Lecture 5: VLIW, Software Pipelining, and Limits to ILP

Professor David A. Patterson
Computer Science 252
Fall 1996

Review: Tomasulo

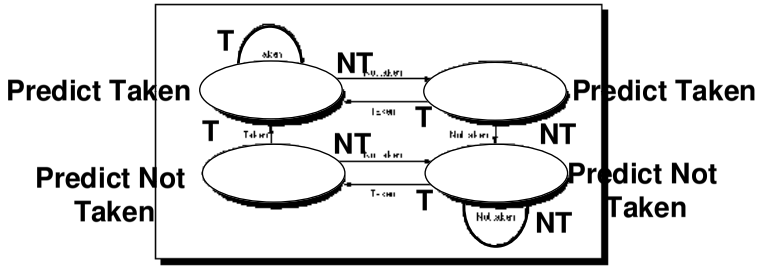
- Prevents Register as bottleneck
- Avoids WAR, WAW hazards of Scoreboard
- Allows loop unrolling in HW
- Not limited to basic blocks (provided branch prediction)
- Lasting Contributions
  - Dynamic scheduling
  - Register renaming
  - Load/store disambiguation
- 360/91 descendants are PowerPC 604, 620; MIPS R10000; HP-PA 8000; Intel Pentium Pro

Dynamic Branch Prediction

- Performance = *ƒ*(accuracy, cost of misprediction)
- Branch History Table is simplest
  - Lower bits of PC address index table of 1-bit values
  - Says whether or not branch taken last time
  - No address check
- Problem: in a loop, 1-bit BHT will cause two mispredictions (avg is 9 iteratios before exit):
  - End of loop case, when it exits instead of looping as before
  - First time through loop on *next* time through code, when it predicts exit instead of looping

Dynamic Branch Prediction

- Solution: 2-bit scheme where change prediction only if get misprediction *twice:* (Figure 4.13, p. 264)



- Red: stop, not taken
- Green: go, taken

BHT Accuracy

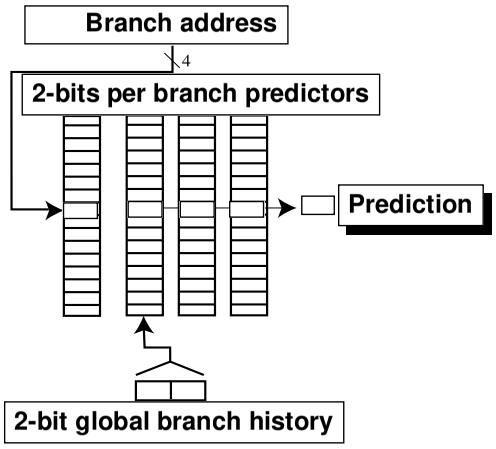
- Mispredict because either:
  - Wrong guess for that branch
  - Got branch history of wrong branch when index the table
- 4096 entry table programs vary from 1% misprediction (nasa7, tomcatv) to 18% (eqntott), with spice at 9% and gcc at 12%
- 4096 about as good as infinite table (in Alpha 211164)

Correlating Branches

- Hypothesis: recent branches are correlated; that is, behavior of recently executed branches affects prediction of current branch; ie., they are correlated
- Idea: record m most recently executed branches as taken or not taken, and use that pattern to select the proper branch history table
- In general, (m,n) predictor means record last m branches to select between 2^m history talbes each with n-bit counters
  - Old 2-bit BHT is then a (0,2) predictor

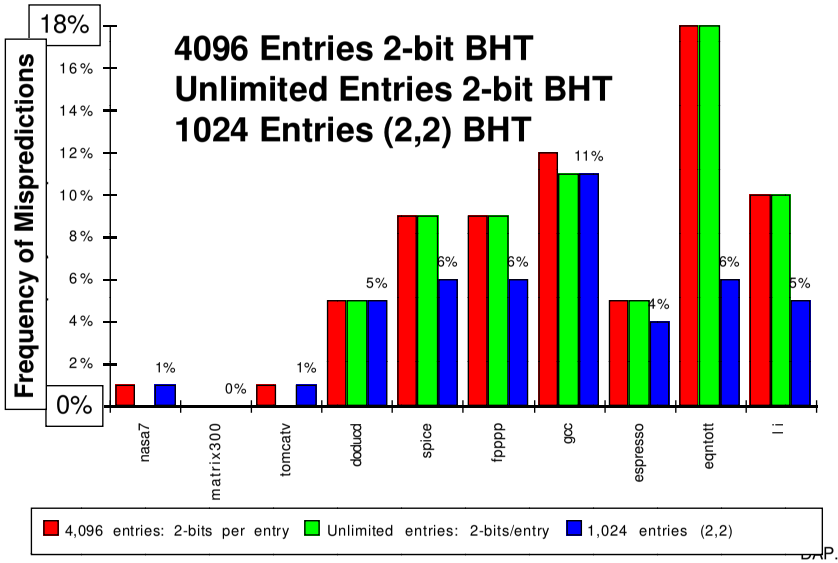
Correlating Branches

(2,2) predictor
- Then behavior of recent branches selects between, say, four predictions of next branch, updating just that prediction



Accuracy of Different Schemes

(Figure 4.21, p. 272)



Re-evaluating Correlation

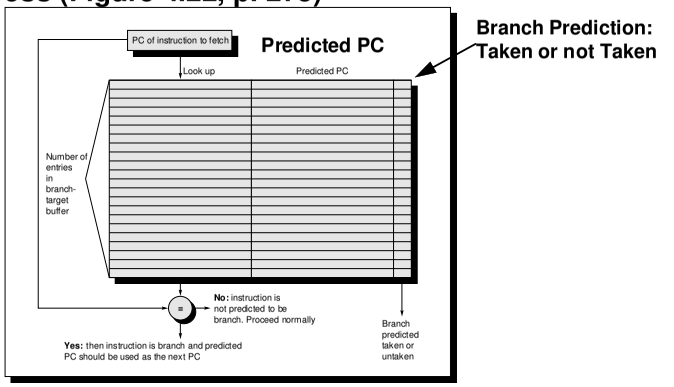
- Several of the SPEC benchmarks have less than a dozen branches responsible for 90% of taken branches:

program	% brances	static	# = 90% branches
compress	14%	236	13
eqntott	25%	494	5
gcc	15%	9531	2020
mpeg	10%	5598	532
real gcc	13%	17361	3214

- Real programs + OS more like gcc
- Small benefits beyond benchmarks for correlation? problems with branch aliases

Need Address @ Same Time as Prediction

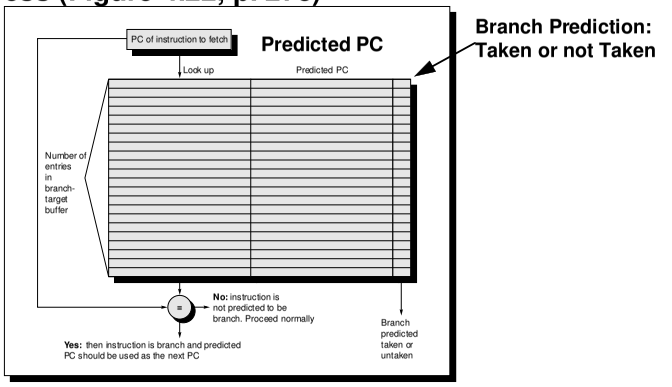
- Branch Target Buffer (BTB): Address of branch index to get prediction AND branch address (if taken)
  - Note: must check for branch match now, since can't use wrong branch address (Figure 4.22, p. 273)



- Return instruction addresses predicted with stack

Need Address @ Same Time as Prediction

- Branch Target Buffer (BTB): Address of branch index to get prediction AND branch address (if taken)
  - Note: must check for branch match now, since can't use wrong branch address (Figure 4.22, p. 273)



- Return instruction addresses predicted with stack

Dynamic Branch Prediction Summary

- Branch History Table: 2 bits for loop accuracy
- Correlation: Recently executed branches correlated with next branch
- Branch Target Buffer: include branch address & prediction

Getting CPI < 1: Issuing Multiple Instructions/Cycle

- Two variations
- Superscalar: varying no. instructions/cycle (1 to 8), scheduled by compiler or by HW (Tomasulo)
  - IBM PowerPC, Sun SuperSparc, DEC Alpha, HP 7100
- Very Long Instruction Words (VLIW): fixed number of instructions (16) scheduled by the compiler
  - Joint HP/Intel agreement in 1998?

Getting CPI < 1: Issuing Multiple Instructions/Cycle

- Superscalar DLX: 2 instructions, 1 FP & 1 anything else
  - Fetch 64-bits/clock cycle; Int on left, FP on right
  - Can only issue 2nd instruction if 1st instruction issues
  - More ports for FP registers to do FP load & FP op in a pair

Type	*PipeStages*						
Int. instruction	IF	ID	EX	MEM	WB		
FP instruction	IF	ID	EX	MEM	WB		
Int. instruction		IF	ID	EX	MEM	WB	
FP instruction		IF	ID	EX	MEM	WB	
Int. instruction			IF	ID	EX	MEM	WB
FP instruction			IF	ID	EX	MEM	WB

- 1 cycle load delay expands to 3 instructions in SS
  - instruction in right half can't use it, nor instructions in next slot

Unrolled Loop that Minimizes Stalls for Scalar

```
1 Loop: LD     F0,0(R1)
2       LD     F6,-8(R1)
3       LD     F10,-16(R1)
4       LD     F14,-24(R1)
5       ADDD   F4,F0,F2
6       ADDD   F8,F6,F2
7       ADDD   F12,F10,F2
8       ADDD   F16,F14,F2
9       SD     0(R1),F4
10      SD     -8(R1),F8
11      SD     -16(R1),F12
12      SUBI   R1,R1,#32
13      BNEZ   R1,LOOP
14      SD     8(R1),F16    ; 8-32 = -24
```

LD to ADDD: 1 Cycle
ADDD to SD: 2 Cycles

14 clock cycles, or 3.5 per iteration

Loop Unrolling in Superscalar

	Integer instruction	*FP instruction*	*Clock cycle*
Loop:	LD F0,0(R1)		1
	LD F6,-8(R1)		2
	LD F10,-16(R1)	ADDD F4,F0,F2	3
	LD F14,-24(R1)	ADDD F8,F6,F2	4
	LD F18,-32(R1)	ADDD F12,F10,F2	5
	SD 0(R1),F4	ADDD F16,F14,F2	6
	SD -8(R1),F8	ADDD F20,F18,F2	7
	SD -16(R1),F12		8
	SD -24(R1),F16		9
	SUBI R1,R1,#40		10
	BNEZ R1,LOOP		11
	SD -32(R1),F20		12

- Unrolled 5 times to avoid delays (+1 due to SS)
- 12 clocks, or 2.4 clocks per iteration

Limits of Superscalar

- While Integer/FP split is simple for the HW, get CPI of 0.5 only for programs with:
  - Exactly 50% FP operations
  - No hazards
- If more instructions issue at same time, greater difficulty of decode and issue
  - Even 2-scalar => examine 2 opcodes, 6 register specifiers, & decide if 1 or 2 instructions can issue
- VLIW: tradeoff instruction space for simple decoding
  - The long instruction word has room for many operations
  - By definition, all the operations the compiler puts in the long instruction word can execute in parallel
  - E.g., 2 integer operations, 2 FP ops, 2 Memory refs, 1 branch
    - » 16 to 24 bits per field => 7*16 or 112 bits to 7*24 or 168 bits wide
  - Need compiling technique that schedules across several branches

CS 252 Administrivia

- Reading Assignments for Lectures 3 to 6
  - Chapter 4, Appendix B
- Exercises for Lectures 3 to 6
  - 4.14 parts a - k, 4.25, B.3 parts a - g, B.15
  - also look at
  - http://http.cs.berkeley.edu/~patterson/252F96/hw1.html
  - Due Monday September 16 at 5PM homework box in 283 Soda (building is locked at 6:45 PM)
  - Done in pairs, but both need to understand whole assignment
- Video in 201 McLaughlin, starting day of lecture

  Mon 9-11AM, 2 - 5 PM; Tue 9 AM - 5 PM;
  Wed 9-11AM, 2 - 10 PM; Thu 9 AM - 6 PM;
  Fri 9 - 5PM, 6 - 10 PM;

Loop Unrolling in VLIW

Memory reference 1	*Memory reference 2*	*FP operation 1*	*FP op. 2*	*Int. op/ branch*	*Clock*
LD F0,0(R1)	LD F6,-8(R1)				1
LD F10,-16(R1)	LD F14,-24(R1)				2
LD F18,-32(R1)	LD F22,-40(R1)	ADDD F4,F0,F2	ADDD F8,F6,F2		3
LD F26,-48(R1)		ADDD F12,F10,F2	ADDD F16,F14,F2		4
		ADDD F20,F18,F2	ADDD F24,F22,F2		5
SD 0(R1),F4	SD -8(R1),F8	ADDD F28,F26,F2			6
SD -16(R1),F12	SD -24(R1),F16				7
SD -32(R1),F20	SD -40(R1),F24			SUBI R1,R1,#48	8
SD -0(R1),F28				BNEZ R1,LOOP	9

Unrolled 7 times to avoid delays

7 results in 9 clocks, or 1.3 clocks per iteration

Need more registers in VLIW

Trace Scheduling

- Parallelism across IF branches vs. LOOP branches
- Two steps:
  - *Trace Selection*
    - » Find likely sequence of basic blocks (*trace*) of (statically predicted) long sequence of straight-line code
  - *Trace Compaction*
    - » Squeeze trace into few VLIW instructions
    - » Need bookkeeping code in case prediction is wrong

Dynamic Scheduling in Superscalar

- Dependencies stop instruction issue
- Code compiler for scalar version will run poorly on SS
  - May want code to vary depending on how superscalar
- Simple approach: separate Tomasulo Control for separate reservation stations for Integer FU/Reg and for FP FU/Reg

Dynamic Scheduling in Superscalar

- How to do instruction issue with two instructions and keep in-order instruction issue for Tomasulo?
  - Issue 2X Clock Rate, so that issue remains in order
  - Only FP loads might cause dependency between integer and FP issue:
    - » Replace load reservation station with a load queue; operands must be read in the order they are fetched
    - » Load checks addresses in Store Queue to avoid RAW violation
    - » Store checks addresses in Load Queue to avoid WAR,WAW

Performance of Dynamic SS

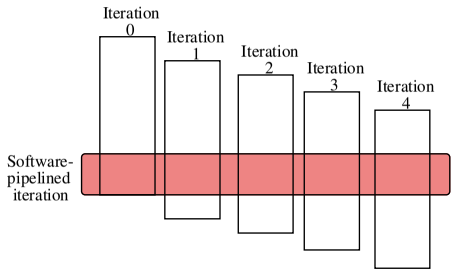
Iteration no.	*Instructions*	*Issues*	*Executes*	*Writes result*
		clock-cycle number		
1	LD F0,0(R1)	1	2	4
1	ADDD F4,F0,F2	1	5	8
1	SD 0(R1),F4	2	9	
1	SUBI R1,R1,#8	3	4	5
1	BNEZ R1,LOOP	4	5	
2	LD F0,0(R1)	5	6	8
2	ADDD F4,F0,F2	5	9	12
2	SD 0(R1),F4	6	13	
2	SUBI R1,R1,#8	7	8	9
2	BNEZ R1,LOOP	8	9	

≈ 4 clocks per iteration

Branches, Decrements still take 1 clock cycle

Software Pipelining

- Observation: if iterations from loops are independent, then can get ILP by taking instructions from different iterations
- Software pipelining: reorganizes loops so that each iteration is made from instructions chosen from different iterations of the original loop (≈ Tomasulo in SW)



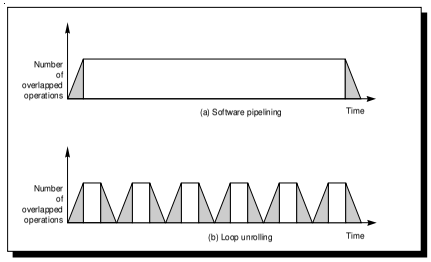
Software Pipelining Example

Before: Unrolled 3 times

```
1   LD    F0,0(R1)
2   ADDD  F4,F0,F2
3   SD    0(R1),F4
4   LD    F6,-8(R1)
5   ADDD  F8,F6,F2
6   SD    -8(R1),F8
7   LD    F10,-16(R1)
8   ADDD  F12,F10,F2
9   SD    -16(R1),F12
10  SUBI  R1,R1,#24
11  BNEZ  R1,LOOP
```

After: Software Pipelined

```
1   SD    0(R1),F4 ; Stores M[i]
2   ADDD  F4,F0,F2 ; Adds to M[i-1]
3   LD    F0,-16(R1);Loads M[i-2]
4   SUBI  R1,R1,#8
5   BNEZ  R1,LOOP
```



- Symbolic Loop Unrolling
  - Less code space
  - Fill & drain pipe only once vs. each iteration in loop unrolling

Limits to Multi-Issue Machines

- Inherent limitations of ILP
  - 1 branch in 5: How to keep a 5-way VLIW busy?
  - Latencies of units: many operations must be scheduled
  - Need about Pipeline Depth x No. Functional Units of independent operations to keep machines busy
- Difficulties in building HW
  - Duplicate FUs to get parallel execution
  - Increase ports to Register File
    - » VLIW example needs 7 read and 3 write for Int. Reg. & 5 read and 3 write for FP reg
  - Increase ports to memory
  - Decoding SS and impact on clock rate, pipeline depth

Limits to Multi-Issue Machines

- Limitations specific to either SS or VLIW implementation
  - Decode issue in SS
  - VLIW code size: unroll loops + wasted fields in VLIW
  - VLIW lock step => 1 hazard & all instructions stall
  - VLIW & binary compatibility is practical weakness

HW support for More ILP

- Avoid branch prediction by turning branches into conditionally executed instructions:

  if (x) then A = B op C else NOP
  - If false, then neither store result nor cause exception
  - Expanded ISA of Alpha, MIPS, PowerPC, SPARC have conditional move; PA-RISC can annul any following instr.
- Drawbacks to conditional instructions
  - Still takes a clock even if “annulled”
  - Stall if condition evaluated late
  - Complex conditions reduce effectiveness; condition becomes known late in pipeline

HW support for More ILP

- *Speculation*: allow an instruction to issue that is dependent on branch predicted to be taken *without* any consequences (including exceptions) if branch is not actually taken ("HW undo")
- Often try to combine with dynamic scheduling
- Tomasulo: separate *speculative* bypassing of results from real bypassing of results
  - When instruction no longer speculative, write results (*instruction commit*)
  - execute out-of-order but commit in order

HW support for More ILP

- Need HW buffer for results of uncommitted instructions: *reorder buffer*
  - Reorder buffer can be operand source
  - Once operand commits, result is found in register
  - 3 fields: instr. type, destination, value
  - Use reorder buffer number instead of reservation station
  - Instructions commit in order
  - As a result, its easy to undo speculated instructions on mispredicted branches or on

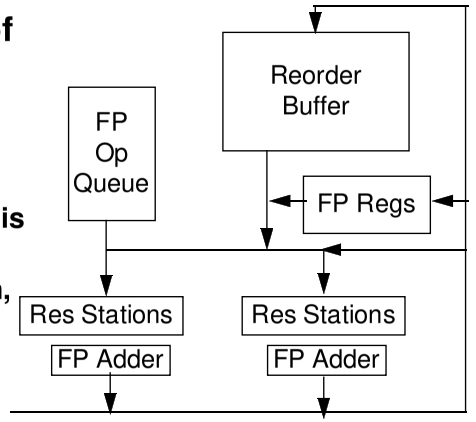


Figure 4.34, page 311

Four Steps of Speculative Tomasulo Algorithm

1. Issue—get instruction from FP Op Queue

   If reservation station and reorder buffer slot free, issue instr & send operands & reorder buffer no. for destination.

2. Execution—operate on operands (EX)

   When both operands ready then execute; if not ready, watch CDB for result; when both in reservation station, execute

3. Write result—finish execution (WB)

   Write on Common Data Bus to all awaiting FUs & reorder buffer; mark reservation station available.

4. Commit—update register with reorder result

   When instr. at head of reorder buffer & result present, update register with result (or store to memory) and

Limits to ILP

- Conflicting studies of amount of parallelism available in late 1980s and early 1990s. Different assumptions about:
  - Benchmarks (vectorized Fortran FP vs. integer C programs)
  - Hardware sophistication
  - Compiler sophistication

Limits to ILP

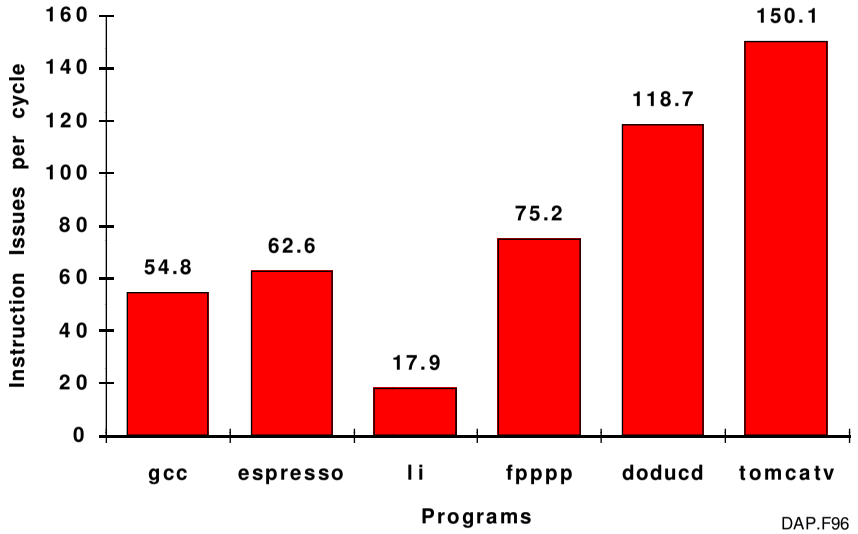
Initial HW Model here; MIPS compilers

1. *Register renaming*–infinite virtual registers and all WAW & WAR hazards are avoided
2. *Branch prediction*–perfect; no mispredictions
3. *Jump prediction*–all jumps perfectly predicted => machine with perfect speculation & an unbounded buffer of instructions available
4. *Memory-address alias analysis*–addresses are known & a store can be moved before a load provided addresses not equal

1 cycle latency for all instructions

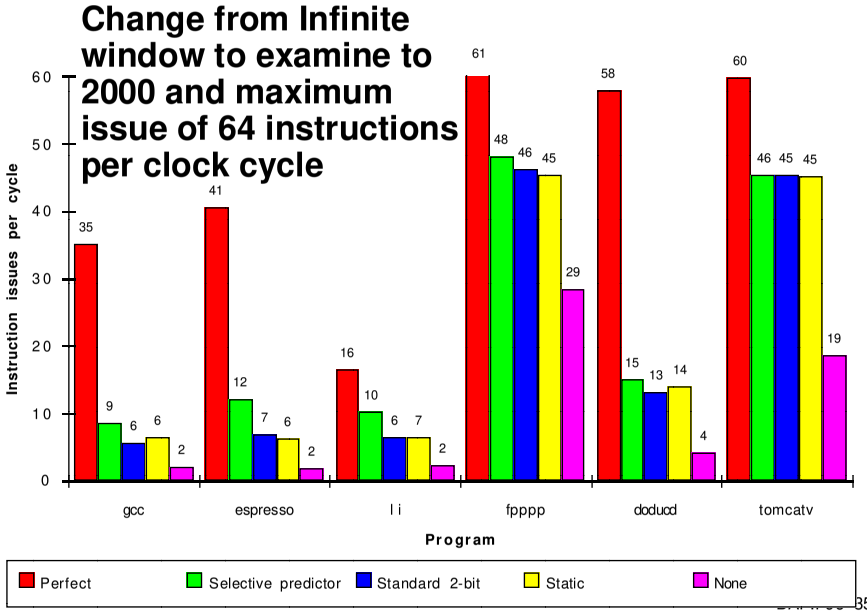
Upper Limit to ILP

(Figure 4.38, page 319)

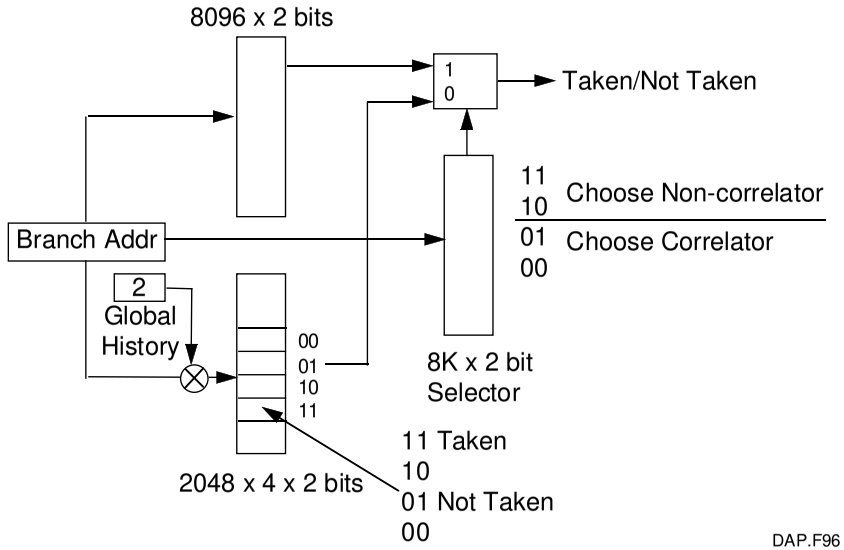


More Realistic HW: Branch Impact

Figure 4.40, Page 323

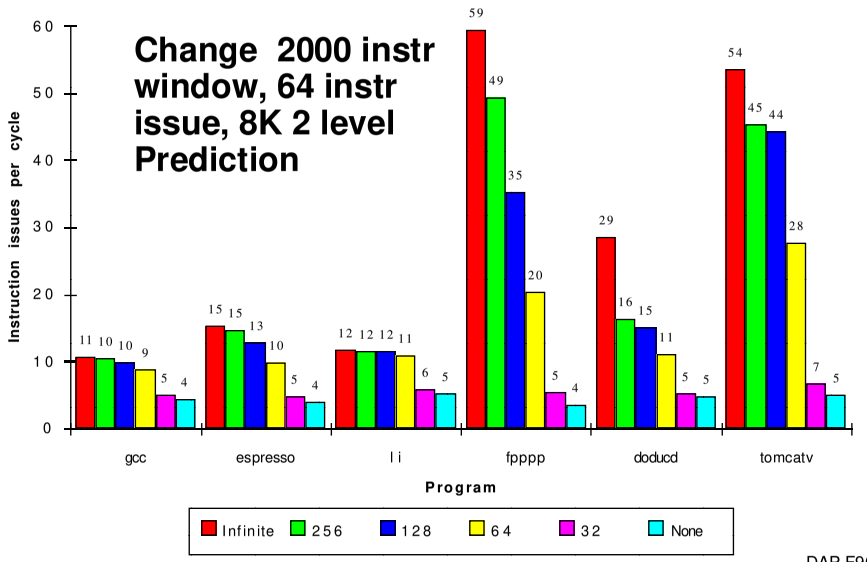


Selective History Predictor



More Realistic HW: Register Impact

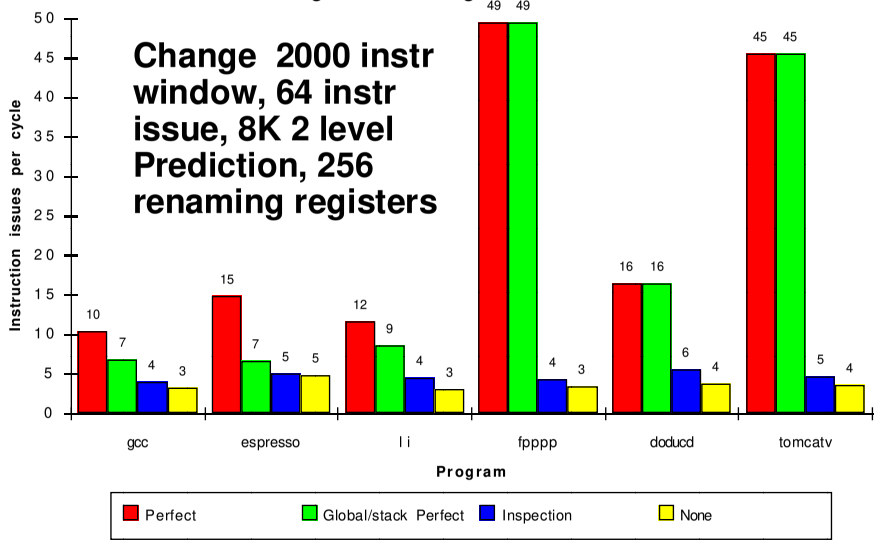
Figure 4.44, Page 328



Infinite 256 128 64 32 None

More Realistic HW: Alias Impact

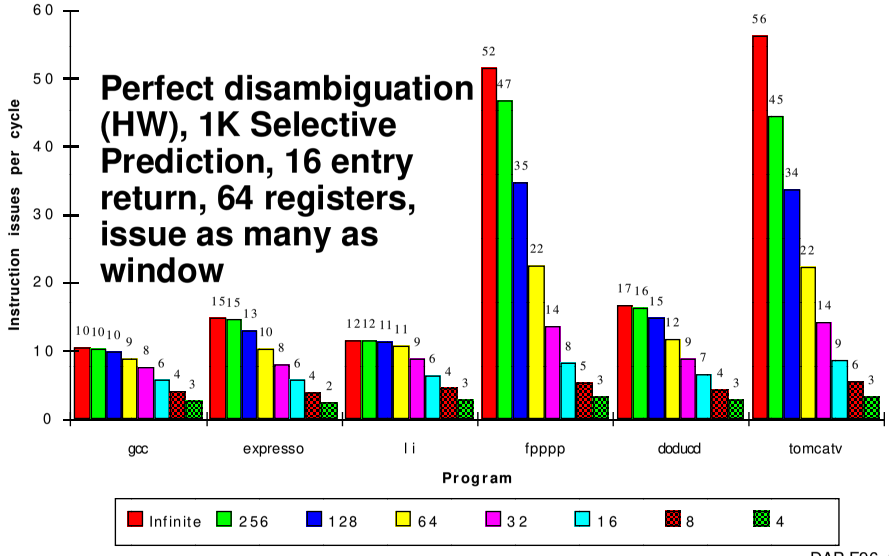
Figure 4.46, Page 330



Perfect Global/Stack perf; heap conflicts Inspec. Assem. None

Realistic HW for '9X: Window Impact

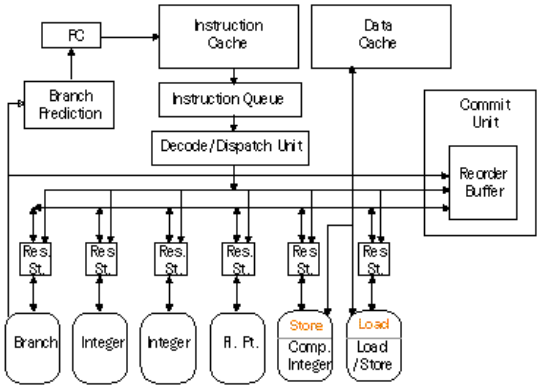
(Figure 4.48, Page 332)



Infinite 256 128 64 32 16 8 4

Dynamic Scheduling in PowerPC 604 and Pentium Pro

- Both In-order Issue, Out-of-order execution, In-order Commit



PPro central reservation station for any functional units with one bus shared by a branch and an integer unit

Dynamic Scheduling in PowerPC 604 and Pentium Pro

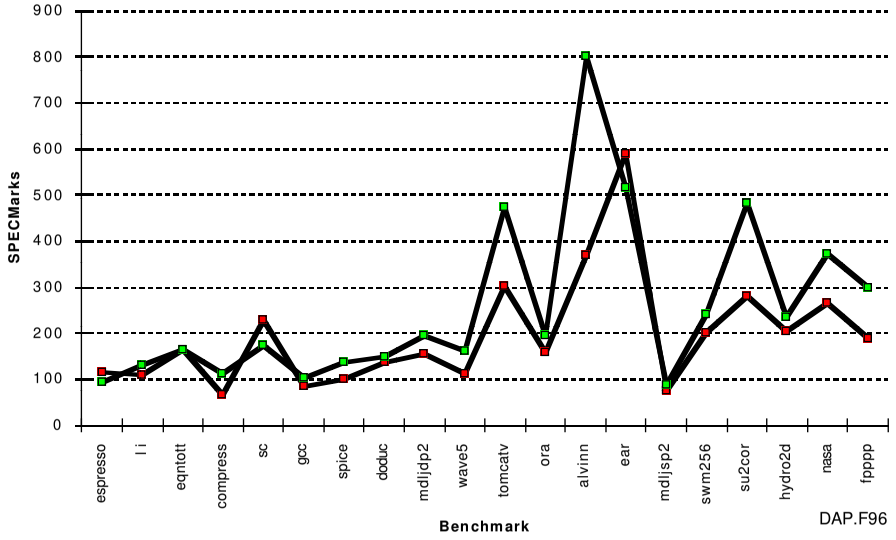
Parameter	PPC	PPro
Max. instructions issued/clock	4	3
Max. instr. complete exec./clock	6	5
Max. instr. commited/clock	6	3
Instructions in reorder buffer	16	40
Number of rename buffers	12 Int/8 FP	40
Number of reservations stations	12	20
No. integer functional units (FUs)	2	2
No. floating point FUs	1	1
No. branch FUs	1	1
No. complex integer FUs	1	0
No. memory FUs	1	1 load +1 store

Dynamic Scheduling in Pentium Pro

- PPro doesn't pipeline 80x86 instructions
- PPro decode unit translates the Intel instructions into 72-bit micro-operations (≈ MIPS)
- Sends micro-operations to reorder buffer & reservation stations
- Takes 1 clock cycle to determine length of 80x86 instructions + 2 more to create the micro-operations
- Most instructions translate to 1 to 4 micro-operations
- Complex 80x86 instructions are executed by a conventional microprogram (8K x 72 bits) that issues long sequences of micro-operations

Braniac vs. Speed Demon(1993)

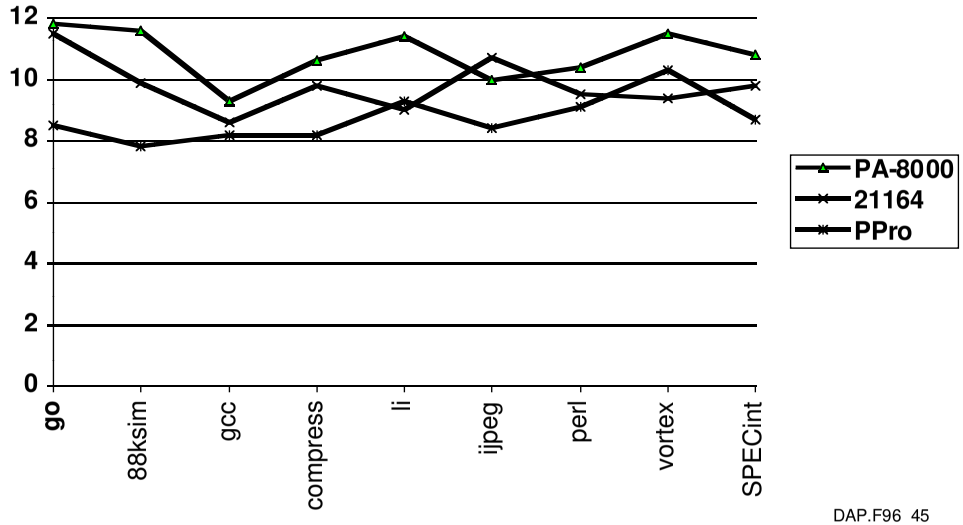
- 8-scalar IBM Power-2 @ 71.5 MHz (5 stage pipe) vs. 2-scalar Alpha @ 200 MHz (7 stage pipe)



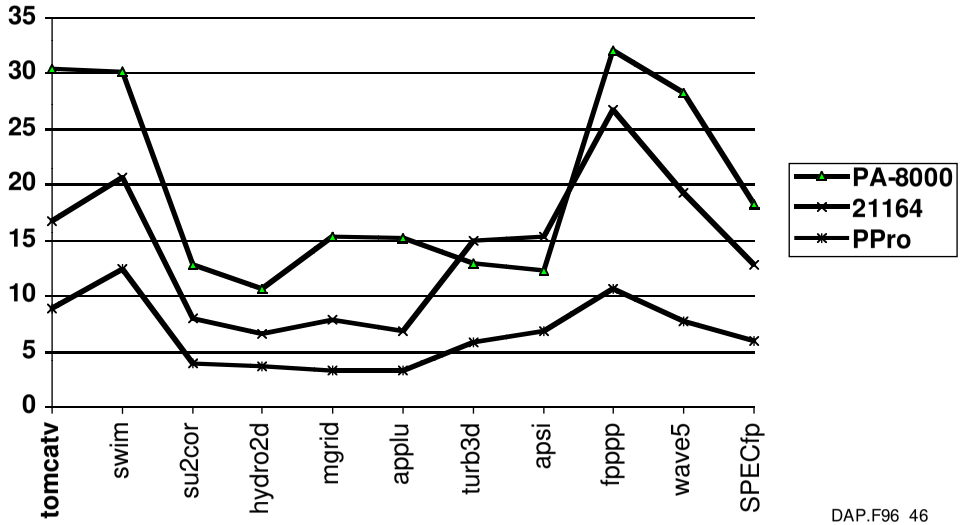
3 Recent Machines

	Alpha 21164	PPro	HP PA-8000
Year	1995	1995	1996
Clock	400 MHz	200 MHz	180 MHz
Cache	8K/8K/96K/2M	8K/8K/0.5M	0/0/2M
Issue rate	2int+2FP	3 instr (x86)	4 instr
Pipe stages	7-9	12-14	7-9
Out-of-Order	6 loads	40 instr (μop)	56 instr
Rename regs	none	40	56

SPECint95base Performance



SPECfp95base Performance



5 minute Class Break

- Lecture Format:
  - ≈ 1 minute: review last time & motivate this lecture
  - ≈ 20 minute lecture
  - ≈ 3 minutes: discuss class manangement
  - ≈ 25 minutes: lecture
  - 5 minutes: break
  - ≈25 minutes: lecture
  - ≈1 minute: summary of today's important topics

Instructon Level Parallelism

- High speed execution based on *instruction level parallelism* (ilp): potential of short instruction sequences to execute in parallel
- High-speed microprocessors exploit ILP by:
  1) pipelined execution: overlap instructions
  2) superscalar execution: issue and execute multiple instructions per clock cycle
  3) Out-of-order execution (commit in-order)
- Memory accesses for high-speed microprocessor?
  - For cache hits

Problems with conventional approach

- Limits to conventional exploitation of ILP:

  1) *pipelined clock rate*: at some point, each increase in clock rate has corresponding CPI increase

  2) *instruction fetch and decode*: at some point, its hard to fetch and decode more instructions per clock cycle

  3) *cache hit rate*: some long-running (scientific) programs have very large data sets accessed with poor locality

Vector Processors

- Vector processors have high-level operations that work on linear arrays of numbers: "vectors"

  e.g., A = BxC, where A, B, C are 64-element vectors of 64-bit floating point numbers
- Properties of vectors:
  - Each result independent of previous result
    => long pipeline, compiler ensures no dependencies
  - single vector instruction implies lots of work (≈ loop)
    => fewer instruction fetches
  - vector instructions access memory with known pattern
    => highly interleaved memory
    => amortize memory latency of over ≈ 64 elements
    => no caches required!
  - reduces branches and branch problems in pipelines

Styles of Vector Architectures

- *vector-register processors*: all vector operations between vector registers (except load and store)
  - Vector equivalent of load-store architectures
  - Includes all vector machines since late 1980s: Cray, Convex, Fujitsu, Hitachi, NEC
- *memory-memory vector processors*: all vector operations are memory to memory

Components of Vector Processor

- *Vector Register*: fixed length bank holding a single vector
  - has at least 2 read and 1 write ports
  - typically 8-16 vector registers, each holding 64-128 64-bit elements
- *Vector Functional Units (FUs)*: fully pipelined, start new operation every clock
  - typically 4 to 8: FP add, FP mult, FP reciprocal (1/X), integer add, logical, shift
- *Vector Load-Store Units (LSUs)*: fully pipelined unit to load or store a vector
- *Scalar registers*: single element for FP scalar or address
- Cross-bar to connect FUs , LSUs, registers

Example Vector Machines

Machine	Year	Clock	Regs	Elements	FUs	LSUs
Cray 1	1976	80 MHz	8	64	6	1
Cray XMP	1983	120 MHz	8	64	6	3
Cray YMP	1988	166 MHz	8	64	8	3
Cray C-90	1991	240 MHz	8	128	8	4
Cray T-90	1996	455 MHz	8	128	8	4
Conv. C-1	1984	10 MHz	8	128	4	1
Conv. C-4	1994	133 MHz	16	128	3	1
Fuj. VP200	1982	133 MHz	8-256	32-1024	3	2
Fuj. VP300	1996	100 MHz	8-256	32-1024	3	2
NEC SX/2	1984	160 MHz	8+8K	256+var	16	8
NEC SX/3	1995	400 MHz	8+8K	256+var	16	8

Vector Linpack Performance

Machine	Year	Clock	100x100	1kx1k	Peak(Procs)
Cray 1	1976	80 MHz	12	110	160(1)
Cray XMP	1983	120 MHz	121	218	940(4)
Cray YMP	1988	166 MHz	150	307	2,667(8)
Cray C-90	1991	240 MHz	387	902	15,238(16)
Cray T-90	1996	455 MHz	705	1603	57,600(32)
Conv. C-1	1984	10 MHz	3	--	20(1)
Conv. C-4	1994	135 MHz	160	2531	3240(4)
Fuj. VP200	1982	133 MHz	18	422	533(1)
NEC SX/2	1984	166 MHz	43	885	1300(1)
NEC SX/3	1995	400 MHz	368	2757	25,600(4)

Vector Instructions

Instr.	Operands	Operation	Comment
ADDV	V1,V2,V3	V1=V2+V3	vector + vector
ADDS	V1,F0,V2	V1=F0+V2	scalar + vector
MULV	V1,V2,V3	V1=V2xV3	vector x vector
MULS	V1,F0,V2	V1=F0xV2	scalar x vector
LV	V1,R1	V1=M[R1..R1+63]	load, stride=1
LVWS	V1,R1,R2	V1=M[R1..R1+63*R2]	load, stride=R2
LVI	V1,R1,V2	V1=M[R1+V2i,i=0..63]	"gather"
CeqV	VM,V1,V2	VMASKi = (V1i=V2i)?	comp. setmask
MOV	VLR,R1	Vec. Len. Reg. = R1	set vector length
MOV	VM,R1	Vec. Mask = R1	set vector mask

DAXPY (Y = a x X + Y)

Assuming vectors X, Y are length 64

Scalar vs. Vector

```
LD    F0,a       ;load scalar a
LV    V1,Rx      ;load vector X
MULS  V2,F0,V1   ;vector-scalar mult.
LV    V3,Ry      ;load vector Y
ADDV  V4,V2,V3   ;add
SV    Ry,V4      ;store the result
```

```
      LD     F0,a
      ADDI   R4,Rx,512   ;last address to load
loop: LD     F2,0(Rx)    ;load X(i)
      MULTD  F2,F0,F2    ;a*X(i)
      LD     F4,0(Ry)    ;load Y(i)
      ADDD   F4,F2,F4    ;a*X(i) + Y(i)
      SD     F4,0(Ry)    ;store into Y(i)
      ADDI   Rx,Rx,#8    ;increment index to X
      ADDI   Ry,Ry,#8    ;increment index to Y
      SUB    R20,R4,Rx   ;compute bound
      BNZ    R20,loop    ;check if done
```

578 (2+9*64) vs. 6 instructions:

64 operation vectors + no loop overhead

also fewer pipeline hazards

Vector Execution Time

- Time = f(vector length, data dependicies, hazards)
- *Initiation rate*: rate that FU consumes vector elements (usually 1, 2 on T-90)
- *Convoy*: set of vector instructions that can begin execution in same clock (no hazards)
- *Chime*: approx. time for a vector operation
- *m* convoys take *m* chimes; if each vector length is n, then they take approx. *m* x *n* clock cycles (ignores overhead)

```
1: LV    V1,Rx      ;load vector X
2: MULS  V2,F0,V1   ;vector-scalar mult.
   LV    V3,Ry      ;load vector Y
3: ADDV  V4,V2,V3   ;add
4: SV    Ry,V4      ;store the result
```

4 conveys
=> 4 x 64 ≈ 256 clocks

Start-up Time

- *Start-up time*: pipeline latency time (depth of FU pipeline)
- Operation Start-up penalty
- Vector load/store 12
- Vector multply 7
- Vector add 6
  - Assumes convoys don't overlap; vector length = n

Convoy	*Start*	*1st result*	*last result*
1. LV	0	12	11+n
2. MULV, LV	12+n	12+n+12	24+2n
3. ADDV	25+2n	25+2n+6	31+3n
4. SV	32+3n	32+3n+12	42+4n

Vector Load/Store Units & Memories

- Start-up overheads usually longer fo LSUs
- Memory system must sustain 1 word/clock cycle
- Many Vector Procs. use banks vs. simple interleaving:

  1) support multiple loads/stores per cycle
  => multiple banks & address banks independently

  2) support non-sequential accesses
- Note: No. memory banks > memory latency to avoid stalls

Summary

- Superscalar and VLIW
  - CPI < 1
  - Dynamic issue vs. Static issue
  - More instructions issue at same time, larger the penalty of hazards
- SW Pipelining
  - Symbolic Loop Unrolling to get most from pipeline with little code expansion, little overhead
- Vector
  - Alternate model accomodates long memory latency
  - Much easier for hardware: more powerful instructions, more predictable memory accesses, fewer branches, ...
  - What % of computation is vectorizable? For new apps?