GCD(a,c) must divide (d-b)
     - An integer, x *divides* another integer, y, if we get an integer quotient when we do the division y/x and there is no remainder.

Finding Dependences: Example

Use the GCD test to determine whether dependences exist in the following loop:

```
for (i=1; i<=100; i=i+1) {
        X[2*i+3] = X[2*i]*5.0;
        }
```

- Given the values a=2, b=3, c=2 and d=0, then GCD(a,c)=2, and d-b=-3.
  - Since 2 does not divide -3, no dependence is possible.

- The CGD test is sufficient to guarantee that no dependence exists, however there are cases whether the test succeeds but no dependence exists.

In general, determining whether a dependence actually exists is NP-complete, but many common cases can be analyzed precisely at low cost!

Dependence Analysis is Critical, but There Are Some Cases Not Applicable…

- Dependence analysis is a critical technology for exploiting parallelism, and used effectively to compile programs to either vector computers or multiprocessors.
- But it applies only under the limited circumstances, namely, among references within a single loop nest and using affine index function.

- We cannot detect array-oriented dependences in the following cases:
  - When objects are referenced via pointers rather than array indices
  - When array indexing is indirect through another array, which happens with many representations of sparse arrays
  - When a dependence may exist for some value of the inputs, but does not exist in actuality when the code is run since the inputs never take on those values

Eliminating Dependent Computations by Compiler

- Copy propagation

```
DADDUI   R1,R2,#4
DADDUI   R1,R1,#4
```

```
DADDUI   R1,R2,#8
```

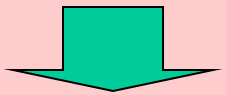
- Tree height reduction
  - Make the height of the tree structure representing a computation *wider but shorter*.

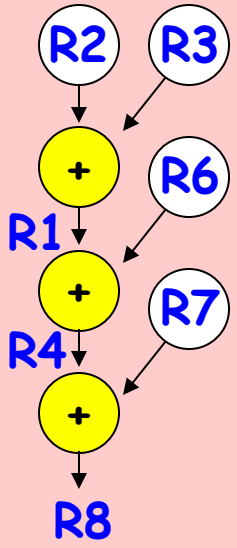
Tree height reduction

Sequential code

```
ADD   R1,R2,R3
ADD   R4,R1,R6
ADD   R8,R4,R7
```

Three execution cycles

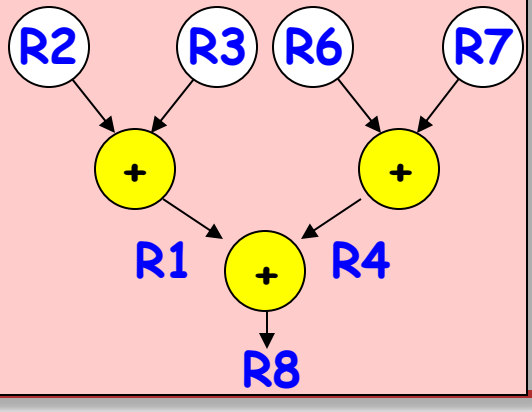




Parallelizable code

```
ADD   R1,R2,R3
ADD   R4,R6,R7
ADD   R8,R1,R4
```

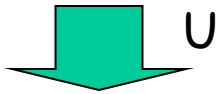
Two execution cycles!



Optimization Technique for a Loop with a Recurrence

Example:

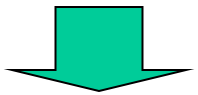
```
for (i=1; i<=100; i++) sum = sum +x[i];
```



Unrolling five times

```
sum = sum +x1+x2+x3+x4+x5;
```

five operations needed.

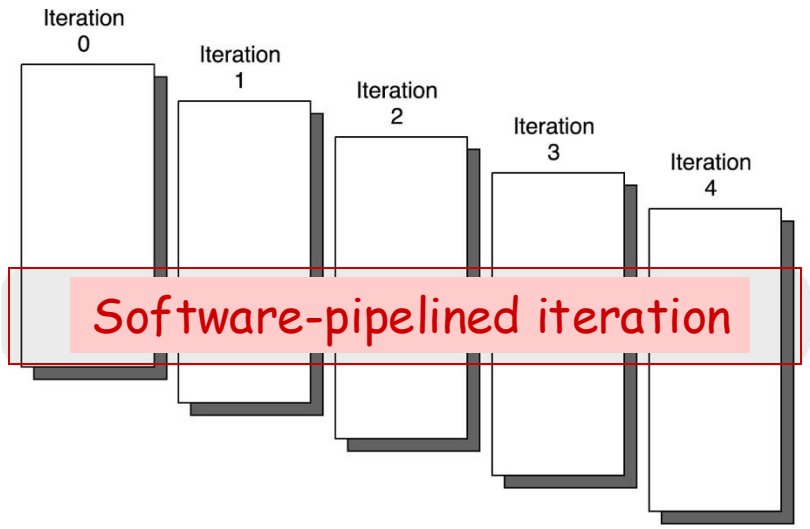


```
sum =( (sum +x1)+(x2+x3))+(x4+x5);
```

Only three dependent operations needed.

Software Pipelining: Symbolic Loop Unrolling

- A technique for reorganizing loops such that each iteration in the software-pipelined code is made from instructions chosen from different iterations of the original loop.
  - The scheduler essentially interleaves instructions from different loop iterations, so as to separate the dependent instructions that occur within a single loop iteration.

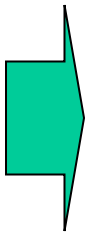


Software pipeline is the software-counterpart to what Tomasulo's algorithm does in hardware.

Software-Pipelining Example

- Example

```
Loop: LD      F0,0(R1)
      ADD     F4,F0,F2
      ST      F4,0(R1)
      DADDUI  R1,R1,#-8
      BNE     R1,R2,Loop
```



Symbolically unrolling in a steady state

```
Iteration i:      LD      F0,0(R1)
                  ADD     F4,F0,F2
                  ST      F4,0(R1)
Iteration i+1:    LD      F0,0(R1)
                  ADD     F4,F0,F2
                  ST      F4,0(R1)
Iteration i+2:    LD      F0,0(R1)
                  ADD     F4,F0,F2
                  ST      F4,0(R1)
```

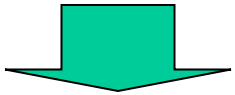
```
Loop  ST      F4,16(R1)   ; stores into M[i]
      ADD     F4,F0,F2    ;adds to M[i-1]
      LD      F0,0(R1)    ;loads M[i-2]
      DADDUI  R1,R1,#-8
      BNE     R1,R2,Loop
```

Software-Pipelining Example (Cont'd)

Final software-pipelined code in a steady state

```
Loop  ST      F4,16(R1)   ; stores into M[i]
      DADDUI  R1,R1,#-8
      ADD     F4,F0,F2    ;adds to M[i-1]
      BNE     R1,R2,Loop
      LD      F0,0(R1)    ;loads M[i-2]
```

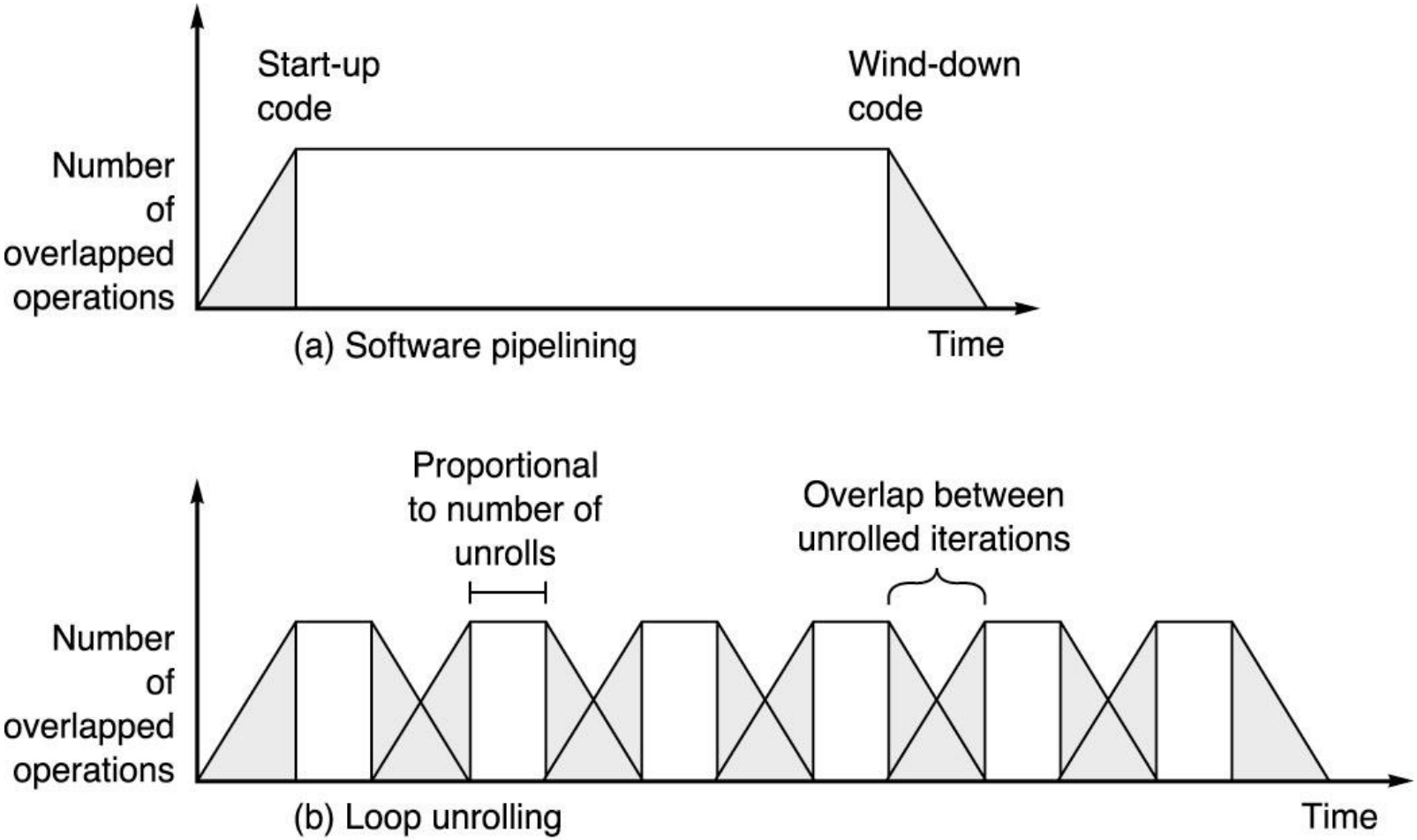
Five cycles per results without any stall in a steady state!



- Before the steady state code;
  LD for iterations 1 and 2, and Add for iteration 1 are needed.
  (*Start-up code,* or also called *prologue*)

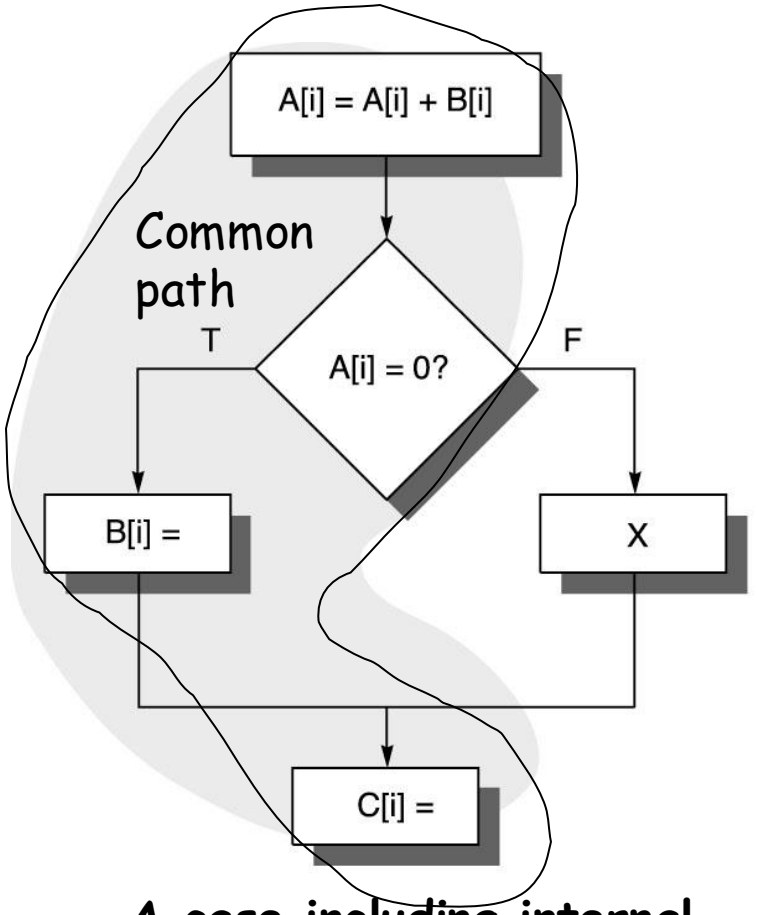
- After the steady state code;
  ADD for the last iteration and SD for last two iterations are needed.
  (*Finish-up code,* or also called *epilogue*)

Execution Pattern for a Software-Pipelined Loop



Global Code Scheduling

- Code scheduling beyond the boundaries of basic blocks to increase the ILP
  - Of course, preserves the data and control dependences
  - Global code motion requires estimates of the relative frequency of different paths, since moving code across branches will often affect the frequency of execution of such code.

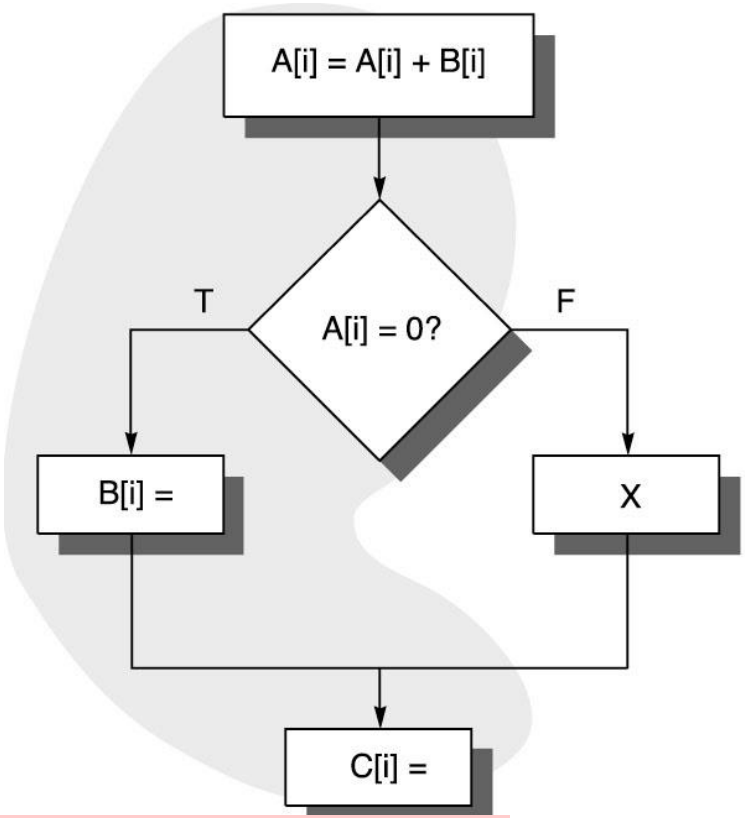


A case including internal control flow in each iteration

How can we make global scheduling for this code fragment ?

Global Code Scheduling Example

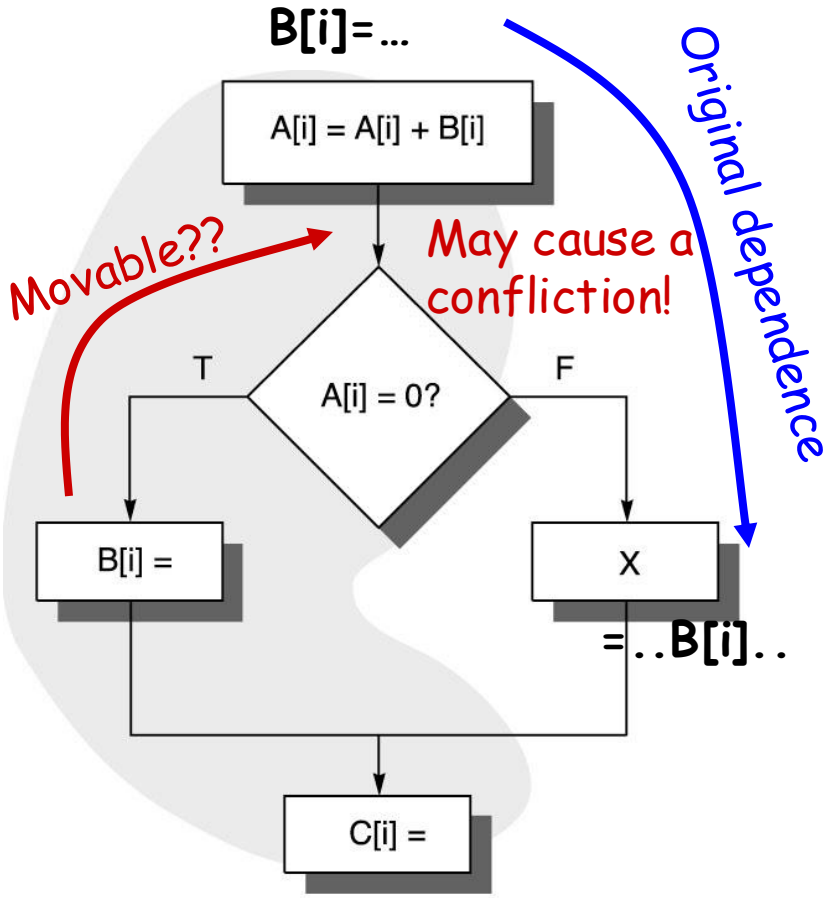
```
            LD      R4,0(R1)      ;load A
            LD      R5,0(R2)      ;load B
            DADDU   R4,R4,R5      ;Add to A
            ST      R4,0(R1)      ;Store A
            ...
            BNEZ    R4,elsepart
            ...                   ;then part
            ST      ...,0(R2)     ;Store to B
Movable??   ...
            J       join
elsepart:   ...                   ;else part
            X                     ;code for X
            ...
join:       ...                   ;after if
            ST      ...,0(R3)     ;store C[i]
```



Can we move the assignments to B and C to earlier in the execution sequence, before the test of A???

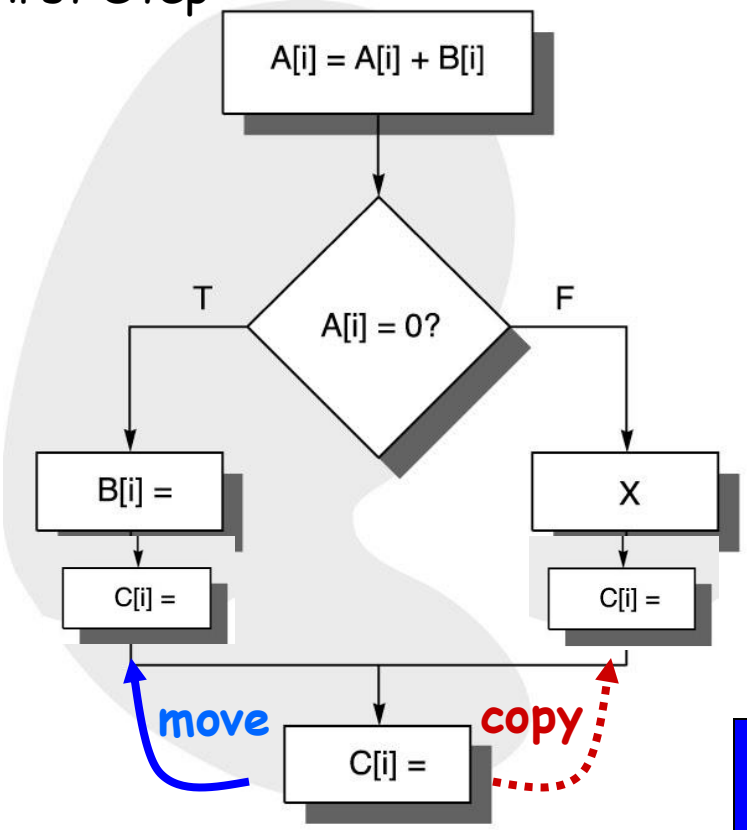
Consideration in Global Code Scheduling

- Global code motion must satisfy a set of constraints to be legal.
- Speculative code motion should be applied to the cases in which the path containing the code would be taken
  - otherwise it *will not* speed the computation up!

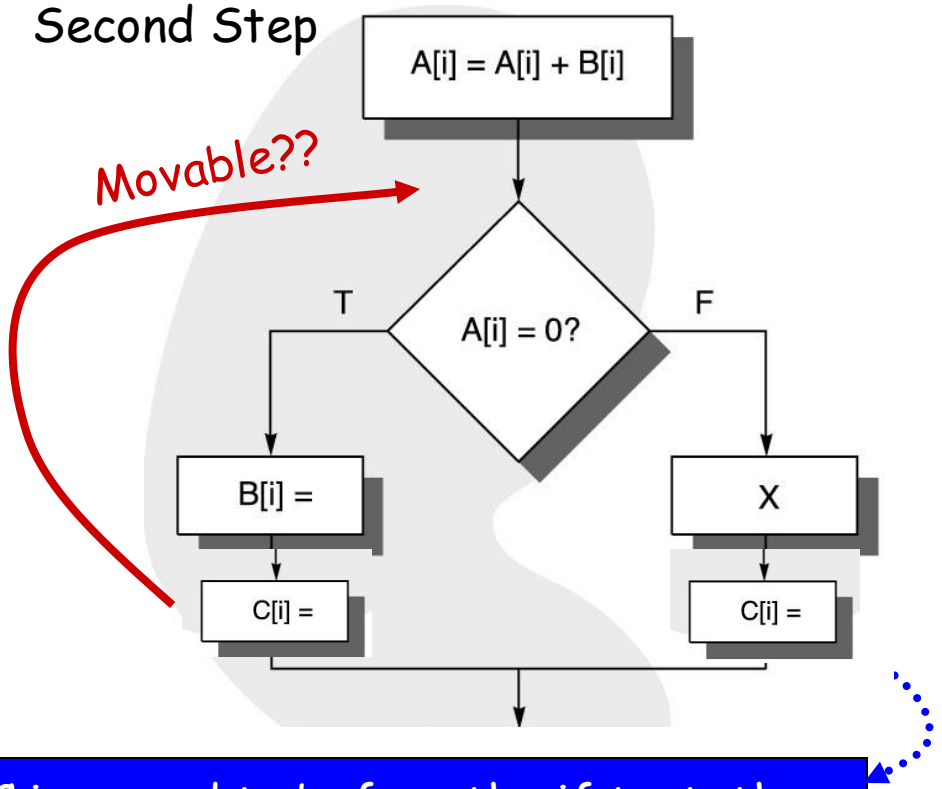


Consideration in Global Code Scheduling (Cont'd)

First Step



Second Step



If C is moved to before the if test, the copy of C in the else branch can usually be eliminated, since it will be redundant.

Factors that the compiler would have to consider in moving the computation and assignment of B

- What are the relative execution frequencies of the then case and the else case in the branch?
  - If the then case is much more frequent, the code motion may be beneficial. If not, it is less likely, although not impossible, to consider moving the code.
- What is the cost of executing the computation and assignment to B above the branch?
  - It may be that there are a number of empty instruction issue slots in the code above the branch and that the instructions for B can be placed into these slots that would otherwise go empty. This opportunity makes the computation of B "free" at least first order
- How will the movement of B change the execution time for the then case?
  - If B is at the start of the critical path for the then case, moving it may be highly beneficial.
- Is B the best code fragment that can be moved above the branch? How does it compare with moving C or other statements within the then case?
- What is the cost of the compensation code that may be necessary for the else case? How effectively can this code be scheduled, and what is its impact on execution time?

Trace Scheduling: Focusing on the Critical Path

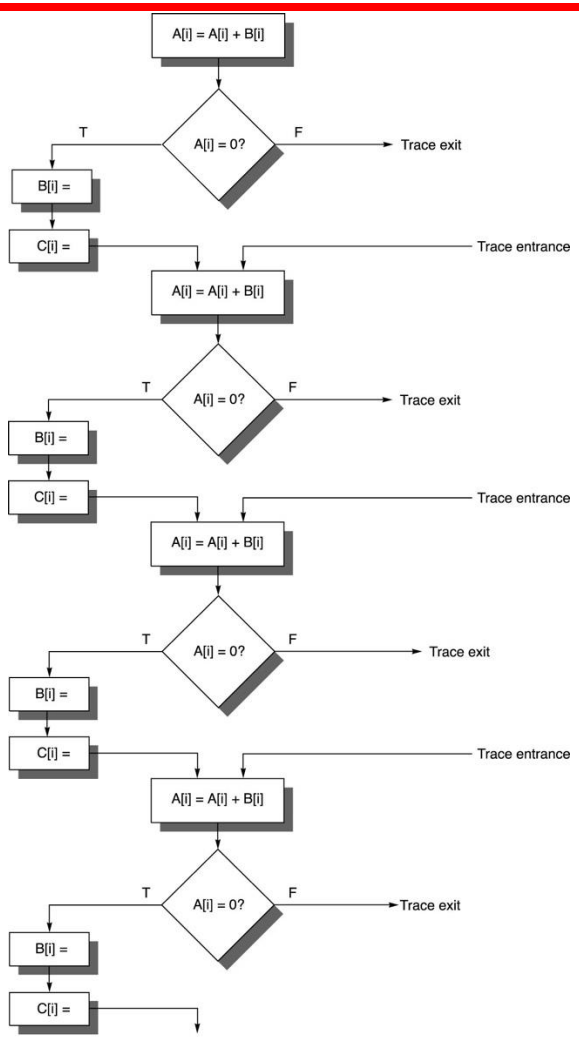
- Trace scheduling is a way to organize the global code motion process, so as to simplify the code scheduling by incurring the costs of possible code motion on the less frequent paths.
  - Because it can generate *significant* overheads on the designated infrequent path, it is best used where profile information indicates significant differences in frequency between different paths and where the profile information is highly indicative of program behavior independent of the inputs.

First Step: trace selection

- Tries to find a likely sequence of basic blocks whose operations will be put together into a smaller number of instructions (traces).
  - Loop unrolling used to generate long traces since loop branches are taken with high probability
  - By using static branch prediction, other conditional branches are also chosen as taken or not taken, so that the resultant trace is a straight-line sequence resulting from concatenating many basic blocks

Trace scheduling has been developed for VLIW processors

Trace Scheduling: Focusing on the Critical Path



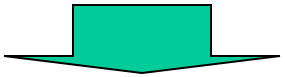
Second Step: trace compaction

- Tries to squeeze the trace into a small number of wide instructions.
  - Trace compaction is code scheduling;
  - It attempts to move operations as early as it can in a sequence (trace), packing the operations into as few wide instructions (or issue packets) as possible.
  - When code is moved across such trace entry and exit points, additional bookkeeping code will often be needed on the entry or exit point.

This trace is generated by unwinding four times.

Superblock Scheduling: Motivation

- One of the major drawbacks of trace scheduling is that the entries and exits into the middle of the trace cause significant complications, requiring the compiler to generate and track the compensation code and often making it difficult to assess the cost of such code.

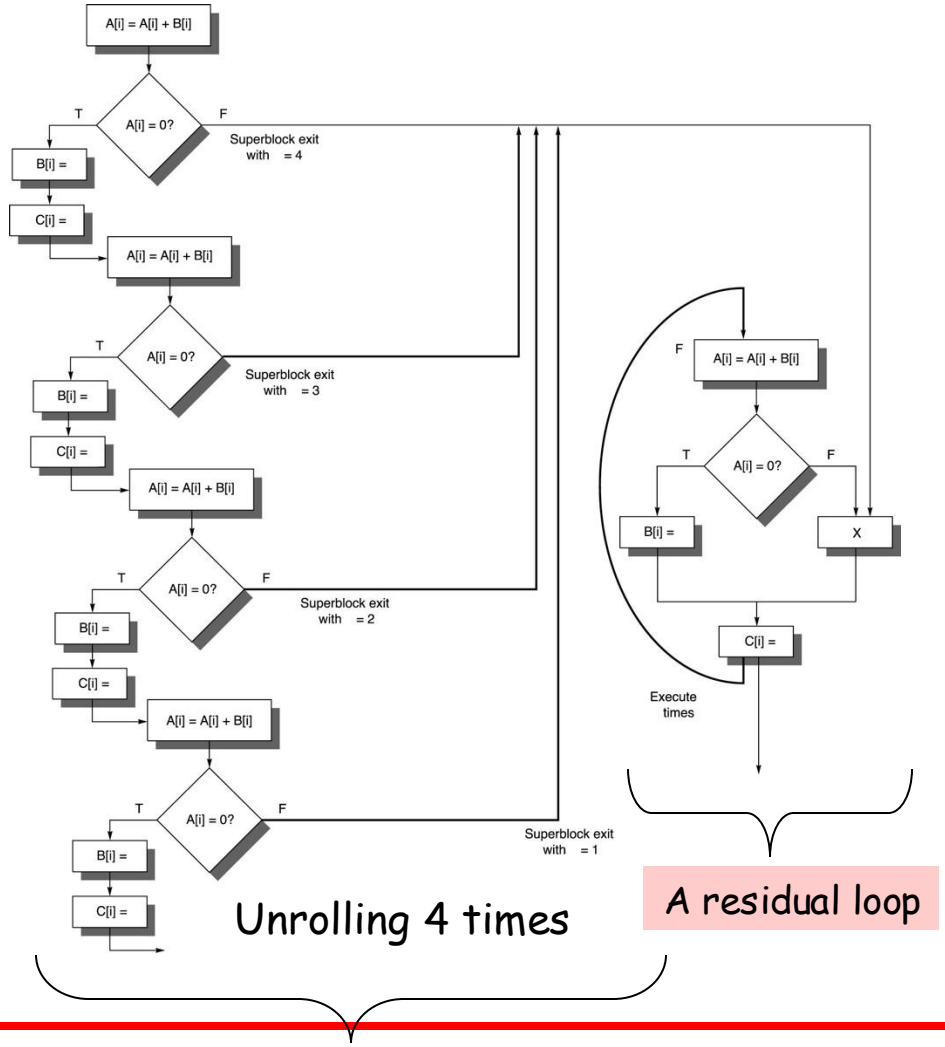


Superblocks are formed by a process similar to that used for traces, but are a form of extended basic blocks, which are restricted to *a single entry point but allow multiple exits*.

Superblock Scheduling

- How can a superblock with only one entrance be constructed?
- ✓ Use *tail duplication* to create a separate block that corresponds to the portion of the trace after the entry.

The superblock approach reduces the complexity of bookkeeping and scheduling versus the more general trace generation approach



Conditional or Predicated Instructions

- *Conditional or predicated instructions* can be used to eliminate branches, converting a control dependence into a data dependence and potentially improving performance.

- Concept: An instruction refers to a condition, which is evaluated as apart of the instruction execution

Example:
consider the following code:

```
if (A==0) {S=T;}
```

Straightforward code:

```
        BNEZ   R1,L
        ADDU   R2,R3,R0
L:
```

Code using a conditional move:

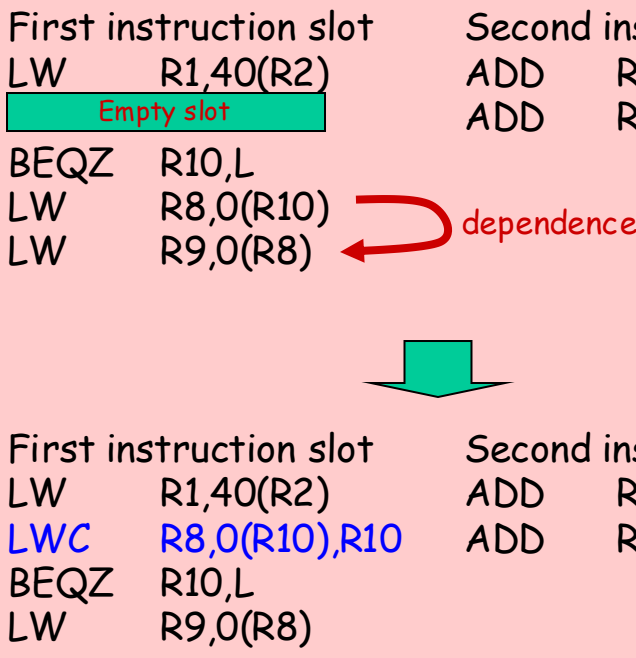
```
CMOVZ R2,R3,R1
```

For a pipelined processor, this moves the place where the dependence must be solved from near the front of the pipeline, where it is resolved for branches, to the end of the pipeline, where the register write occurs.

Application of Predication to All Instructions

- Support full predication for all instruction
  - When the predicate is false, the instruction becomes a no-op.
  - Full predication allows us to simply convert large blocks of code that are branch dependent
  - An if-then-else statement within a loop can be entirely converted to predicated execution, so that the code in the then case executes only if the value of the condition is true, and the code in the else case executes only if the value of the condition is false.
- Predication is particularly valuable with global code scheduling, since it can eliminate nonloop branches, which significantly complicate instruction scheduling: *Intel/HP IA-64*

Example:



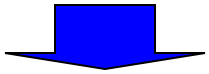
Limits on Instruction-Level Parallelism

Studies of the Limitation of ILP

Ideal Hardware Model to measure the potential ILP of programs

1. *Register renaming*: There are an infinite number of virtual registers available, and hence all WAW and WAR hazards are avoided and an unbounded number of instructions can begin execution simultaneously.
2. *Branch prediction*: Branch prediction is perfect. All conditional branches are predicted exactly.
3. *Jump prediction:* All jumps (including jump register used for return and computed jumps) are perfectly predicted. When combined with perfect branch prediction, this is equivalent to having a processor with perfect speculation and an unbounded buffer of instructions available for execution.
4. *Memory address alias analysis:* All memory addresses are known exactly and a load can be moved before a store provided that the addresses are not identical.

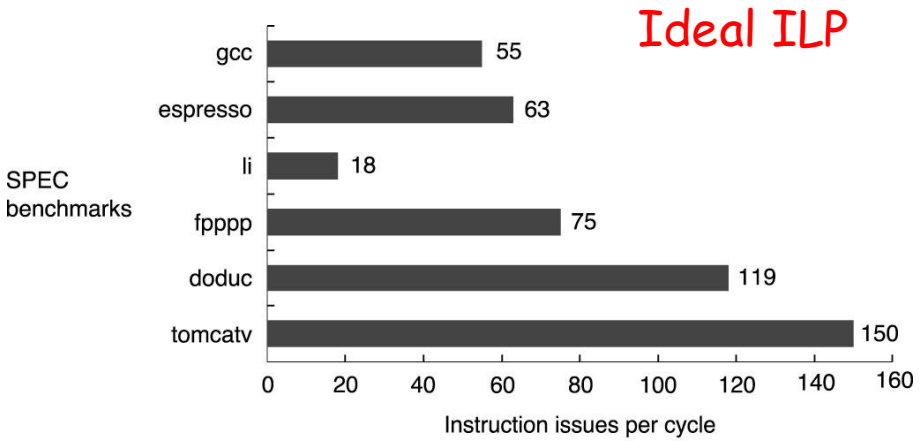
- Assumptions 2 and 3 eliminate *all* control dependences.
- Assumptions 1 and 4 eliminate *all but the true* data dependences.



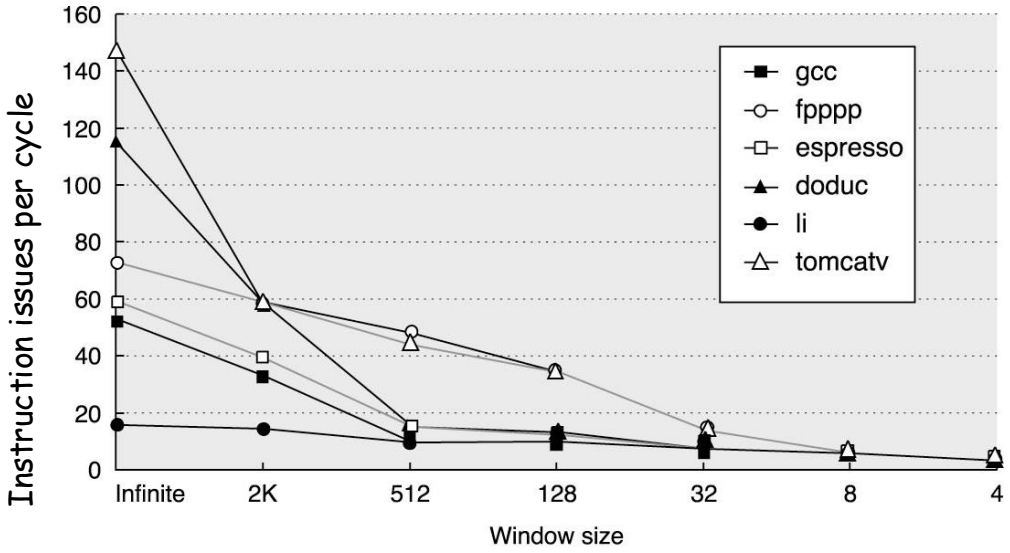
These four assumptions mean that *any* instruction in the program's execution can be scheduled on the cycle immediately following the execution of the predecessor on which it depends

Ideal Processor Model and Available ILP of SPEC92 Benchmarks

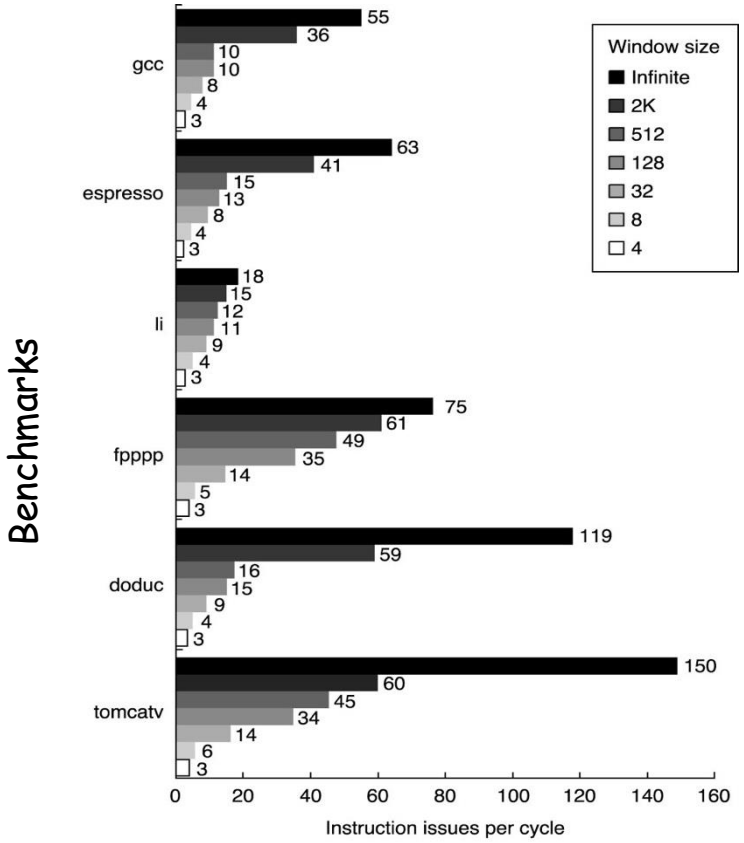
- Unlimited issue-bandwidth
- Unlimited Function Units
- Perfect Cache
  - All loads and stores always complete in one cycle
- One-cycle latency of all functional units



Factors Limiting Available ILP: Window Size

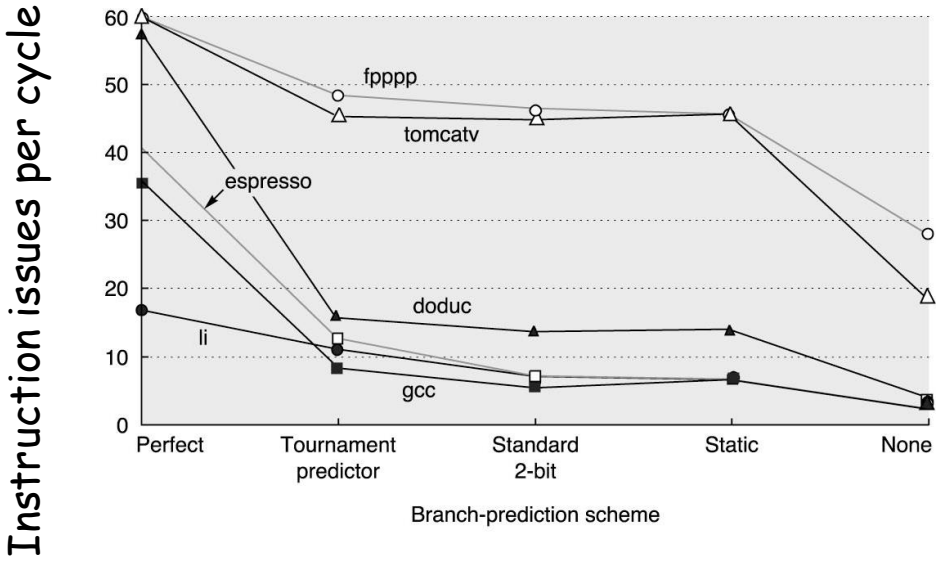


The effects of reducing the size of the window



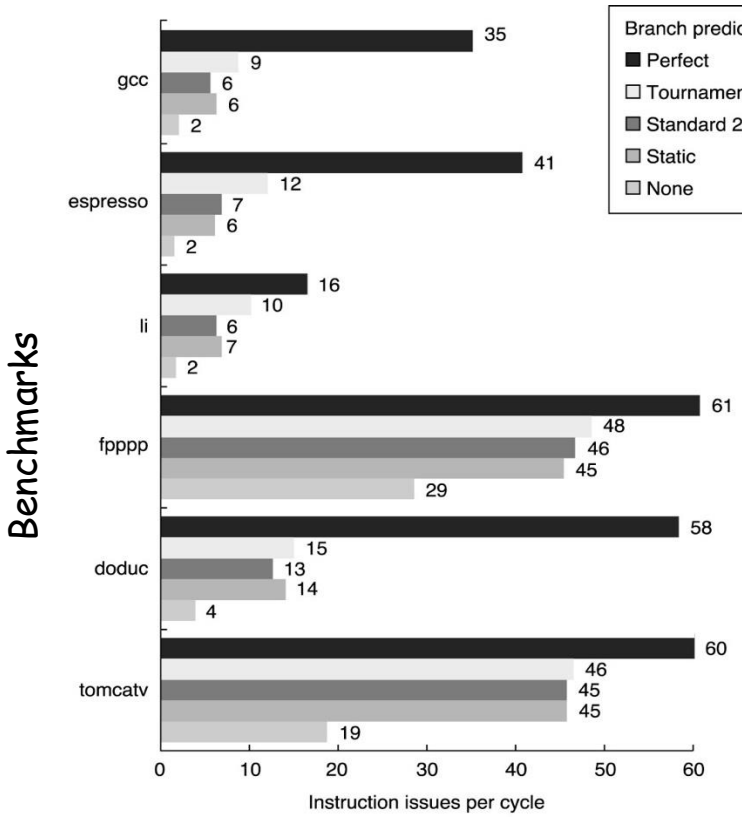
The effect of window size shown by each application

Factors Limiting Available ILP: Branch Prediction



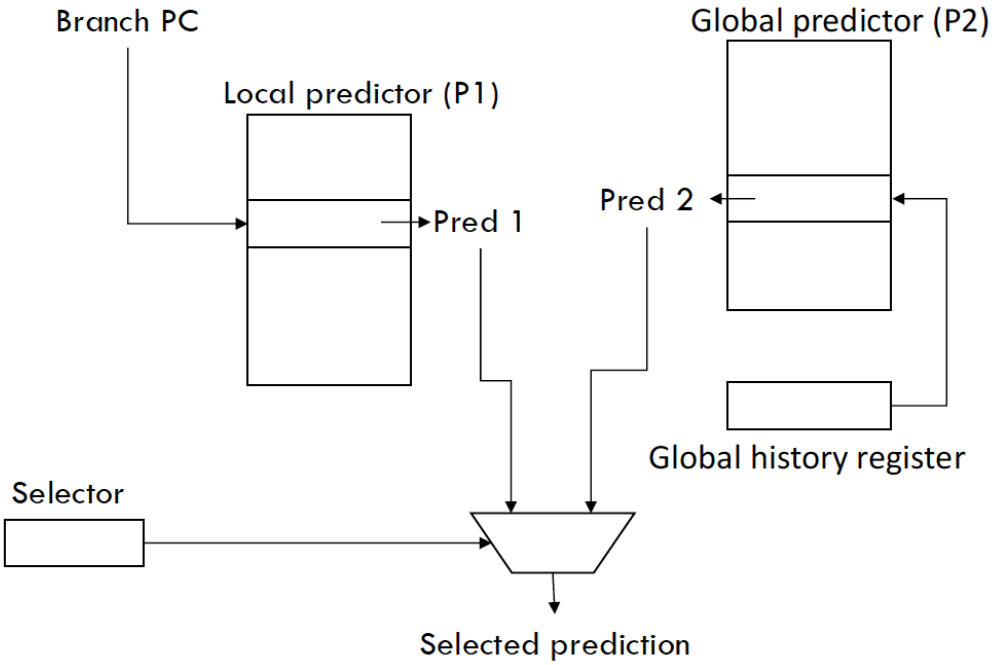
Effect of branch-prediction schemes

(2K windows size, max 64-way inst. issue/cycle assumed)



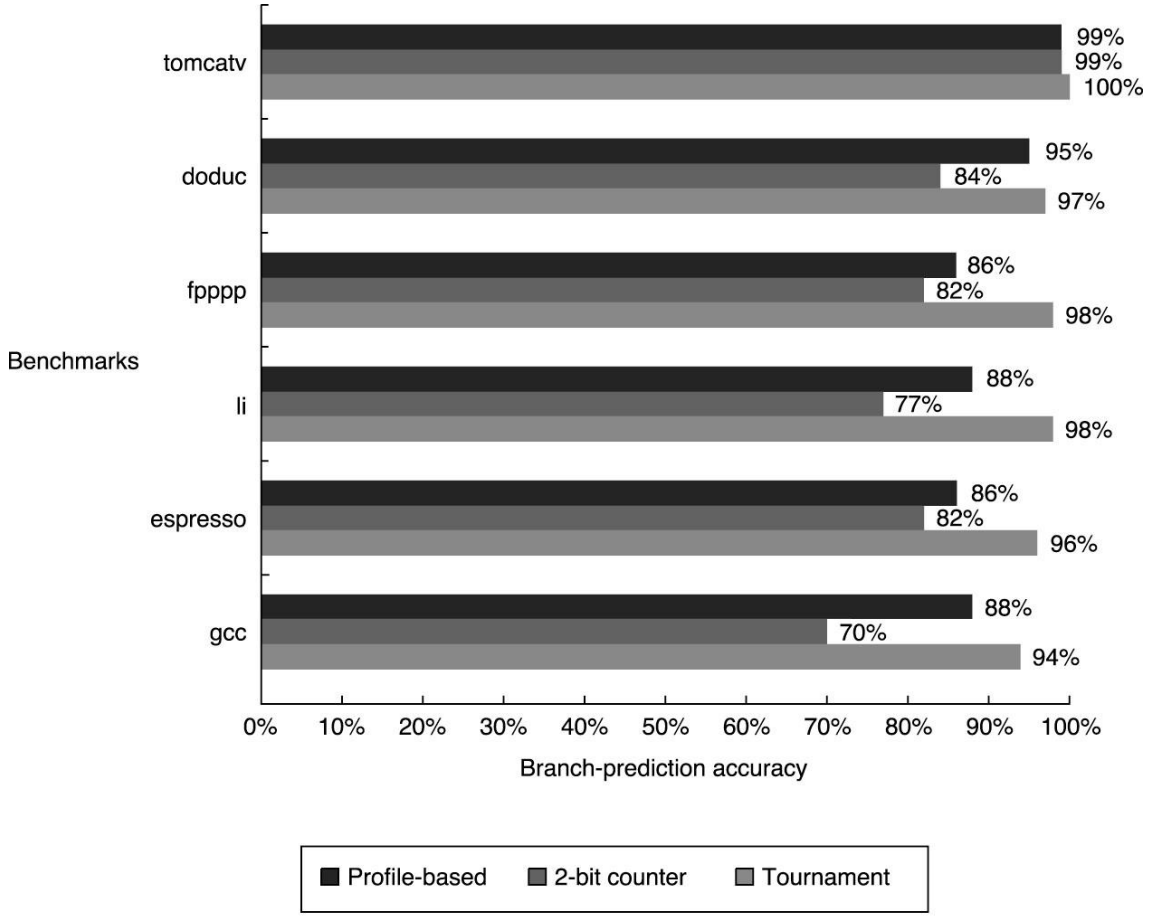
Effect of branch-prediction schemes sorted by application

Tournament Predictor Example



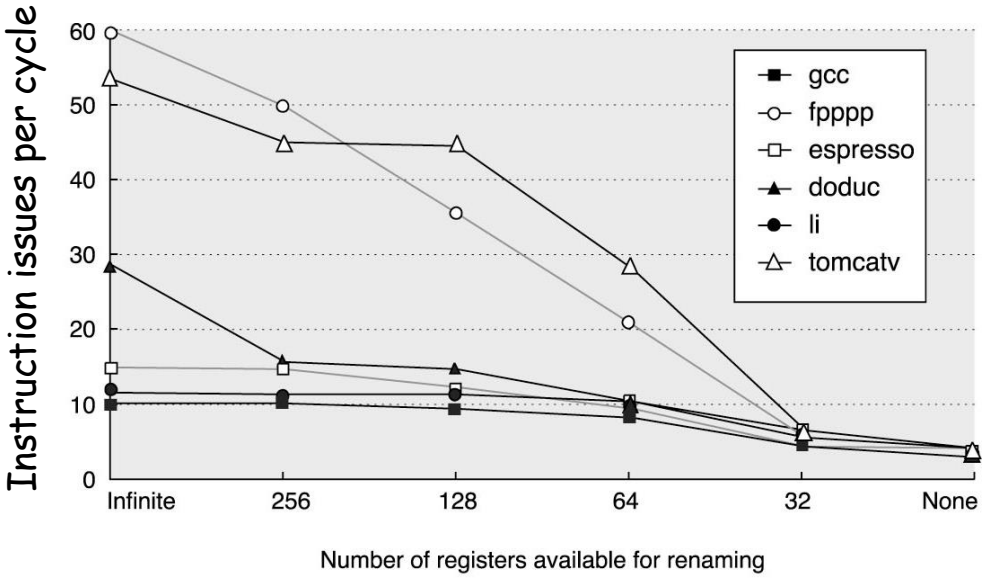
- Features:
  - Combine global and local predictors
  - Use a selector to select the prediction outcome from one of them
  - Update both predictors as well as the selector for each branch
- Selector Update
  - Update selector based on the prediction correctness of two predictors
- Predictor Indexing
  - Global predictor: indexed by history register
  - Local predictor: indexed by branch PC

Factors Limiting Available ILP: Branch Prediction



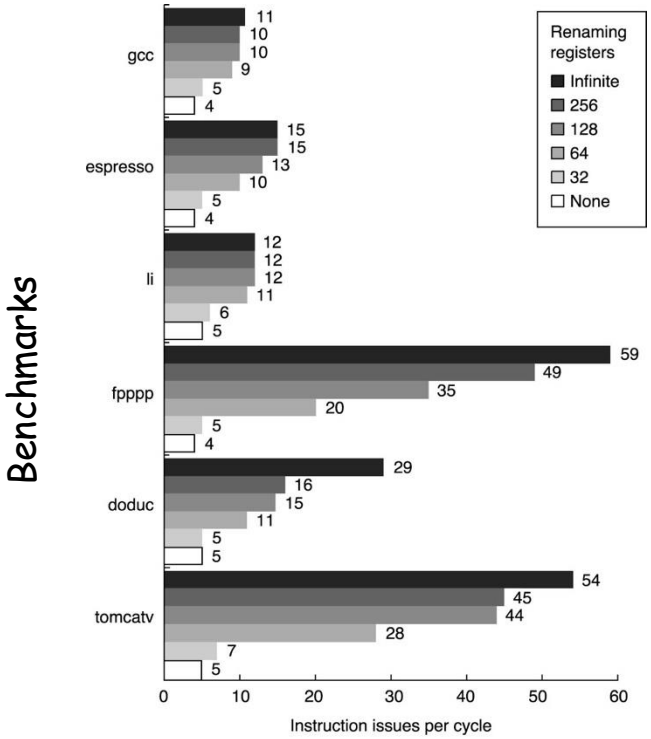
Branch-prediction accuracy for the conditional branches in the SPEC92 subset

Factors Limiting Available ILP: Finite Registers



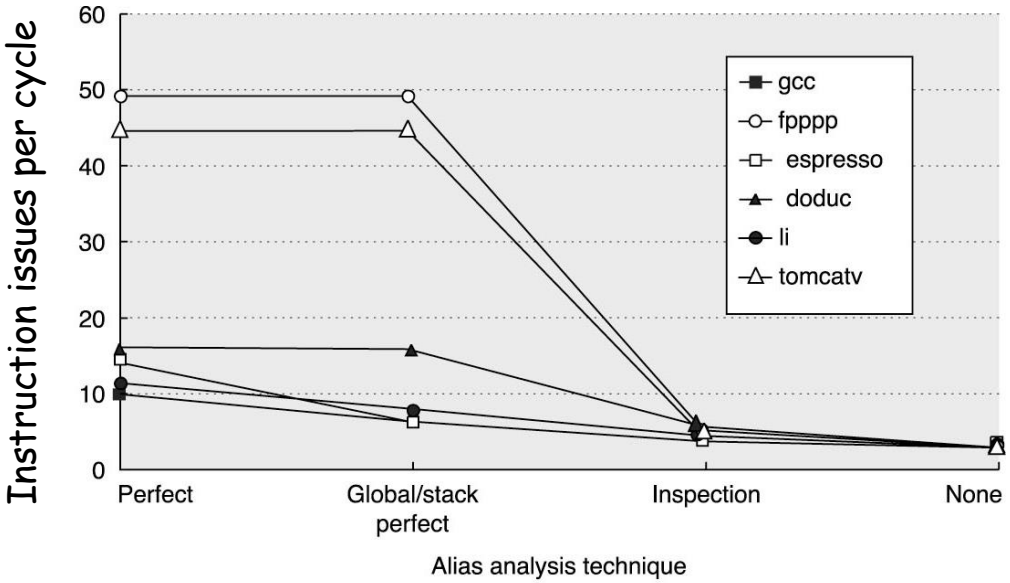
Effect of finite numbers of registers available for renaming

(2K windows size, max 64-way inst. issue/cycle, tournament predictor assumed)



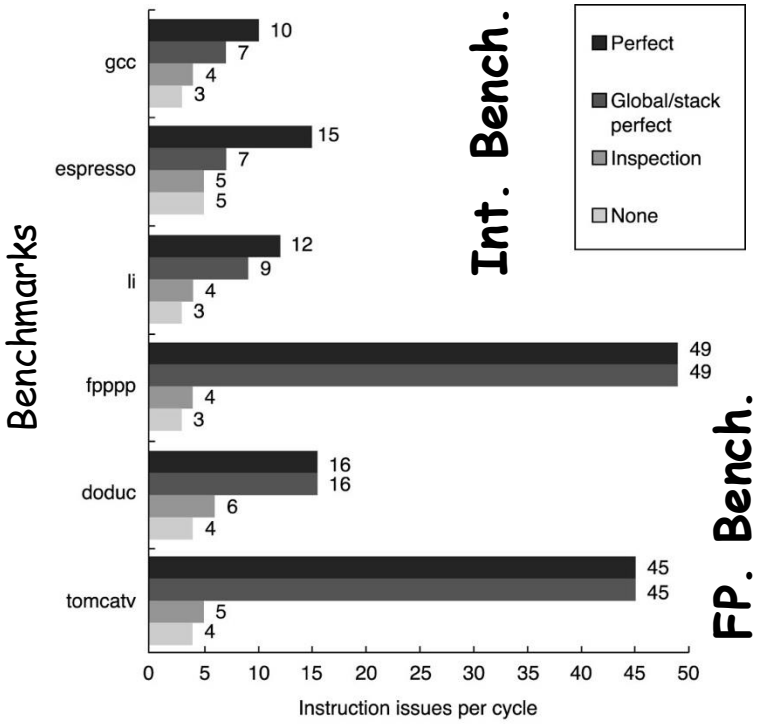
Reduction in available parallelism is significant when fewer than an unbounded number of renaming registers are available.

Factors Limiting Available ILP: Imperfect Alias Analysis



Effect of various alias analysis techniques on the amount of ILP

(2K windows size, max 64-way inst. issue/cycle, tournament predictor, +256FP+256INT reg. for renaming assumed)



Effect of varying levels of alias analysis on individual programs

Limitations on ILP for Realizable Processors

1. Up to 64 instruction issues per clock with no issue restrictions, or roughly 10 times the total issue width of the widest processor in 2011 (or later in core design).

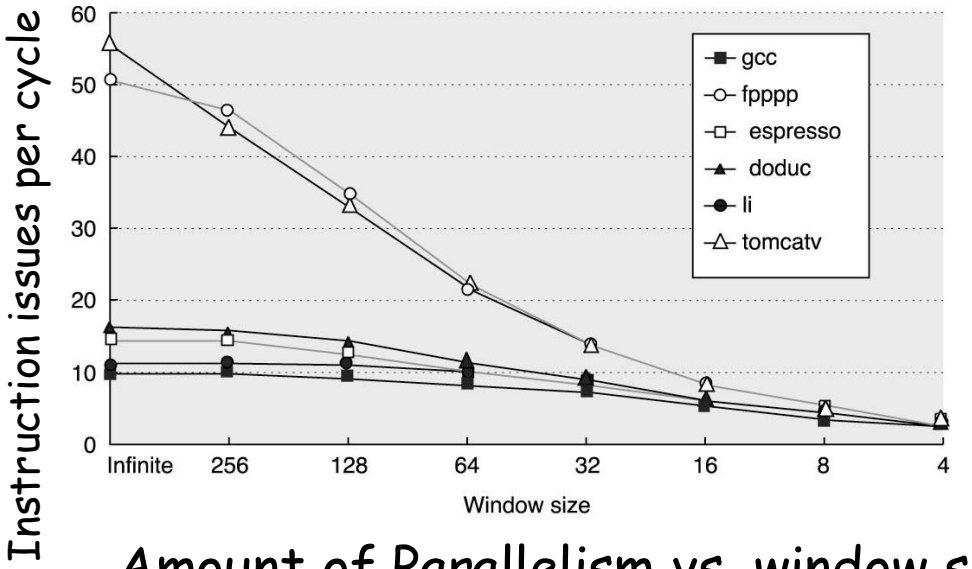
   The practical implications of very wide issue widths on clock rate, logic complexity, and power may be the most important limitation on exploiting ILP

2. A tournament predictor with 1K entries and a 16-entry return predictor. This predictor is fairly comparable to the best predictors in 2011; the predictor is not a primary bottleneck.

3. Perfect disambiguation of memory references done dynamically-this is ambitious but perhaps attainable for small window sizes (and hence small issue rates and load-store buffers) or through a memory dependence predictor.

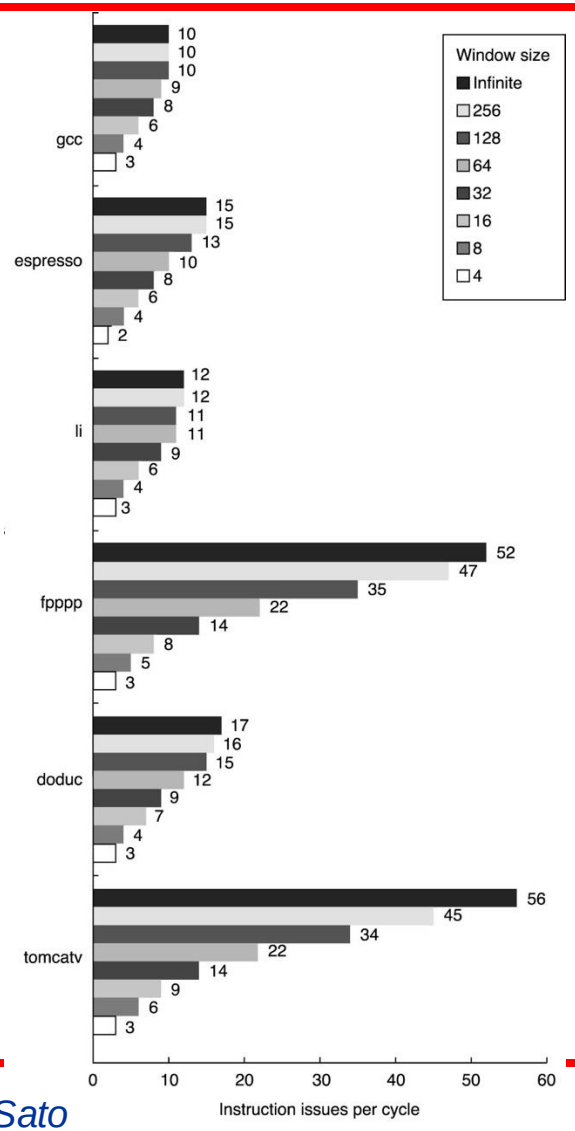
4. Register renaming with 64 additional integer and 64 additional FP registers, which is slightly less than the most aggressive processor in 2011.
   - Core i7 has128 entries and IBM Power7has 200 entries in the reorder buffer, no spit between integer and FP.

Limitations on ILP for Realizable Processors (cont'd)



Amount of Parallelism vs. window size

- **High loop-level parallelism in FP programs is restricted by window-size**
- **Branch prediction/register-renaming are limiting factors for limited ILP in Integer programs, rather than window size.**



How Should We Make a Right Decision on Design Tradeoff (1/4)

How best to use the limited resources available on an integrated circuit???

- Simpler processors with larger caches and higher clock rates

vs

- More emphasis on instruction-level parallelism with a slower clock and smaller caches

How Should We Make a Right Decision on Design Tradeoff(2/4)

Example: Consider the following three hypothetical, but not atypical, processors which we run with the SPEC gcc benchmark:

1. A simple MIPS two-issue static pipe running at a clock rate of 4GHz and achieving a pipeline CPI of 0.8. This processor has a cache system that yields 0.005 misses per instruction.
2. A deeply pipelined version of a two-issue MIPS processor with slightly smaller caches and a 5 GHz clock rate. The pipeline CPI of the processor is 1.0, and the small caches yield 0.0055 misses per instruction on average
3. A speculative superscalar with a 64-entry window. It achieves one-half of the ideal issue rate measured for this window size, i.e., 4.5. This processor has the smallest caches, which lead to 0.01 misses per instruction, but it hides 25% of the miss penalty on every miss by dynamic scheduling. This processor has a 2.5GHz clock.

Assume that the main memory time (which sets the miss penalty) is 50ns. Determine the relative performance of these three processors.

Which one has the best MIPS rate on GCC execution?

How Should We Make a Right Decision on Design Tradeoff (3/4)

Cache CPI = (Misses Per Instruction) × (Miss Penalty)

Miss Penalty = (Memory Access Time) / (Clock Cycle)

Clock cycle times for the processors are

Processor1@4GHz = 250ps

Processor2@5GHz = 200ps

Processor3@2.5GHz = 400ps

↓

Miss penalty1=50ns/250ps=200 cycles

Miss penalty2=50ns/200ps=250 cycles

Miss penalty3=75%x50ns/400ps=94 cycles

↓

Cache CPI1 = 0.005x200 = 1.0

Cache CPI2 = 0.0055x250 = 1.4

Cache CPI3 = 0.01x94 = 0.94

How Should We Make a Right Decision on Design Tradeoff (4/4)

Pipeline CPI of Processor3:

Pipeline CPI3 = 1/(Issue Rate) = 1/(9 x 0.5) = 1/4.5 = 0.22

CPI for each processor

CPI1 = 0.8+1.0=1.8

CPI2 = 1.0+1.4=2.4

CPI3 = 0.22+0.94=1.16

Instruction Execution Rate in MIPS = (Clock Rate)/CPI

IER1 = 4000MHz/1.8 = 2222MIPS

IER2 = 5000MHz/2.4 = 2083MIPS

IER3 = 2500MHz/1.16 = 2155MIPS

In this example, the simple two-issue static superscalar looks best!
performance depends on both CPI and Clock rate

Hardware versus Software Speculation (1/2)

- Dynamic run time disambiguation of memory address
  - Support for speculative memory references can help overcome the conservation of the compiler
- Hardware-based speculation works better when control flow is unpredictable and when hardware-based branch prediction is superior to software-based branch prediction done at compile time.
  - For many Integer programs
- Hardware-based speculation maintains a completely precise exception model even for speculated instructions
- Hardware-based speculation does not require compensation or bookkeeping code
- Hardware-based speculation with dynamic scheduling does not require different code sequences to achieve good performance for different implementations of an architecture.
- Compiler-based approaches may benefit from the ability to see further in the code sequence, resulting in better code scheduling than a purely hardware-driven approach

Hardware versus Software Speculation (2/2)

- The major disadvantage of supporting speculation in hardware is the complexity and additional hardware resources required.
- ✓ and should be evaluated against
  - complexity of a compiler for a software-based approach and the amount and usefulness of the simplifications in a processor that relies on such a compiler

Combination of the dynamic and compiler-based approaches to achieve the best of each???

Thread Level Parallelism

- ILP can be quite limited or hard to exploit, but there may be significant parallelism occurring naturally at a higher level in the application.
- ➢ Exploit Thread-Level Parallelism!
  - ✓ Parallel queries and updates in an online transaction-processing system, 3D simulations in scientific applications
- What are a thread and thread-level parallelism?
  - ✧ A *thread* is a separate process with its own instructions and data.
  - ✧ Each thread has all the state (instructions, data, PC, register state etc.) necessary to allow it to execute.
  - ✧ *Thread-level parallelism* is explicitly represented by the use of multiple threads of execution that are inherently parallel.

Multithreading: Exploitation of Thread-Level Parallelism for Instruction-Level Parallel Processing

- *Multithreading* allows multiple threads to share the functional units of a single processor in an overlapping fashion.
  - ✧ Try to keep the function units busy during stalls due to dependency in the code and/or memory accesses
  - ✓ The processor must duplicate the independent state of each thread,
    - e.g., separate PC, page table
    - Memory can be shared through the virtual memory
  - ✓ The ability to change to a different thread relatively quickly should be supported
    - more efficient than a process switch, which takes 100~1000 cycles

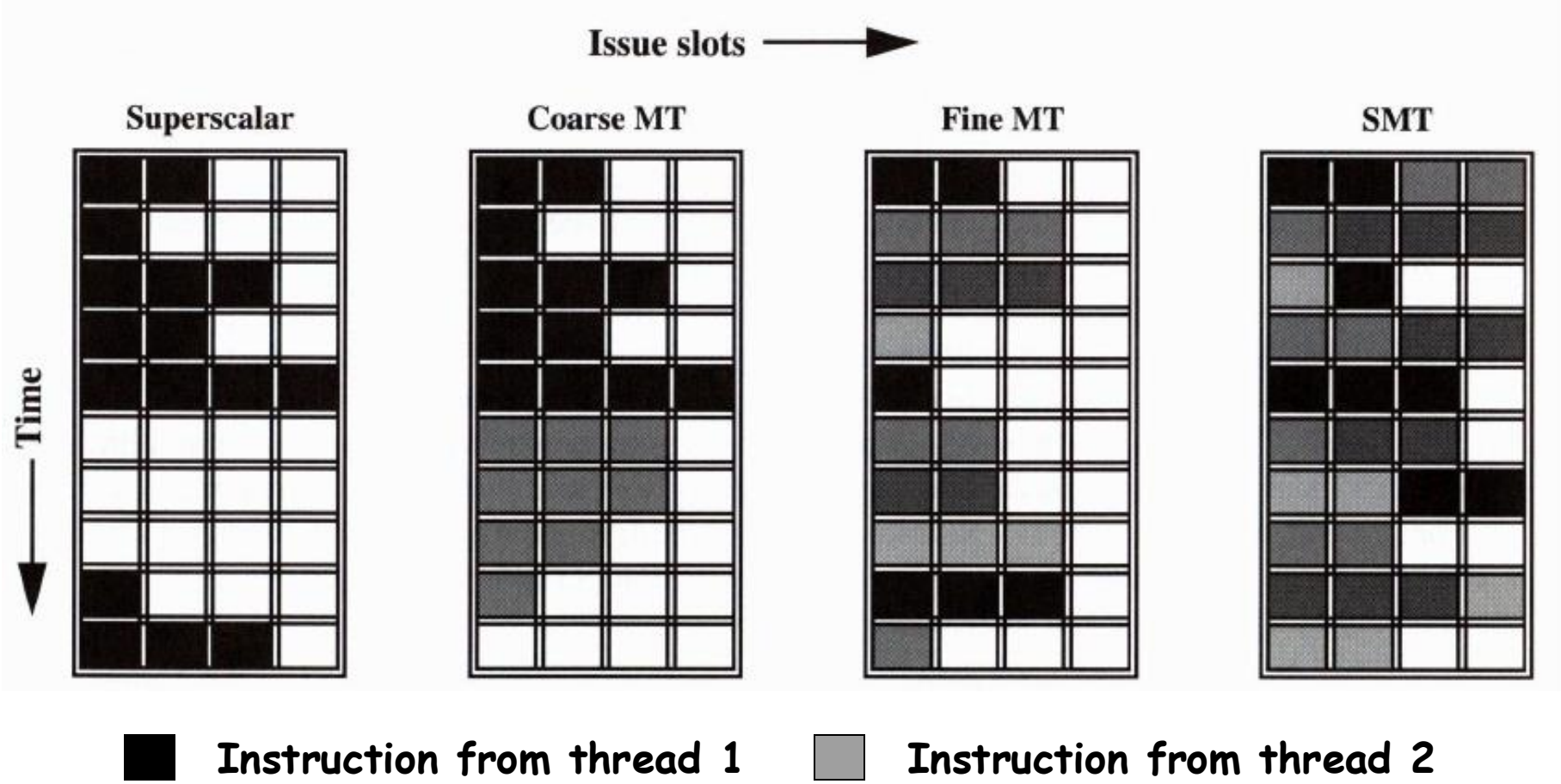
Approaches to Multithreading

- Fine-grain multithreading
  - Switches between threads on each instruction, causing the execution of multiple threads to be interleaved.
  - ☺ Can hide throughput loss that arise from both short and long stalls
  - ☹ Slows down the execution of the individual threads
  - Sun Niagara processor (T1~T5), GPU
- Coarse-grained multithreading
  - Switches threads only on costly stalls, such as L2/L3 $ misses.
  - ☺ Relieves the need to have thread-switching be essentially free and is much less likely to slow the processor down
  - ☹ Limits in its ability to overcome throughput losses, especially from shorter stalls, because of a large overhead in thread-switching
    - ☹ Pipeline refill << stall time

Approaches to Multithreading

- Simultaneous multithreading issues multiple instructions from multiple threads using the issue slots in a single clock cycle.
  - TLP and ILP are exploited simultaneously
  - Ideally, the issue slot usage is limited by imbalances in the resource needs and resource availability over multiple threads
  - In practice, other factors can also restrict how many slots are used.
    - How many active threads are considered
    - Finite limitation on buffers
    - The ability to fetch enough instructions from multiple threads
    - Practical limitations of what instruction combinations can issue from one thread and from multiple threads
  - Intel Core i7, IBM Power7

Four Different Approaches to Use the Issue Slots of a Superscalar Processor



Hardware Mechanism to Support SMT

- A large set of virtual registers that can be used to hold the register sets of independent threads
  - Separate renaming tables are kept for each thread
- ✓ Instructions from multiple threads can be mixed in the data path without confusing sources and destinations across the threads.

Multithreading can be built on top of an out-of-order processor by
- **adding a per-thread renaming table,**
- **keeping separate PCs, and**
- **providing the capability for instructions from multiple threads to commit**

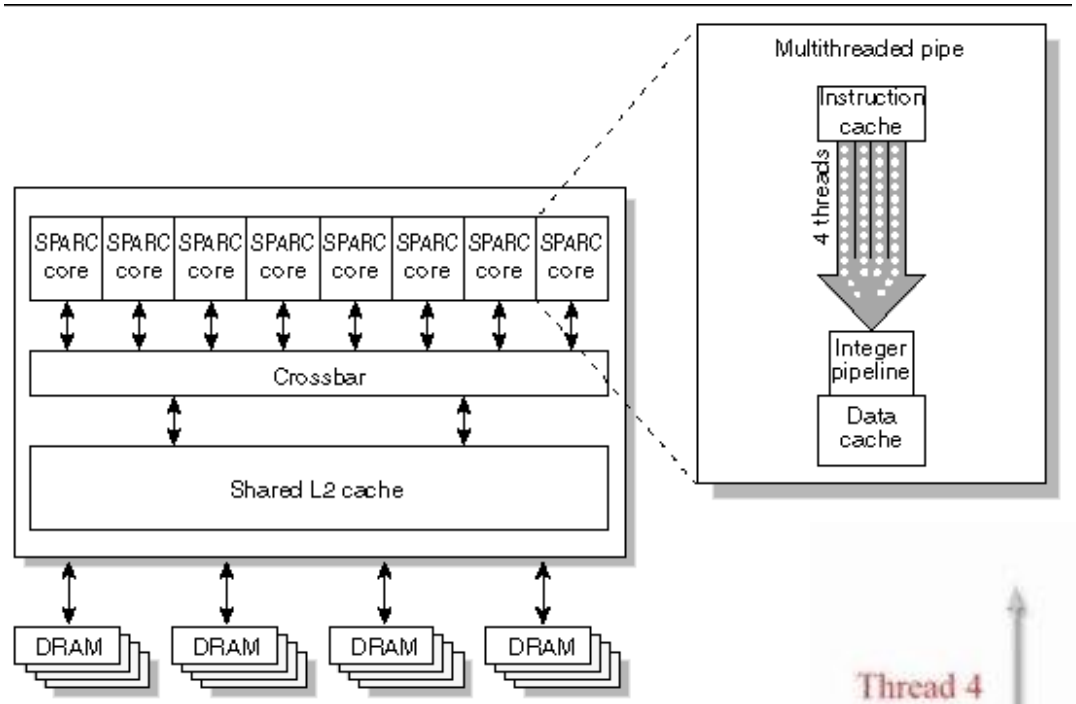
Design Challenges in SMT

- Dealing with a larger register file needed to hold multiple contexts
- Not affecting the clock cycle, particularly in critical steps such an instruction issue, where more candidate instructions need to be considered, and in instruction completion, where choosing what instructions to commit may be challenging
- Ensuring that the cache and TLB conflicts generated by the simultaneous execution of multiple threads do not cause significant performance degradation

Implementation Example: Effectiveness of Fine-Grained Multithreading on the Sun T1

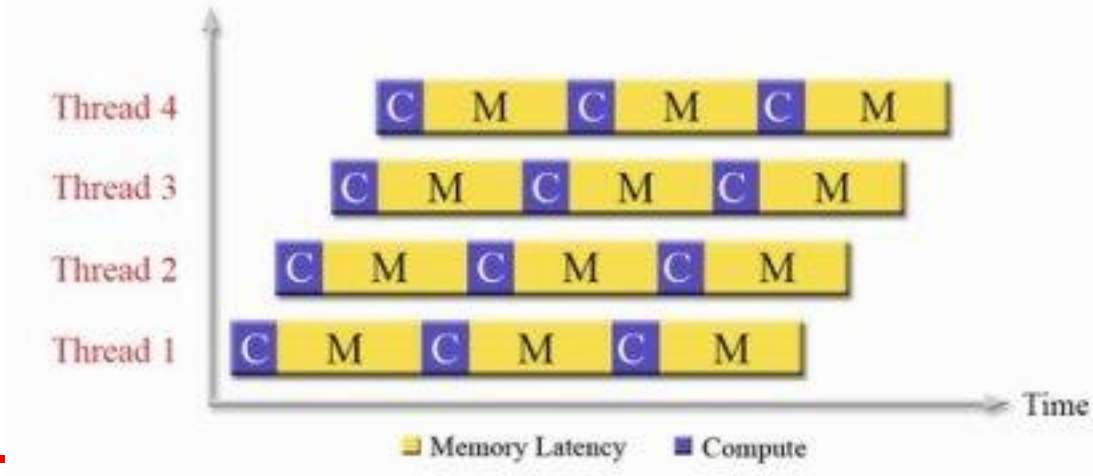
- Introduced in 2005 by Sun (acquired by Oracle)
- Focusing on exploiting TLP, using both multiple cores and multithreading to produce throughput
  - Eight processor cores, each supporting four threads
  - Simple six-stage, single-issue pipeline (one stage for thread switching)
- Fine-grained multithreading (but not SMT), switching to a new thread on each clock cycle.
  - The processor is idle only when all four threads are idle or stalled.
- 3-cycle delay after load/branch
  - Can be hidden by other threads.
- A single set of floating-point functional unit is shared by all eight cores.
  - Floating-point performance was not a focus!

Sun T1 Architecture and its Fine-Grain Multithreading

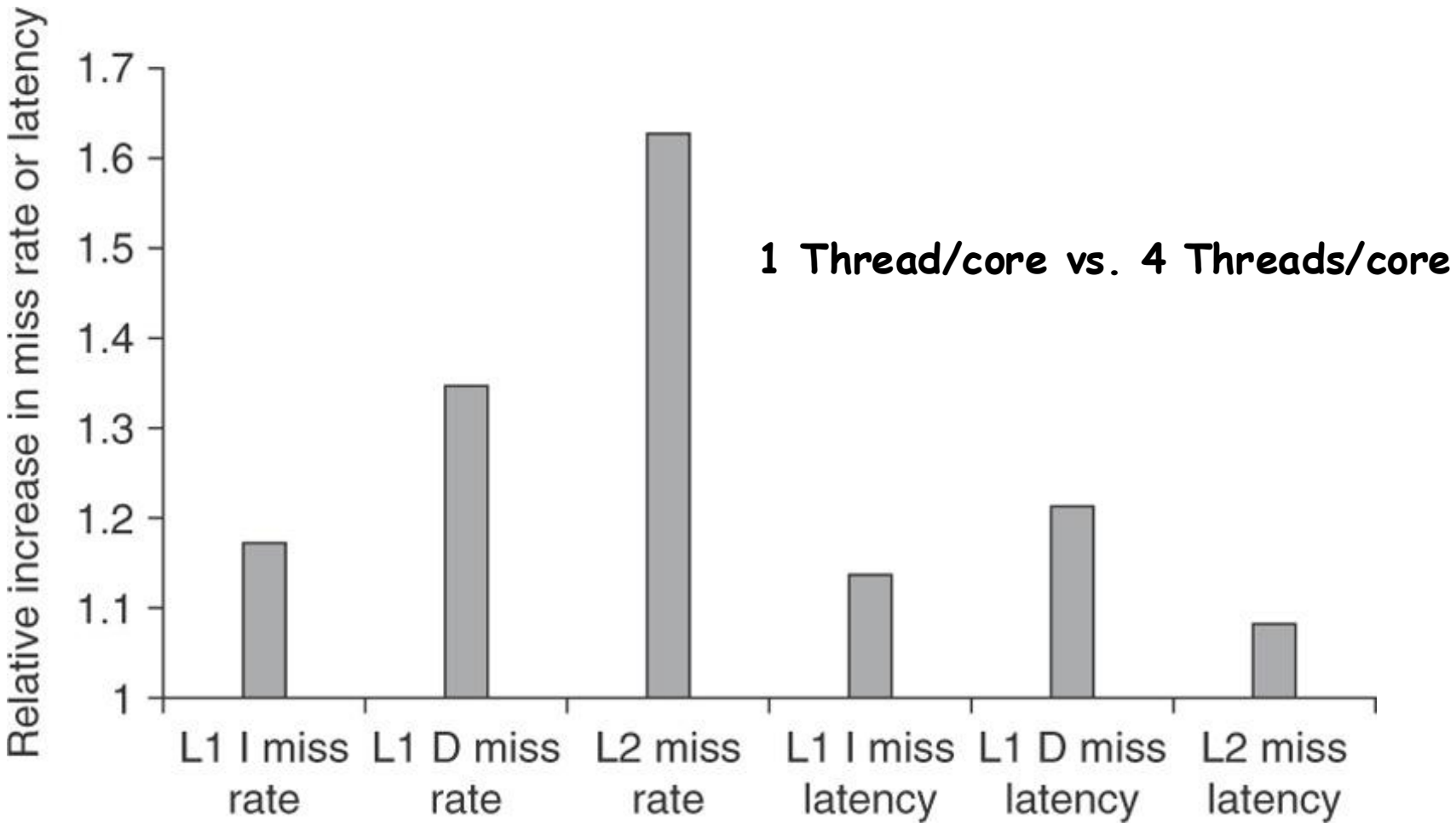


Processor Architecture Diagram

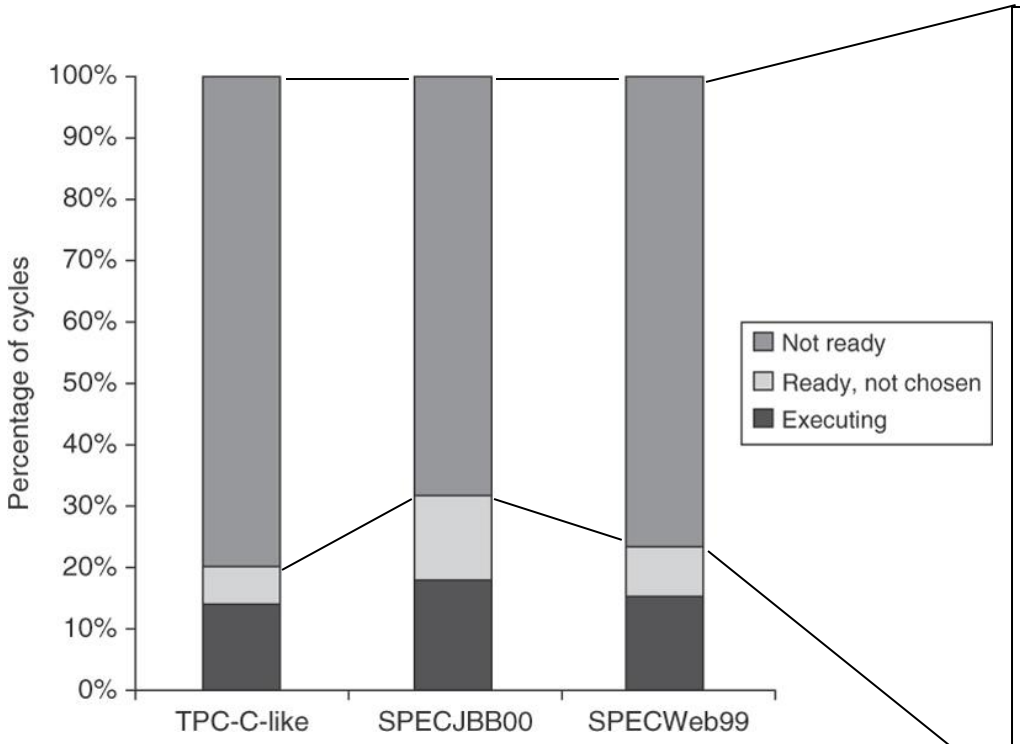
Fine-Grain Multithreading on T1

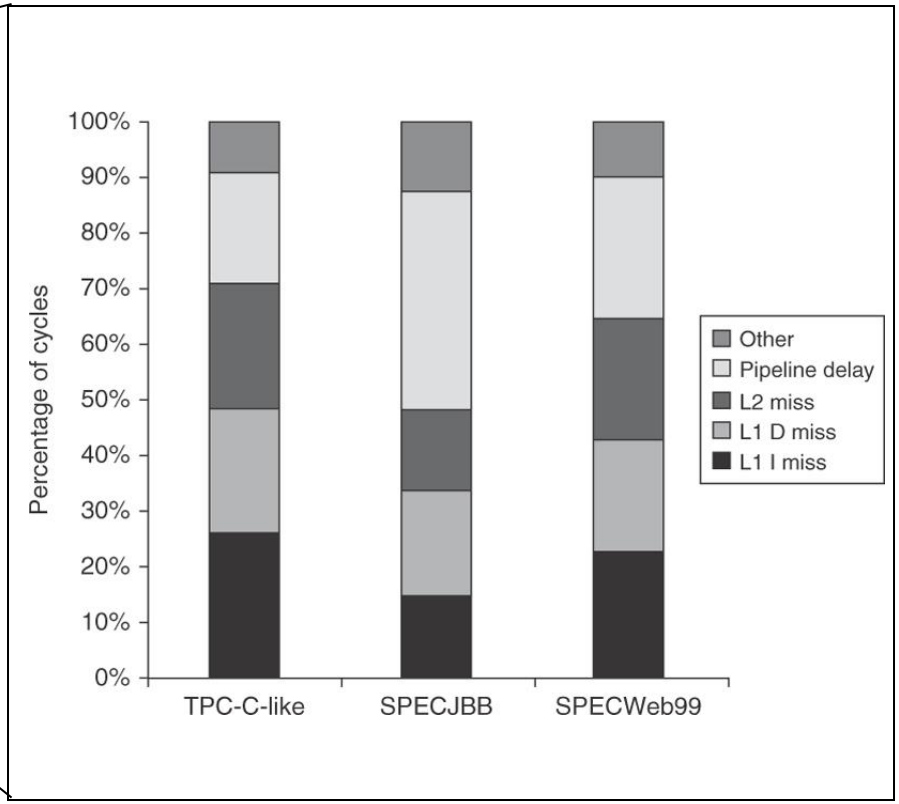


Relative Change in Miss Rate and Miss Latency due to Multithreading



Breakdown of the Status on an Average Thread





Benchmark	Per-thread CPI	Per-core CPI
TPC-C	7.2	1.80
SPECJBB	5.6	1.40
SPECWeb99	6.6	1.65

Ideal CPI/thread =4
Ideal CPI/core =1

Power5 SMT Implementation

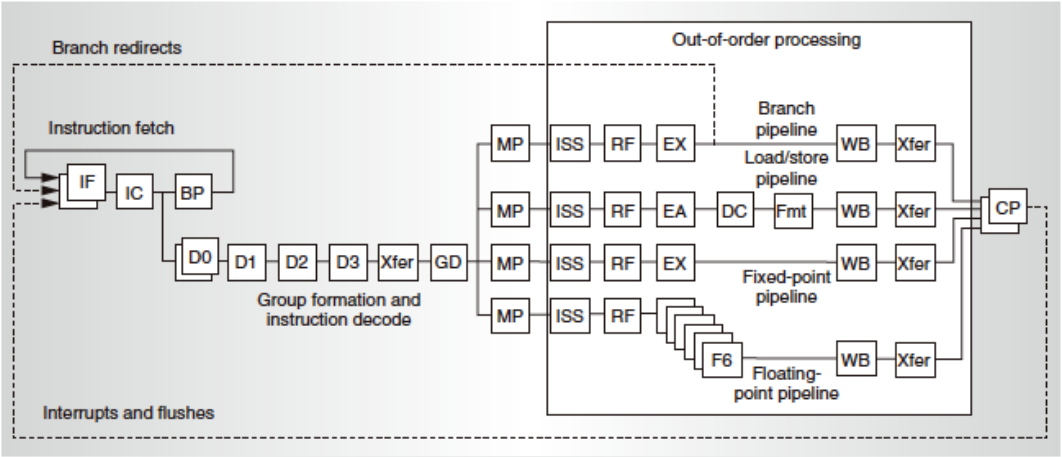


Figure 3. Power5 instruction pipeline (IF = instruction fetch, IC = instruction cache, BP = branch predict, D0 = decode stage 0, Xfer = transfer, GD = group dispatch, MP = mapping, ISS = instruction issue, RF = register file read, EX = execute, EA = compute address, DC = data caches, F6 = six-cycle floating-point execution pipe, Fmt = data format, WB = write back, and CP = group commit).

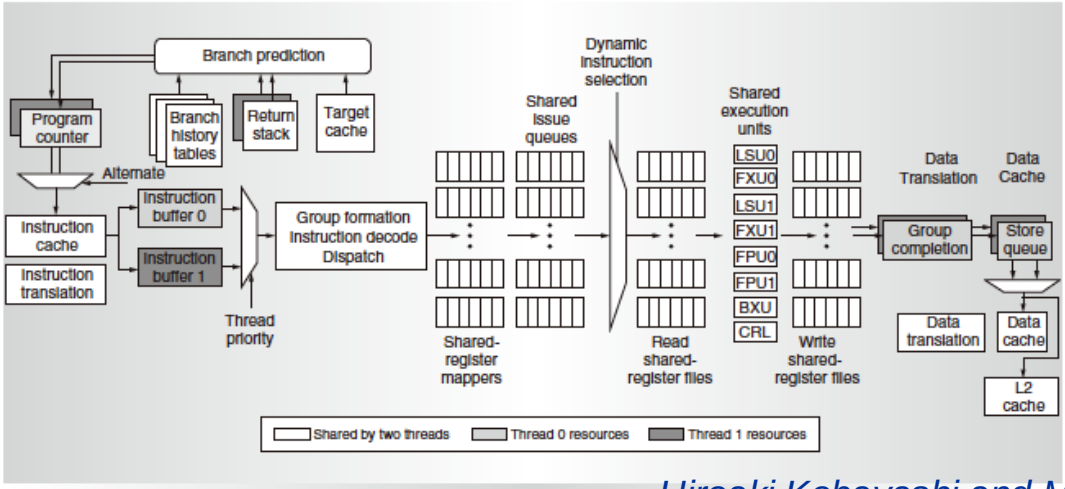


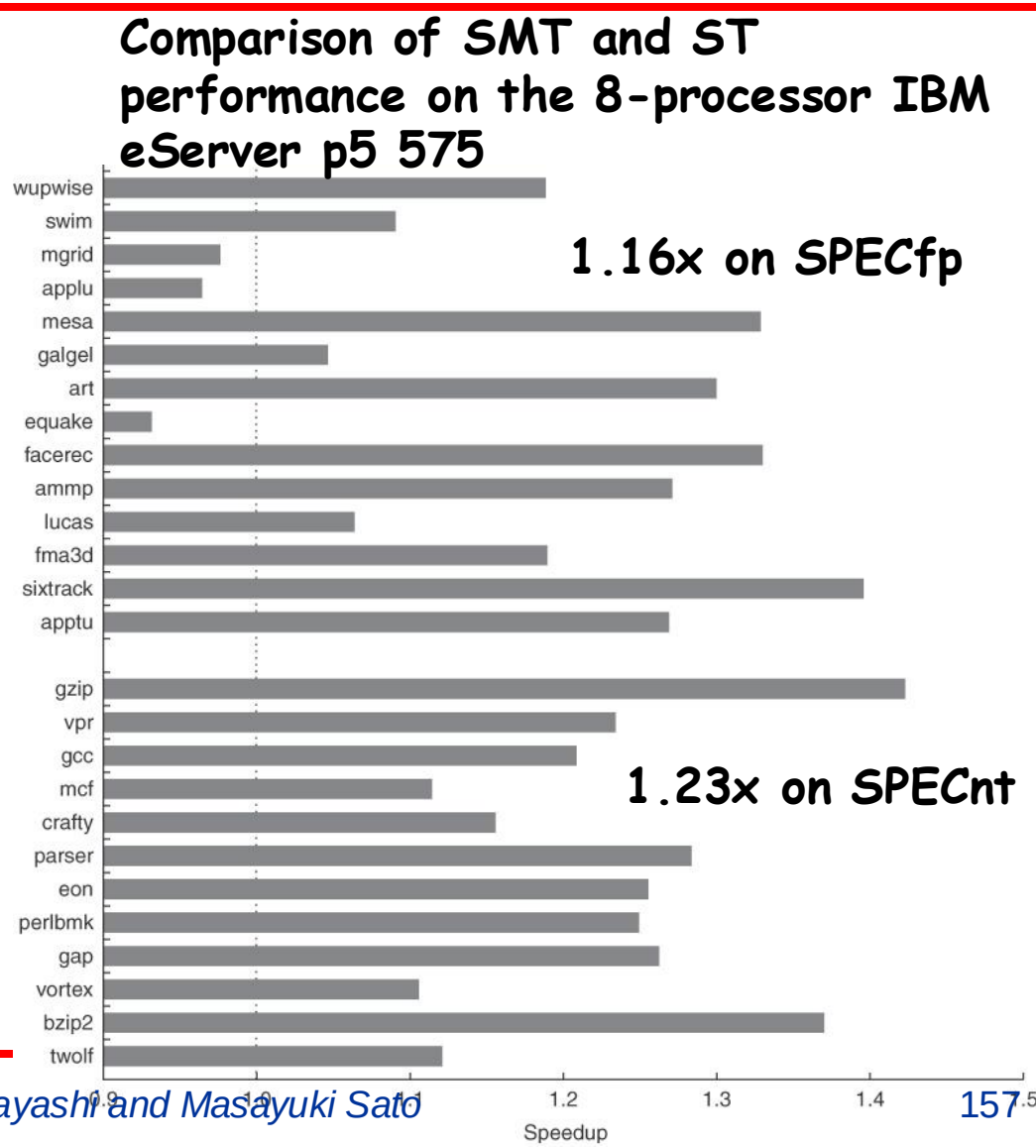
Figure 4. Power5 instruction data flow (BXU = branch execution unit and CRL = condition register logical execution unit).

Power5 used the same pipeline as the Power4, but it added SMT support:

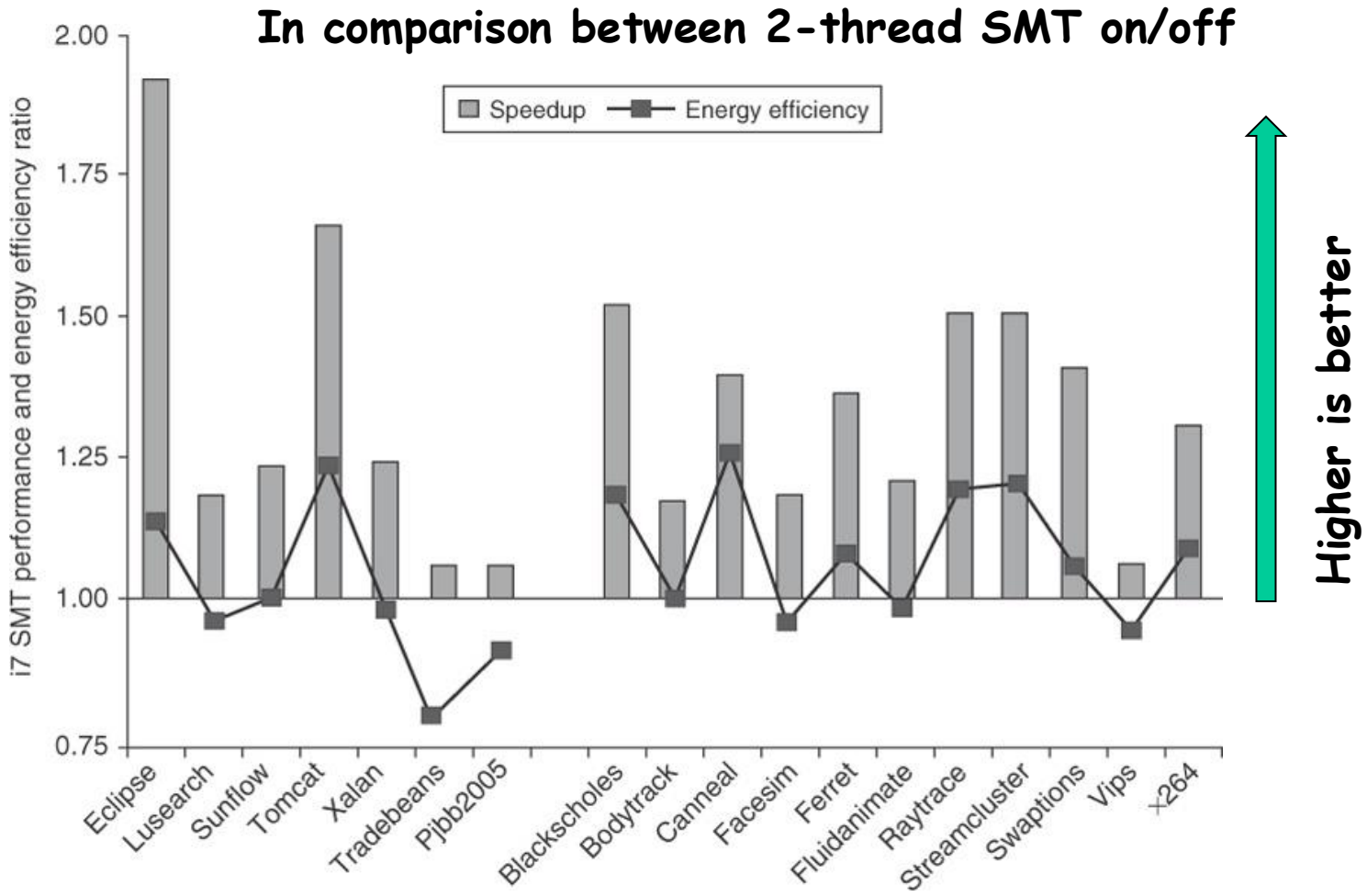
- Increasing the assiciativity of the L1 Inst. cache and I-TLB
- Adding per-thread load and store queues
- Increasing the size of L2 and L3 caches
- Adding separate instruction prefetch and buffering
- Increasing the number of virtual registers from 152 to 240
- Increasing the size of several issue queues

Potential Performance Advantages from SMT

- How much performance can be gained by implementing SMT???
  - Pentium4 Extreme
    - 1.01x on SPECint
    - 1.07 on SPECfp
    - 0.9 to 1.58 on 26 SPEC benches
  - IBM power5 (2005)
    - 0.93x (performance loss) to 1.41x



Core i7 SMT Performance and Energy Efficiency



What Limits Multiple-Issue Processors? (1/2)

- Modern microprocessors are primarily power limited.

  The key question is whether a technique is energy efficient!

  - Does it increase power consumption faster than it increases performance??
  - Unfortunately, the current techniques to boost the multiple-issue processors all have energy-inefficiency!

Reasons are

1. Issuing multiple instructions incurs some overhead in logic that grows faster than the issue rate grows
   - Overhead due to instruction issue analysis, including dependence checking, register renaming, and similar functions.
   - Without voltage reduction to decrease power, lower CPIs are likely to lead to lower ratios of performance per watt, simply due to overhead.

What Limits Multiple-Issue Processors? (2/2)

Reasons are

2. (more important!) Growing gap between peak issue rates and sustained performance.
   - If we want to *sustain* four instructions per clock, we must *fetch* more, *issue* more, and *initiate* execution on more than four instructions!
   - The power will be proportional to the peak rate, but performance will be at the sustained rate
3. Speculation is inefficient!
   - It can never be perfect! There is inherently waste in executing computations before we know whether they advance the program.
   - If speculation were perfect, it could save power, since it would reduce the execution time and save static power.

What Limits Multiple-Issue Processors? (1/2)

What about focusing on improving clock rate?

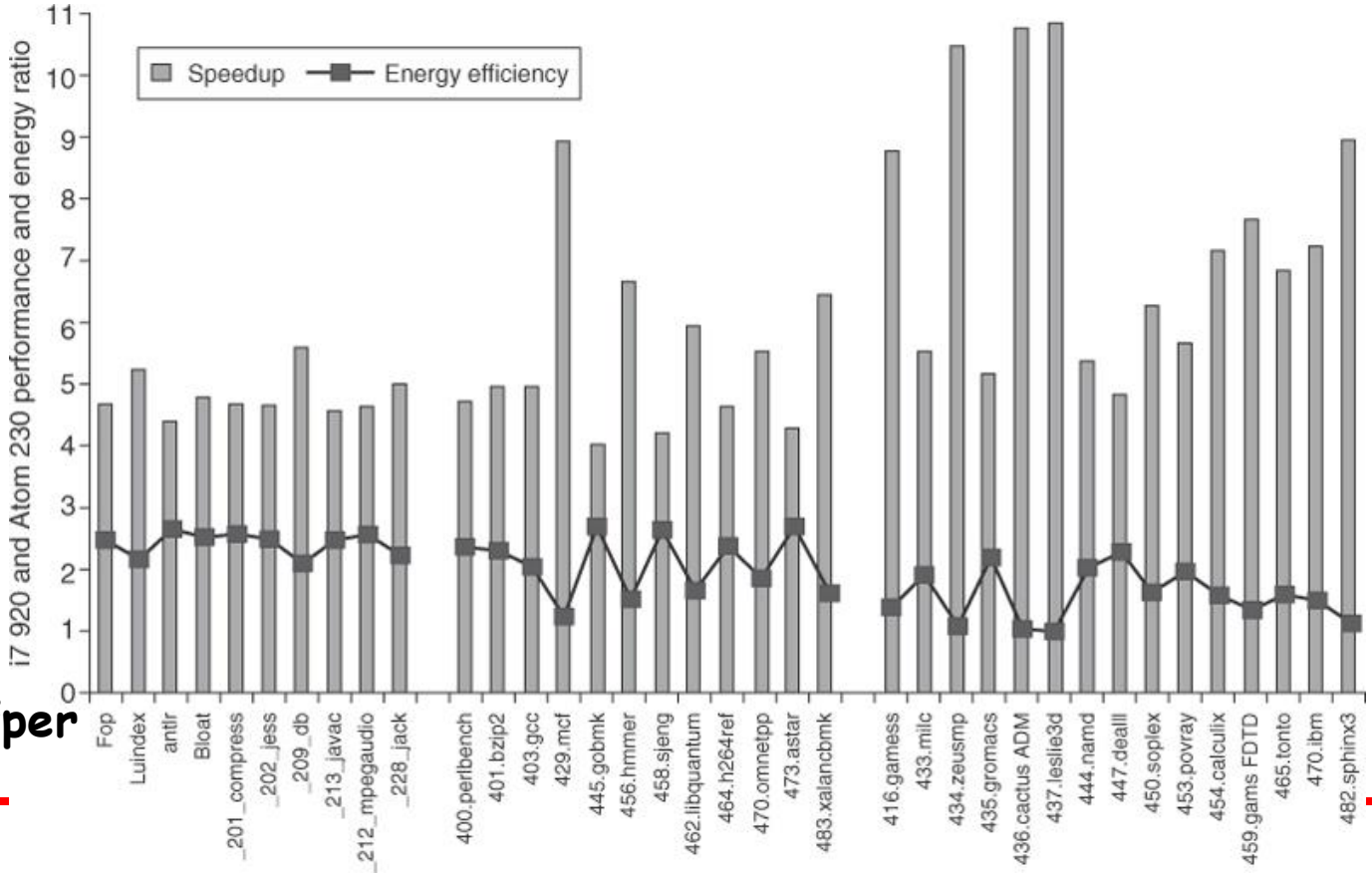
- Increasing the clock rate will increase transistor switching frequency and directly increase power consumption!
- To achieve a faster clock rate, need to increase pipeline depth, but deeper pipelines incur additional overhead penalties as well as causing higher switching rates.
- Example: Pentium III vs Pentium 4
  - It consumes roughly an amount of power proportional to the difference in clock rate, but its performance is somewhat less than the ratio of clock rates because of overhead and ILP limitations

Fallacies and Pitfalls

- Fallacy: *It is easy to predict the performance and energy efficiency of two different versions of the same instruction set architecture, if we hold the technology constant.*
  - Dismissing returns appear and silicon and energy costs of wider issue dominate the performance gain
  - Compiling for processors with significant amount of ILP has become extremely complex.
- Pitfall: *Improving only one aspect of a multiple-issue processor and expecting overall performance improvement*
  - Many factors limit the performance of multiple-issue machines, and improving one aspect of a processor often exposes some other aspect that previously did not limit performance.
    - Branch prediction, register renaming, size of issue windows etc…

Fallacies and Pitfalls

- Fallacy: *It is easy to predict the performance and energy efficiency of two different versions of the same instruction set architecture, if we hold the technology constant.*



PerfRatio
$= Perf_{i7}/Perf_{atom}$

Energy efficiencyRatio
=
$PerfperEnergy_{atom}/PerfperEnergy_{i7}$

Overview of i7, ARM A8 and Atom 230

Area	Specific characteristic	Intel i7 920 Four cores, each with FP	ARM A8 One core, no FP	Intel Atom 230 One core, with FP
Physical chip properties	Clock rate	2.66 GHz	1 GHz	1.66 GHz
	Thermal design power	130 W	2 W	4 W
	Package	1366-pin BGA	522-pin BGA	437-pin BGA
Memory system	TLB	Two-level All four-way set associative 128 I/64 D 512 L2	One-level fully associative 32 I/32 D	Two-level All four-way set associative 16 I/16 D 64 L2
	Caches	Three-level 32 KB/32 KB 256 KB 2–8 MB	Two-level 16/16 or 32/32 KB 128 KB–1MB	Two-level 32/24 KB 512 KB
	Peak memory BW	17 GB/sec	12 GB/sec	8 GB/sec
Pipeline structure	Peak issue rate	4 ops/clock with fusion	2 ops/clock	2 ops/clock
	Pipeline scheduling	Speculating out of order	In-order dynamic issue	In-order dynamic issue
	Branch prediction	Two-level	Two-level 512-entry BTB 4K global history 8-entry return stack	Two-level

Fallacies and Pitfalls

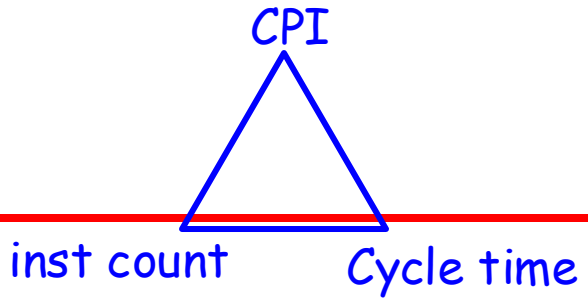
- Fallacy: *Processor with lower CPIs will always be faster*
- Fallacy: *Processor with faster clock rates will always be faster*

Performance determined by the product of CPI and clock rate!

Processor	Clock rate	SPECCInt2006 base	SPECCFP2006 baseline
Intel Pentium 4 670	3.8 GHz	11.5	12.2
Intel Itanium -2	1.66 GHz	14.5	17.3
Intel i7	3.3 GHz	35.5	38.4

Pentium 4 was the most aggressively pipelined processor with over 20 stages!
larger CPI due to limited ILP exploitation and
a larger branch misprediction penalty.

5) Processor performance equation



$$\text{CPU time} = \frac{\text{Seconds}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Cycle}}$$

	Inst Count	CPI	Clock Rate
Program	X		
Compiler	X		
Inst. Set.	X	X	
Organization		X	X
Technology			X

Concluding Remarks

- As 2000 began, the tremendous interest in multiple-issue organizations came about because of an interest in improving performance without affecting the standard uniprocessor programming model
  - Raising the clock rates of multiple-issue processors in the last 10 years.
  - Similar architectures with dynamically scheduled, multiple-issue (3 to 4 inst./clock) processors such as Power5, Athlon, Pentim4 but
  - 10-20x in clock rates, 4-8x in cache capacity, 2-4x in renaming registers, and 2x in load-store units
- However, its unclear that the initial success in achieving high-clock-rate processors that issue three to four instructions per clock can be carried much further due to limitation in available ILP, efficiency in exploiting that ILP, and power concerns.
- The advantage in higher clock rate obtained by very deeper pipelines (20-30 stages) is largely lost by additional pipeline stalls
  - Trade-offs in pipeline depth, issue rate, and other characteristics

Shifting to multiple cores at a lower clock rate on a chip instead seeking higher-clock rate version of a single core on a chip by taking advantage of TLP and DLP!

Characteristics of Four IBM Power Processors

	Power4	Power5	Power6	Power7
Introduced	2001	2004	2007	2010
Initial clock rate (GHz)	1.3	1.9	4.7	3.6
Transistor count (M)	174	276	790	1200
Issues per clock	5	5	7	6
Functional units	8	8	9	12
Cores/chip	2	2	2	8
SMT threads	0	2	2	4
Total on-chip cache (MB)	1.5	2	4.1	32.3

For ILP

For TLP

- Modest improvement in the ILP support, but
- the dominant portion of the increase in transistor count went to increasing the caches and core numbers.