

CS422

Computer Architecture

Spring 2004

Lecture 17, 26 Feb 2004

Bhaskaran Raman
Department of CSE
IIT Kanpur

<http://web.cse.iitk.ac.in/~cs422/index.html>

Speculation

- Wish to move instructions across branches
 - To eliminate possible stalls
 - For better scheduling
 - Appropriate conditional instructions may not always exist
- Example:

```
if (N == 0) {  
    A = *X;  
} else {  
    A++;  
}
```

Speculation: An Example

LW	R1, 0(R2)	// Load N		LW	R1, 0(R2)	// Load N
BNEZ	R1, L1	// Test N		LW	R3, 0(R4)	// Load X
LW	R3, 0(R4)	// Load X		LW	R5, 0(R3)	// Load *X
LW	R5, 0(R3)	// Load *X	→	BEQZ	R1, L3	// Test N
JMP	L2	// Skip else		LW	R5, 0(R6)	// Load A
L1: LW	R5, 0(R6)	// Load A		ADDI	R5, R5, #1	// Incrmt.
ADDI	R5, R5, #1	// Incrmt.		L2: SW	0(R6), R5	// Store A
L2: SW	0(R6), R3	// Store A				

- Compiler predicts that the “then” clause is most likely
- Speculatively schedules the “then” clause
 - Eliminates 2 stalls, and the JMP instruction

Exception Behaviour

LW **R1, 0(R2)** **// Load N**
LW **R3, 0(R4)** **// Load X**
LW **R5, 0(R3)** **// Load *X**
BEQZ **R1, L3** **// Test N**
LW **R5, 0(R6)** **// Load A**
ADDI **R5, R5, #1** **// Incrmt.**

L2: SW **0(R6), R5** **// Store A**

- Terminating vs. non-terminating exceptions
- While doing such scheduling:
 - Correct program ==> no extra terminating exceptions
 - Incorrect program ==> should preserve any terminating exceptions

Preserving Exception Behaviour

- **Approach 1:** ignore terminating exceptions for speculated instructions
 - Incorrect programs may not be terminated

```
LW      R1, 0(R2)    // Load N
LW*     R3, 0(R4)    // Load X, speculated
LW*     R5, 0(R3)    // Load *X, speculated
BEQZ    R1, L3       // Test N
LW      R5, 0(R6)    // Load A
ADDI    R5, R5, #1   // Incrmt.
L2: SW   0(R6), R5   // Store A
```

Preserving Exception Behaviour (continued)

- **Approach 2: *poison bits***
 - Set poison bit in result register of conditional instruction, if exception occurs
 - Raise exception if any other instruction uses that register

LW R1, 0(R2) // Load N

LW* R3, 0(R4) // Load X, set poison bit on exception

LW* R5, 0(R3) // Load *X, set poison bit on exception

BEQZ R1, L3 // Test N

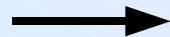
LW R5, 0(R6) // Load A

ADDI R5, R5, #1 // Incrmt.

L2: SW 0(R6), R5 // Store A

Preserving Exception Behaviour (continued)

```
if (N == 0) {  
    A = *X;  
    N++;  
} else {  
    A++;  
}
```



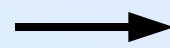
```
LW    R1, 0(R2)    // Load N  
LW*   R3, 0(R4)    // Load X, speculative  
ADDI* R10, R1, #1  // N++, speculative  
LW*   R5, 0(R3)    // Load *X, speculative  
BEQZ  R1, L3       // Test N  
LW    R5, 0(R6)    // Load A  
ADDI  R5, R5, #1   // Incrmt.  
MOV   R10, R1      // So that R10 has N  
L2: SW    0(R6), R5 // Store A
```

- Extra register R10 gets used up
- Extra instruction in “else” clause

Preserving Exception Behaviour (continued)

- **Approach 3: buffer results**
 - Instructions *boosted* past branches, flagged as boosted (in opcode)
 - Results of boosted instructions forwarded and used, like in Tomasulo
 - When branch is reached, result of speculation is checked
 - Result committed if prediction correct
 - Result discarded otherwise
 - Solution close to fully hardware-based speculation

Boosted Instructions: An Example

<pre> if (N == 0) { A = *X; N++; } else { A++; } </pre>		<pre> LW R1, 0(R2) // Load N LW+ R3, 0(R4) // Load X, boosted ADDI+ R1, R1, #1 // N++, boosted LW+ R5, 0(R3) // Load *X, boosted BEQZ R1, L3 // Test N LW R5, 0(R6) // Load A ADDI R5, R5, #1 // Incrmt. L2: SW 0(R6), R5 // Store A </pre>
---	---	---

- The “+” denotes a boosted instruction, and is boosted across the next branch, which is predicted taken

Hardware-Based Speculation

- Combination of branch prediction, speculation, and dynamic scheduling
- *Data flow execution*: instruction executes as soon as the data it requires is ready
- Advantages over software approach:
 - Memory disambiguation is better
 - Better branch prediction
 - Precise exception model
 - No book-keeping code
 - Works for “old” software too
- Disadvantage: hardware cost and complexity

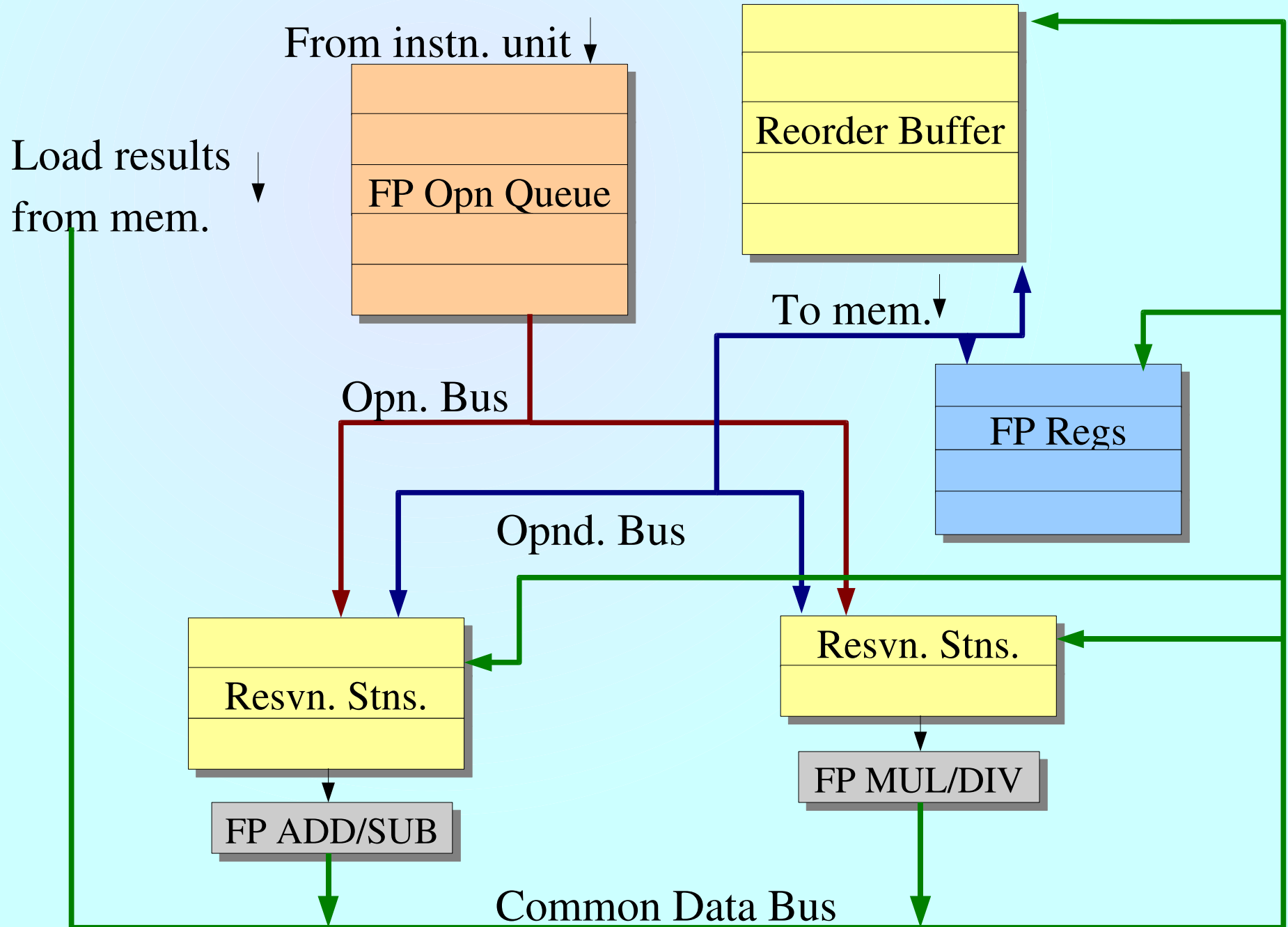
Speculation in Tomasulo

- Speculate using branch prediction
- Go ahead and execute based on speculation
- Use results of speculated instructions for other instructions, just as in Tomasulo
- But, *commit* result only after knowing if speculation was correct
 - *In-order* commit
 - Using *reorder buffer*
 - Also achieves *precise exceptions*

The Reorder Buffer

- Similar to the store buffer in functionality
- Replaces the store and load buffers
- Virtual registers are the reorder buffer entries
 - The reservation stations are not virtual registers anymore
- Reorder Buffer Data Structure
 - Instruction type: branch, store, or ALU/load
 - Destination: register or memory location
 - Value: which has to be committed

Tomasulo Using the Reorder Buffer



Pipeline Stages

- Issue, EX, WB, Commit
- Issue allocates a reorder buffer entry
 - Entries allocated in circular fashion
- Commit writes result back to destination
 - Frees up the reorder buffer entry
 - For branch instruction
 - Prediction correct ==> commit
 - Else, flush reorder buffer

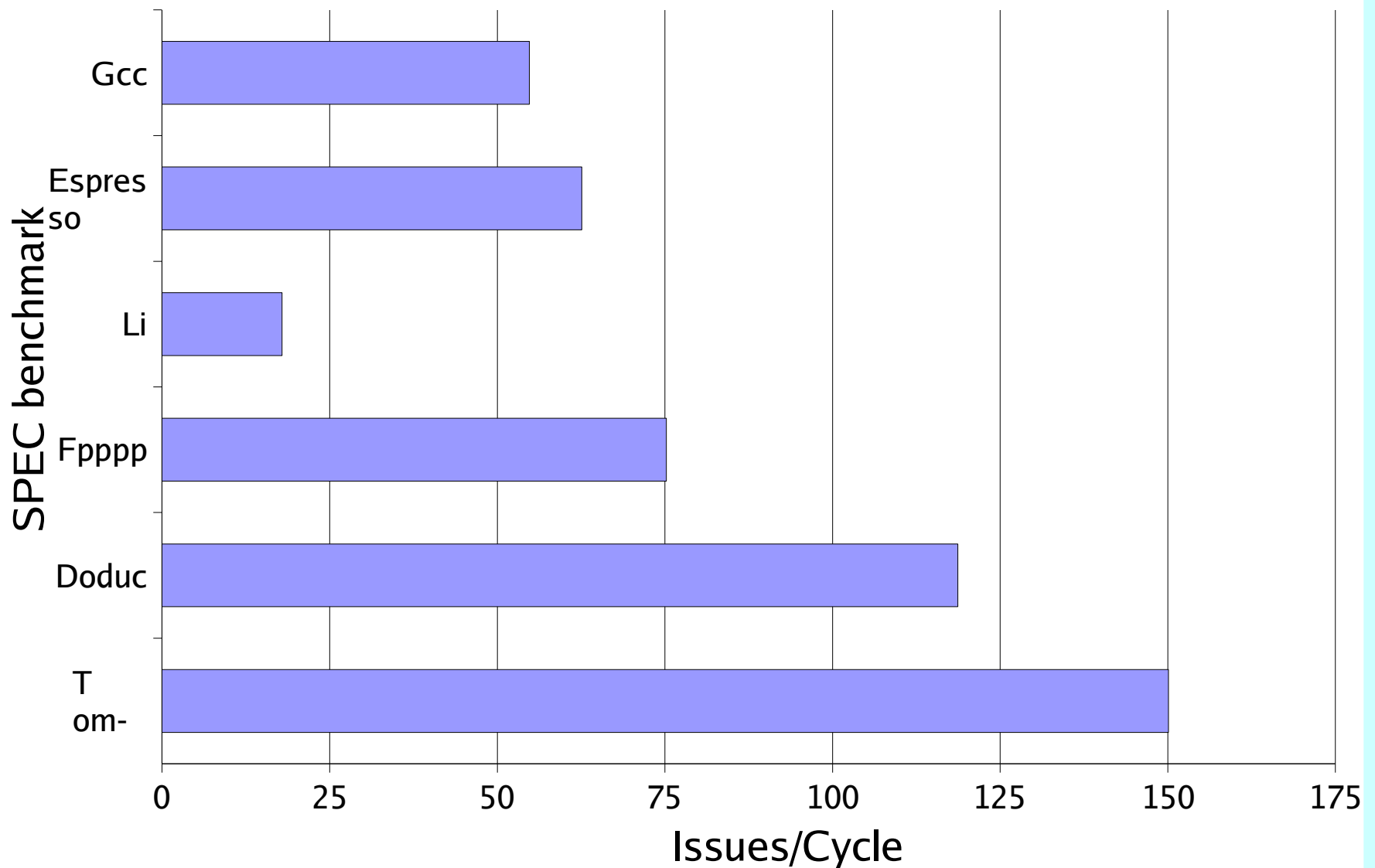
Summary of ILP Techniques

- Software techniques
 - Compiler scheduling, Loop unrolling, Software pipelining, Trace scheduling (VLIW), Static branch prediction, Speculation
- Hardware support for software
 - Conditional instructions, poison bits
- Hardware techniques
 - Hardware scheduling, Dynamic branch prediction, Hardware speculation
- Which hardware technique(s) to use?

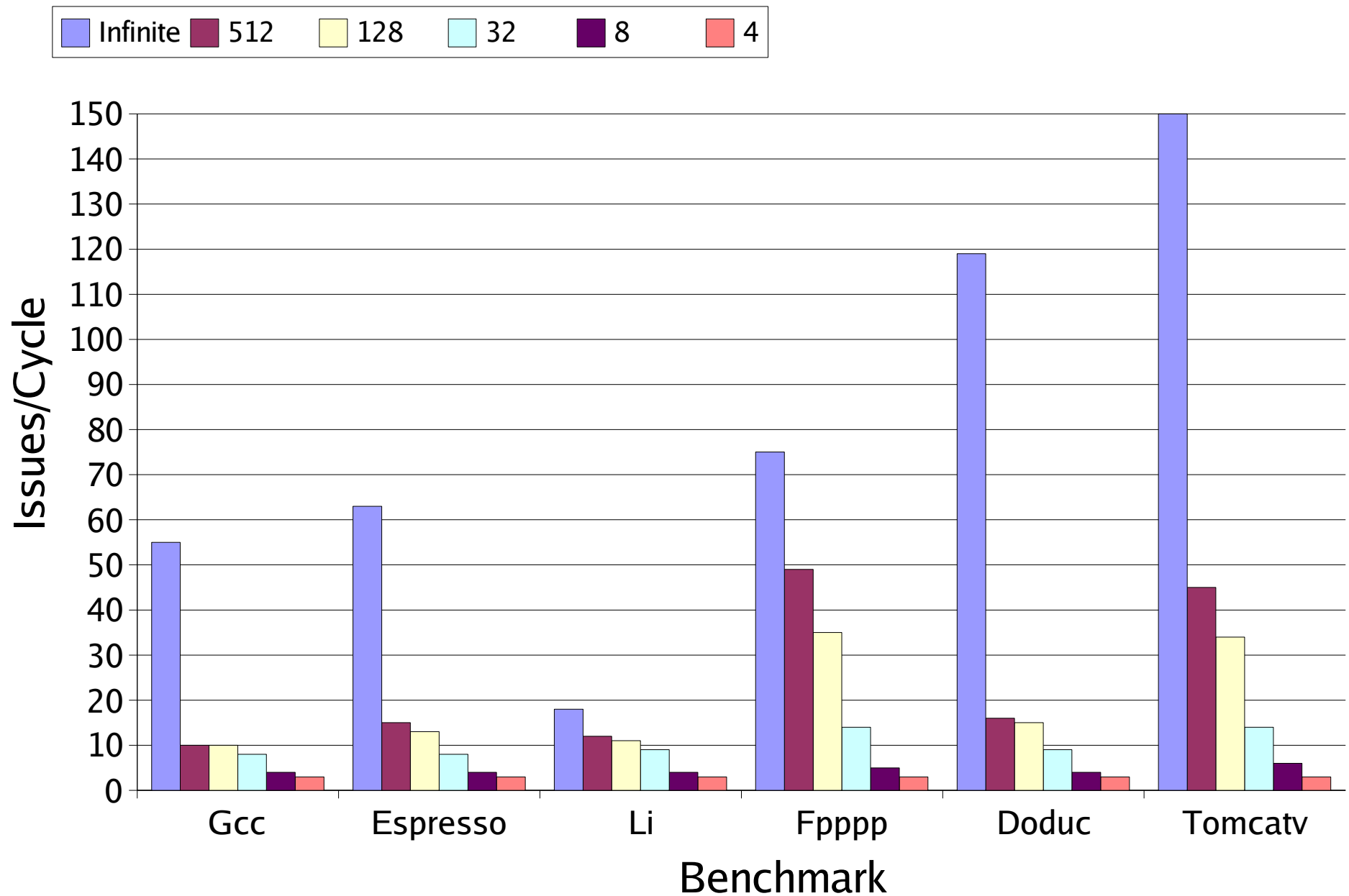
How Much ILP is Available?

- Assume infinite hardware resources
 - Infinite virtual registers
 - Perfect branch prediction, jump prediction
 - Perfect memory disambiguation
- Every instruction is scheduled as early as possible
 - Restricted only by data flow

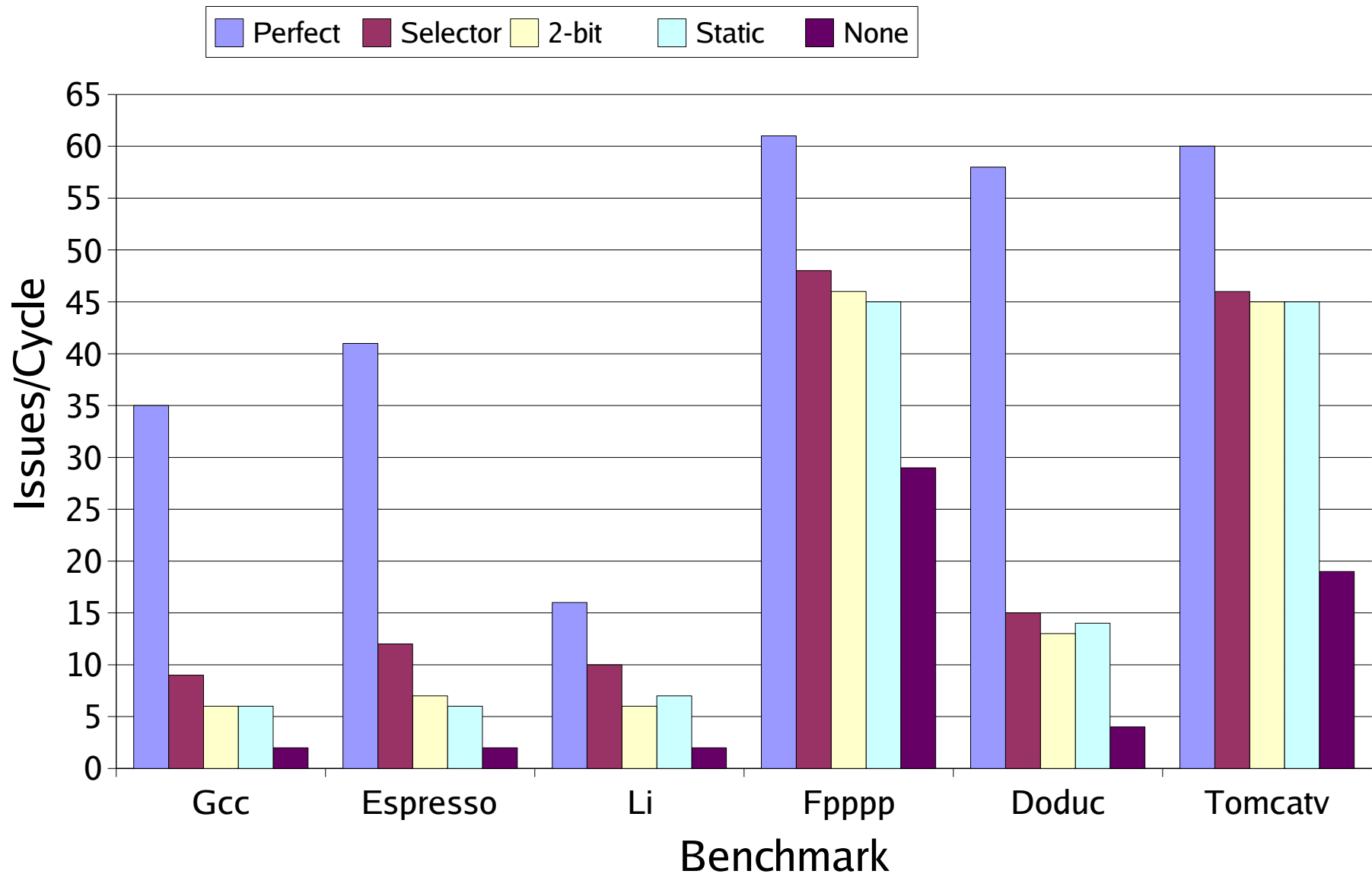
Available ILP in Programs



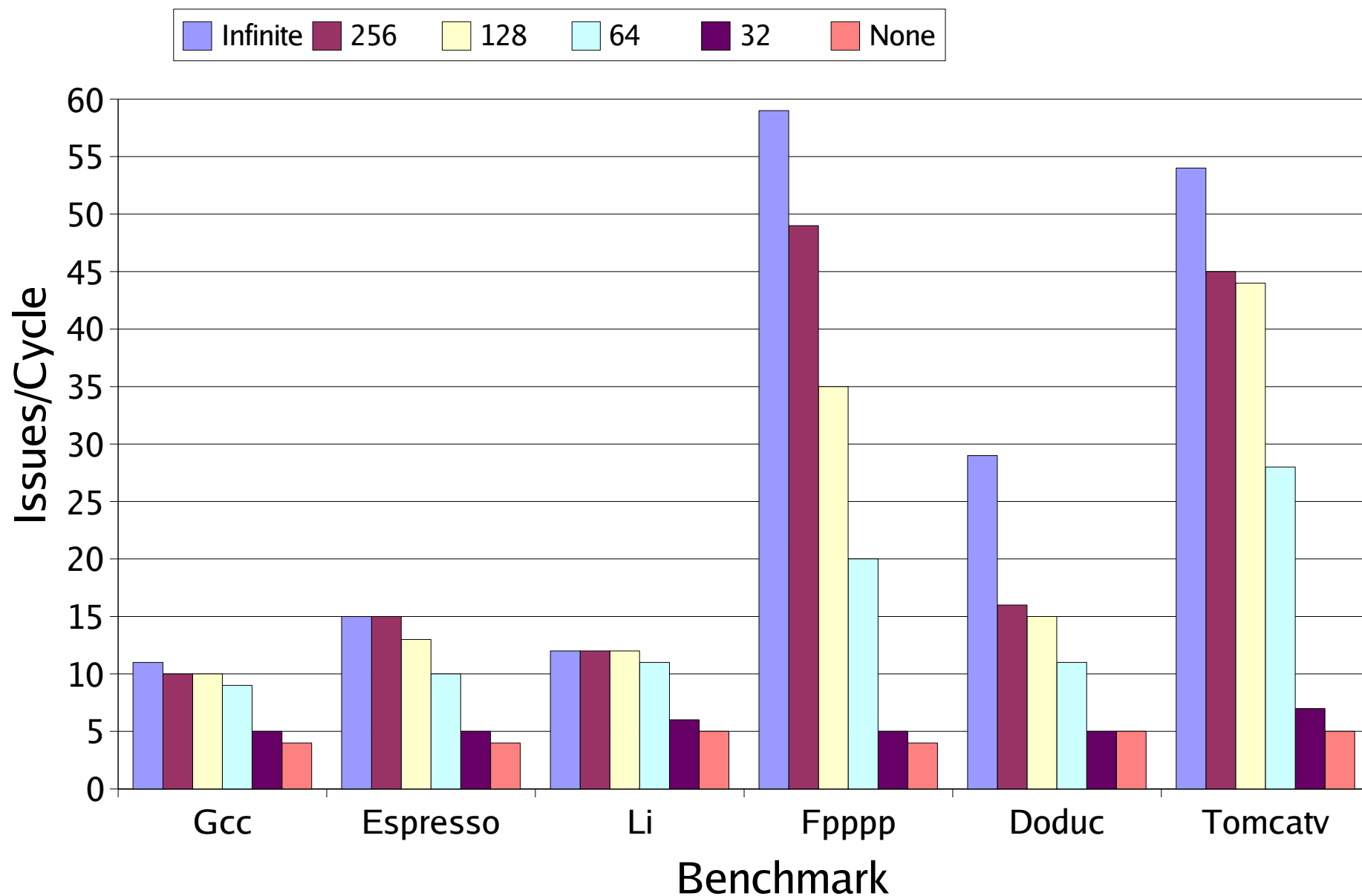
Window Size Limitation



Effect of Imperfect Branch Predictions



Effect of Finite Virtual Register Set



A Realizable Processor

- Up to 64 instruction issues per cycle
- Selective predictor with 1K entries, and a 16-entry return predictor
- Perfect memory disambiguation
- Register renaming with 64 integer virtual registers, and 64 FP virtual registers

ILP for a Realizable Processor

