Branch Prediction and Multiple-Issue Processors

Venkatesh Akella

EEC 270

Winter 2004

Based on Material provided by

Prof. Al Davis and Prof. David Culler

Branch Prediction

- Size of basic blocks limited to 4-7 instructions
- Delayed branches not a solution in multiple-issue processors
- Why? Hard to find independent instructions and remember the mess they create for precise exceptions
- To resolve a branch need two things (a) branch target address and (b) branch direction
- Prediction deals with (b) I.e. getting the direction
- Branch Penalty is governed by (a)
- Deeper pipeline – bad news as BP is higher

Static Branch Prediction

- Let the compiler figure out the branch direction for each branch instruction

Three strategies:

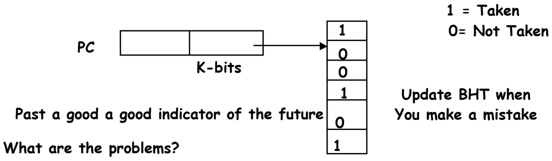
a) Always Predict Taken - Misprediction is 34%

b) Forward Not Taken; Backward Taken --- Misprediction is 10% - 40%

c) Profile-driven – using realistic benchmarks and real data and for each branch determine the direction – Hennessey & McFarling and Larus and Ball

Dynamic Branch Prediction

- Run Time
- Hardware assisted
- Intuition – branches direction is not random, they are BIMODAL i.e. either strongly taken or not taken
- One-bit Branch Prediction Buffer or Branch History Table (BHT) – Smith 1981



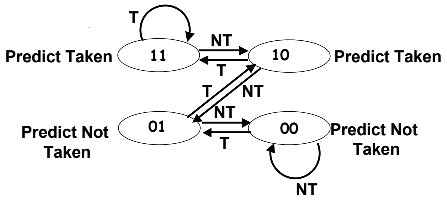
What are the problems?

a) Aliasing due to limited size of the BHT (tag can be stored to avoid this problem)

b) 1-bit history may not be sufficient? Eg: consider a loop that iterates 10 times – You will mispredict 2/10 so accuracy is 80%

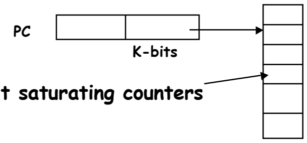
Dynamic Branch Prediction (Jim Smith, 1981)

- Better Solution: 2-bit scheme where change prediction only if get misprediction *twice:*



- Adds *hysteresis* to decision making process

2-bit Counters



PC

K-bits

2^k 2-bit saturating counters

- Upto 93.5% accuracy
- If K is sufficiently large, each branch maps to a unique counter
- Can store tags if you want to avoid aliasing
- How about m-bit counters?
- Doesn't benefit much

How do you improve further?

- Can we capture the actual history of the specific branch and use that to make our prediction? – LOCAL HISTORY
- Can we capture the sequential correlation between branches – GLOBAL HISTORY
- do both?
- Make multiple predictions and choose the right prediction based on the context of the particular branch – TOURNAMENT predictors

Using Local History

Consider the simple for loop

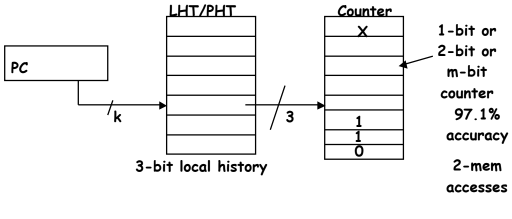
FOR (I=1; I<5; I++) { something …}

If the branch is at the end of the loop body, it has following pattern – $(1110)^N$

The sequence of the branch history is

1110111011101110 ……

Basically, if we know what the branch did the last three times, we can predict EXACTLY what it will do next.



97.1% accuracy

2-mem accesses

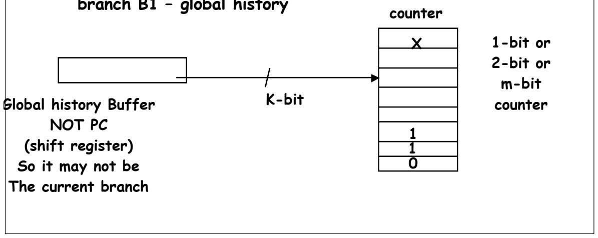
Global History (Correlated Branch Prediction)

If (x < 1) … B1

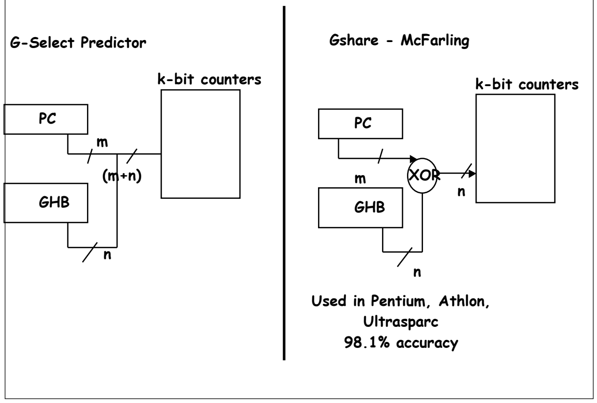
if (x > 1) … B2

Observation – if B1 is taken then B2 is not taken

This is a characteristic of structured programming (nested procedure calls and nested conditionals). So, whether B2 is taken or not is related to the previous branch B1 – global history

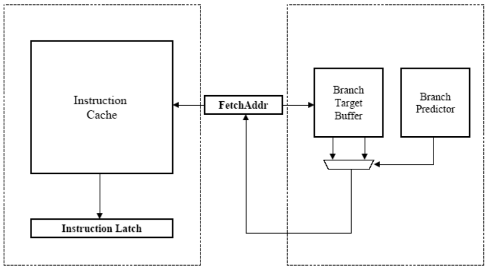


Hybrid Predictor



Used in Pentium, Athlon, Ultrasparc
98.1% accuracy

Processor Front-End

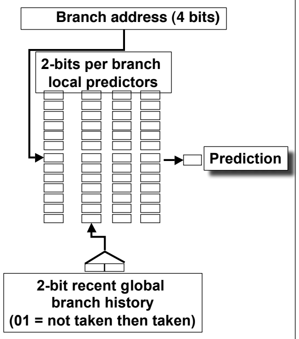


Correlating Branches

Idea: taken/not taken of recently executed branches is related to behavior of next branch (as well as the history of that branch behavior)

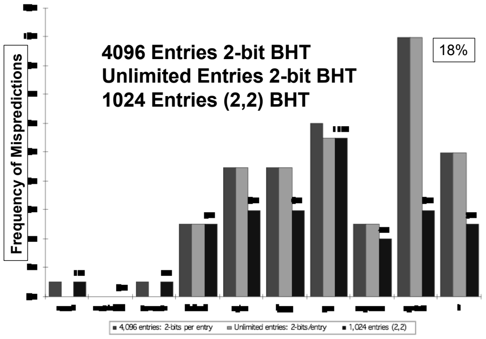
- Then behavior of recent branches selects between, say, 4 predictions of next branch, updating just that prediction

- (2,2) predictor: 2-bit global, 2-bit local


Branch address (4 bits)
2-bits per branch local predictors
Prediction
2-bit recent global branch history
(01 = not taken then taken)

Accuracy of Different Schemes

(Figure 3.15, p. 206)



Re-evaluating Correlation

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

program	branch %	static	# = 90%
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?

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%
- For SPEC92,
  4096 about as good as infinite table

Tournament Predictors

- Motivation for hybrid branch predictors is 2-bit predictor failed on important branches; by adding global information, performance improved
- Tournament predictors: use 2 predictors, 1 based on global information and 1 based on local information, and combine with a selector
- Hopes to select right predictor for right branch (or right context of branch)

Tournament Predictor in Alpha 21264

- 4K 2-bit counters to choose from among a global predictor and a local predictor
- Global predictor also has 4K entries and is indexed by the history of the last 12 branches; each entry in the global predictor is a standard 2-bit predictor
  - 12-bit pattern: ith bit 0 => ith prior branch not taken;
    ith bit 1 => ith prior branch taken;
- Local predictor consists of a 2-level predictor:
  - Top level a local history table consisting of 1024 10-bit entries; each 10-bit entry corresponds to the most recent 10 branch outcomes for the entry. 10-bit history allows patterns 10 branches to be discovered and predicted.
  - Next level Selected entry from the local history table is used to index a table of 1K entries consisting a 3-bit saturating counters, which provide the local prediction
- Total size: 4K*2 + 4K*2 + 1K*10 + 1K*3 = 29K bits!

  (~180,000 transistors)

% of predictions from local predictor in Tournament Prediction Scheme

Accuracy of Branch Prediction

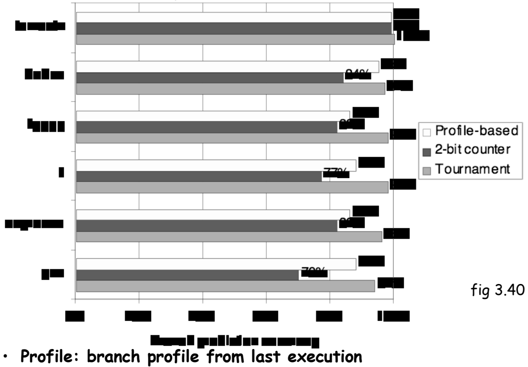
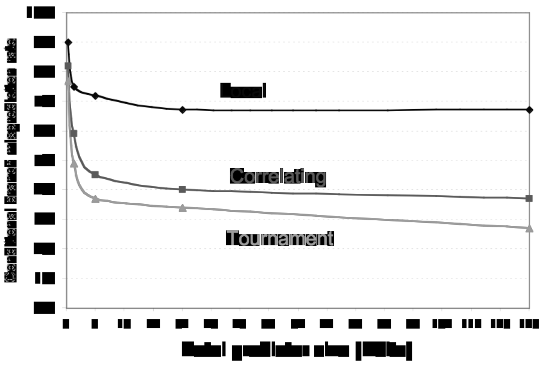


fig 3.40

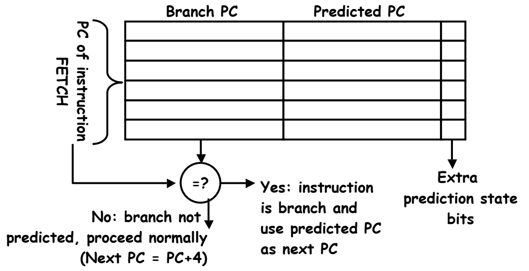
- Profile: branch profile from last execution
  (static in that in encoded in instruction, but profile)

Accuracy v. Size (SPEC89)



Need Address at Same Time as Prediction

- Branch Target Buffer (BTB): Indexed using the branch instruction Address to get prediction AND branch address (if taken)
- Accessed in IF stage



Branch PC

Predicted PC

PC of instruction FETCH

=?

Yes: instruction is branch and use predicted PC as next PC

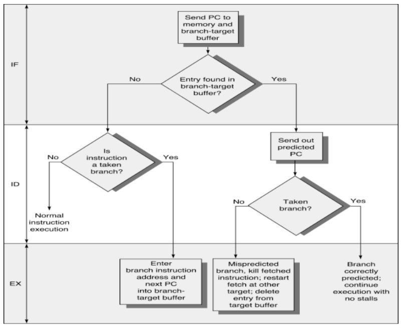
No: branch not predicted, proceed normally (Next PC = PC+4)

Extra prediction state bits

Branch Target Buffer

- Exists in the IF Stage
- This is a cache (Need the tags as well)
- Need to look-up whole PC (last bits won't do) because this stage we do not know the opcode yet
- Need to keep only predict taken branches only, others follow normal fetch sequence.

How Branch Target Buffer is Used?

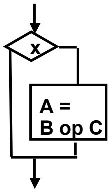

IF
ID
EX
Send PC to memory and branch-target buffer
Entry found in branch-target buffer?
No
Yes
Is instruction a taken branch?
No
Yes
Normal instruction execution
Send out predicted PC
Taken branch?
No
Yes
Enter branch instruction address and next PC into branch-target buffer
Mispredicted branch, kill fetched instruction; restart fetch at other target; delete entry from target buffer
Branch correctly predicted; continue execution with no stalls

Predicated Execution

- 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.
  - This transformation is called "if-conversion"



- Drawbacks to conditional instructions
  - Still takes a clock even if "annulled"
  - Stall if condition evaluated late
  - Complex conditions or condition becomes known late in pipeline => reduces effectiveness

Branch Folding

- Branch Folding – Instead of storing Next PC or BTA, how about storing the target instruction itself or multiple instructions if it is a multi-issue processor

Eg: L2 : b L

L : add R1, R2, R3

L2:	Add r1,r2, r3

At address corresponding to L2, you store the add instruction instead of the unconditional branch instruction b L

ZERO cycle BRANCH

(eliminated one instruction all together)

Advanced Approaches

- Trace Caches – aggressive prefetching
- Return Address Caches – jr $Ra – when $Ra is return address of a procedure.

85% of indirect jumps are due to procedure returns.

BTB does not work very well because procedure is called from many different places

So, you a separate stack cache to push $Ra and pop them off

Special Case Return Addresses

- Register Indirect branch hard to predict address
- SPEC89 85% such branches for procedure return
- Since stack discipline for procedures, save return address in small buffer that acts like a stack: 8 to 16 entries has small miss rate

Pitfall: Sometimes bigger and dumber is better

- 21264 uses tournament predictor (29 Kbits)
- Earlier 21164 uses a simple 2-bit predictor with 2K entries (or a total of 4 Kbits)
- SPEC95 benchmarks, 22264 outperforms
  - 21264 avg. 11.5 mispredictions per 1000 instructions
  - 21164 avg. 16.5 mispredictions per 1000 instructions
- Reversed for transaction processing (TP) !
  - 21264 avg. 17 mispredictions per 1000 instructions
  - 21164 avg. 15 mispredictions per 1000 instructions
- TP code much larger & 21164 hold 2X branch predictions based on local behavior (2K vs. 1K local predictor in the 21264)

Dynamic Branch Prediction Summary

- Prediction becoming important part of scalar execution
- Branch History Table: 2 bits for loop accuracy
- Correlation: Recently executed branches correlated with next branch.
  - Either different branches
  - Or different executions of same branches
- Tournament Predictor: more resources to competitive solutions and pick between them
- Branch Target Buffer: include branch address & prediction
- Predicated Execution can reduce number of branches, number of mispredicted branches
- Return address stack for prediction of indirect jump

Multiple Issue

- Goal how to reduce CPI below 1.0
- Consider two consecutive blocks of instructions

Gj – {i1, i2, i3, i4} and Gi = {i5,i6, i7, i8}

Gj is already in execution

1. Fetch Gi
2. Check for all structural hazards that instructions in Gj may introduce
3. Check for data hazards between Gi and between instructions in Gi and Gj
4. Read operands and execute

Flavors of Multiple Issue Processors

- Vector = execute a loop in parallel – directly on array data structures
- Superscalar
  - Static = in-order-execution (if I5 has a problem, HALT)
    - Eg: SUN ULTRA SPARC II/III
  - Dynamic = out-of-order execution (let I6 if I5 has a resource conflict
    - » No Speculation – If i5 is a branch do not allow I6 till branch is resolved
      - IBM Power 2
    - » With Speculation- Allow I6 but be prepared to rollback (Pentium 3, Pentium 4, Alpha 21264, MIPS R10K)
- VLIW
  - Compiler determines what to execute in parallel (Trimedia)
  - EPIC (basis for Itanium)

Multiple Issue Headaches

- Increased I-Cache Fetch BW
- Alignment problems may not allow 4 instructions to be fetched
- Need to check for more hazards
- Branches – 25% of instructions are branches, so you need to resolve a branch every cycle!
- Increased ports on register file and memory

So, how do we proceed

1. Pipeline the Issue unit into 2 stages
2. Restricted Issue eg: one int and one FP

Getting CPI < 1: Issuing Multiple Instructions/Cycle

- **Superscalar MIPS: 2 instructions, 1 FP & 1 anything**
  - Fetch 64-bits/clock cycle; Integer 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

Multiple Issue Issues

- **issue packet: group of instructions from fetch unit that could potentially issue in 1 clock**
  - If instruction causes structural hazard or a data hazard either due to earlier instruction in execution or to earlier instruction in issue packet, then instruction does not issue
  - 0 to N instruction issues per clock cycle, for N-issue
- **Performing issue checks in 1 cycle could limit clock cycle time: $O(n^2\text{-}n)$ comparisons**
  - => issue stage usually split and pipelined
  - 1st stage decides how many instructions from within this packet can issue, 2nd stage examines hazards among selected instructions and those already been issued
  - => higher branch penalties => prediction accuracy important

Multiple Issue Challenges

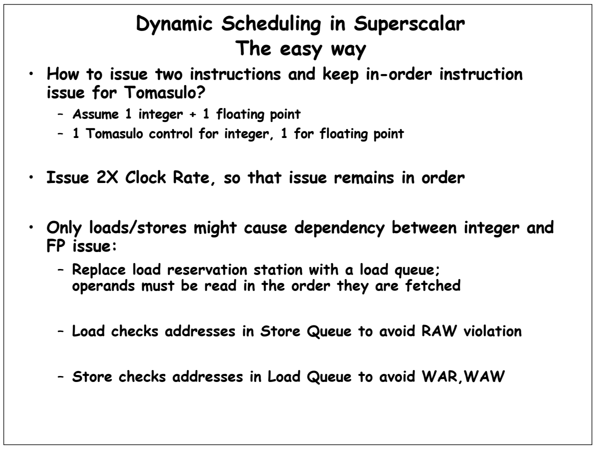
- **While Integer/FP split is simple for the HW, get CPI of 0.5 only for programs with:**
  - Exactly 50% FP operations AND No hazards
- **If more instructions issue at same time, greater difficulty of decode and issue:**
  - Even 2-scalar => examine 2 opcodes, 6 register specifiers, & decide if 1 or 2 instructions can issue; (N-issue ~$O(N^2\text{-}N)$ comparisons)
  - Register file: need 2x reads and 1x writes/cycle
  -

Multiple Issue Headaches

- Rename logic: must be able to rename same register multiple times in one cycle! For instance, consider 4-way issue:

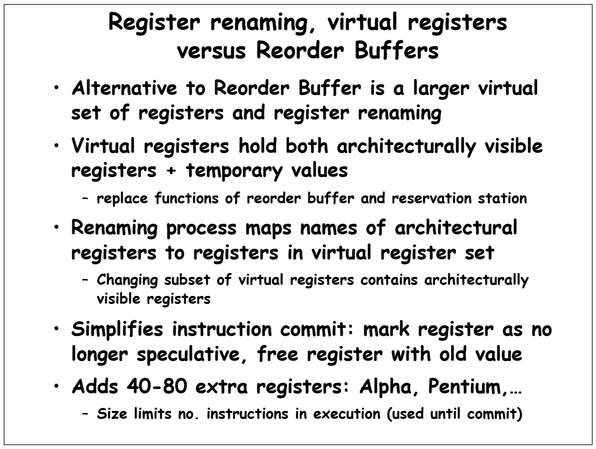
```
add r1, r2, r3             add p11, p4, p7
sub r4, r1, r2      ⇒      sub p22, p11, p4
lw  r1, 4(r4)              lw  p23, 4(p22)
add r5, r1, r2             add p12, p23, p4
```

  Imagine doing this transformation in a single cycle!
- Result buses: Need to complete multiple instructions/cycle
  - So, need multiple buses with associated matching logic at every reservation station.
  - Or, need multiple forwarding paths



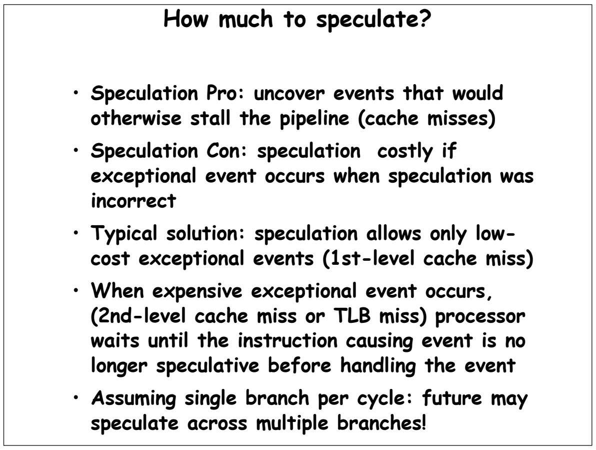
Dynamic Scheduling in Superscalar The easy way

- How to issue two instructions and keep in-order instruction issue for Tomasulo?
  - Assume 1 integer + 1 floating point
  - 1 Tomasulo control for integer, 1 for floating point
- Issue 2X Clock Rate, so that issue remains in order
- Only loads/stores 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



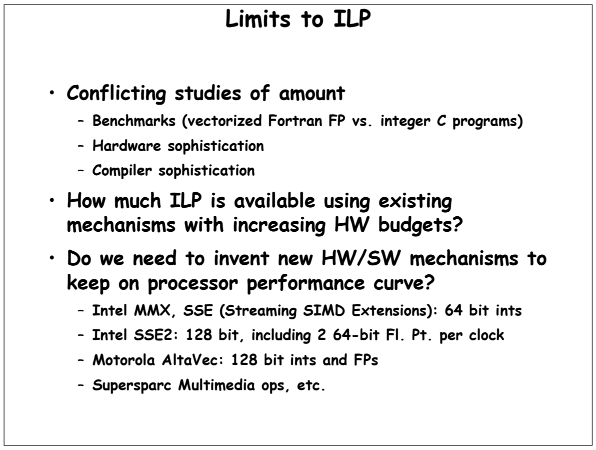
Register renaming, virtual registers versus Reorder Buffers

- Alternative to Reorder Buffer is a larger virtual set of registers and register renaming
- Virtual registers hold both architecturally visible registers + temporary values
  - replace functions of reorder buffer and reservation station
- Renaming process maps names of architectural registers to registers in virtual register set
  - Changing subset of virtual registers contains architecturally visible registers
- Simplifies instruction commit: mark register as no longer speculative, free register with old value
- Adds 40-80 extra registers: Alpha, Pentium,…
  - Size limits no. instructions in execution (used until commit)



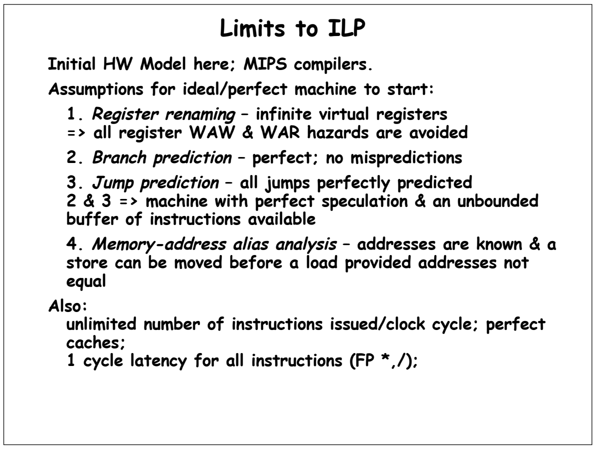
How much to speculate?

- Speculation Pro: uncover events that would otherwise stall the pipeline (cache misses)
- Speculation Con: speculation costly if exceptional event occurs when speculation was incorrect
- Typical solution: speculation allows only low-cost exceptional events (1st-level cache miss)
- When expensive exceptional event occurs, (2nd-level cache miss or TLB miss) processor waits until the instruction causing event is no longer speculative before handling the event
- Assuming single branch per cycle: future may speculate across multiple branches!



Limits to ILP

- Conflicting studies of amount
  - Benchmarks (vectorized Fortran FP vs. integer C programs)
  - Hardware sophistication
  - Compiler sophistication
- How much ILP is available using existing mechanisms with increasing HW budgets?
- Do we need to invent new HW/SW mechanisms to keep on processor performance curve?
  - Intel MMX, SSE (Streaming SIMD Extensions): 64 bit ints
  - Intel SSE2: 128 bit, including 2 64-bit Fl. Pt. per clock
  - Motorola AltaVec: 128 bit ints and FPs
  - Supersparc Multimedia ops, etc.



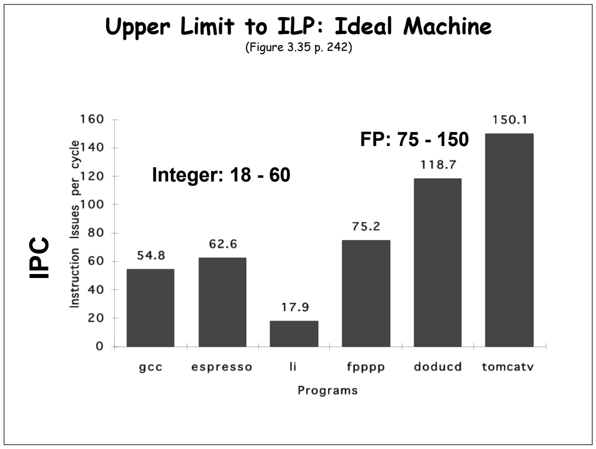
Limits to ILP

Initial HW Model here; MIPS compilers.

Assumptions for ideal/perfect machine to start:

1. *Register renaming* – infinite virtual registers
=> all register WAW & WAR hazards are avoided
2. *Branch prediction* – perfect; no mispredictions
3. *Jump prediction* – all jumps perfectly predicted
2 & 3 => 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

Also:
unlimited number of instructions issued/clock cycle; perfect caches;
1 cycle latency for all instructions (FP *,/);

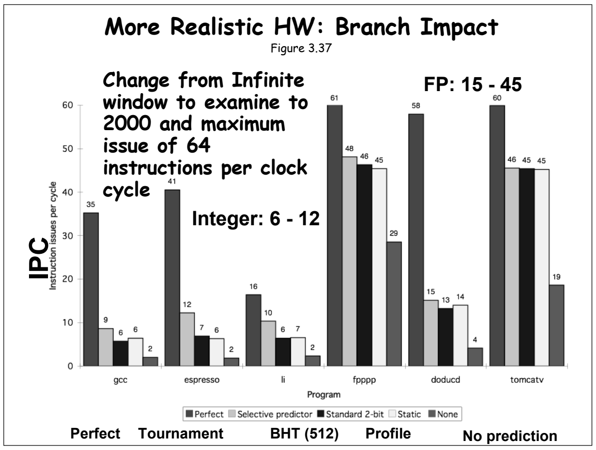


Upper Limit to ILP: Ideal Machine

(Figure 3.35 p. 242)

FP: 75 - 150

Integer: 18 - 60



More Realistic HW: Branch Impact

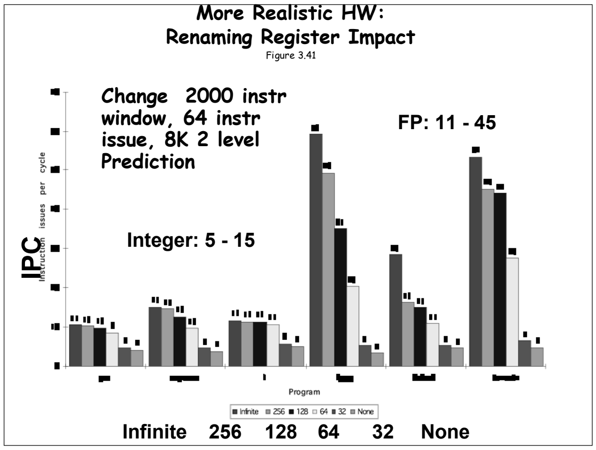
Figure 3.37

Change from Infinite window to examine to 2000 and maximum issue of 64 instructions per clock cycle

FP: 15 - 45

Integer: 6 - 12

Perfect Tournament BHT (512) Profile No prediction



More Realistic HW: Renaming Register Impact

Figure 3.41

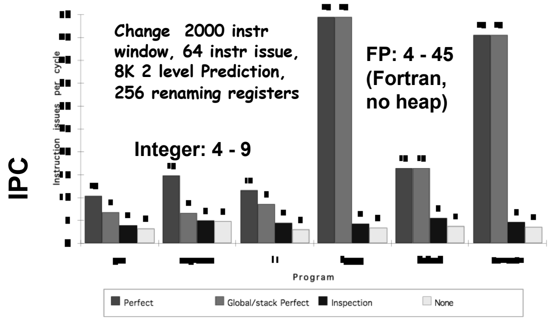
Change 2000 instr window, 64 instr issue, 8K 2 level Prediction

FP: 11 - 45

Integer: 5 - 15

Infinite 256 128 64 32 None

More Realistic HW: Memory Address Alias Impact



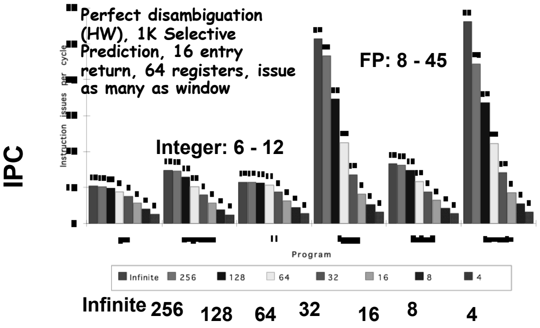
Change 2000 instr window, 64 instr issue, 8K 2 level Prediction, 256 renaming registers

FP: 4 - 45 (Fortran, no heap)

Integer: 4 - 9

Realistic HW: Window Impact

(Figure 3.46)



Perfect disambiguation (HW), 1K Selective Prediction, 16 entry return, 64 registers, issue as many as window

FP: 8 - 45

Integer: 6 - 12

How to Exceed ILP Limits of this study?

- WAR and WAW hazards through memory
  - eliminated WAW and WAR hazards on registers through renaming, but not in memory usage
- Unnecessary dependences (compiler not unrolling loops so iteration variable dependence)
- Overcoming the data flow limit: value prediction, predicting values and speculating on prediction
  - Address value prediction and speculation predicts addresses and speculates by reordering loads and stores; could provide better aliasing analysis, only need predict if addresses =
- Use multiple threads of control

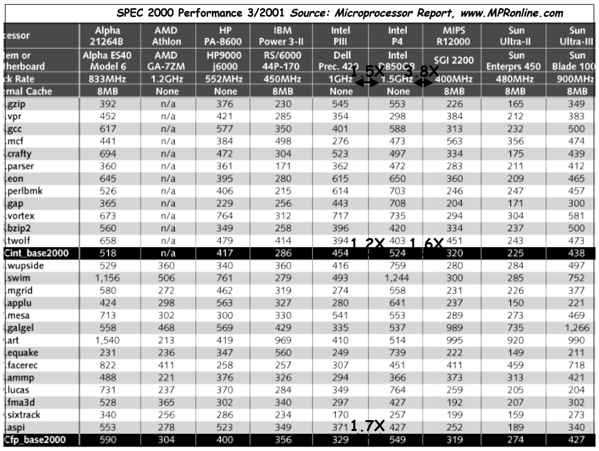
Workstation Microprocessors 3/2001

Processor	Alpha 21264B	AMD Athlon	HP PA-8600	IBM Power3-II	Intel Pentium III	Intel Pentium 4	MIPS R12000	Sun Ultra-II	S Ul
Clock Rate	833MHz	1.2GHz	552MHz	450MHz	1.0GHz	1.5GHz	400MHz	480MHz	90
Cache (I/D/L2)	64K/64K	64K/64K/256K	512K/1M	32K/64K	16K/16K/256K	12K/8K/256K	32K/32K	16K/16K	32
Issue Rate	4 issue	3 x86 instr	4 issue	4 issue	3 x86 instr	3 x ROPs	4 issue	4 issue	4 i
Pipeline Stages	7/9 stages	9/11 stages	7/9 stages	7/8 stages	12/14 stages	22/24 stages	6 stages	6/9 stages	14/1
Out of Order	80 instr	72ROPs	56 instr	32 instr	40 ROPs	126 ROPs	48 instr	None	N
Rename regs	48/41	36/36	56 total	16 int/24 fp	40 total	128 total	32/32	None	N
BHT Entries	4K × 9-bit	4K × 2-bit	2K × 2-bit	2K × 2-bit	>= 512	4K × 2-bit	2K × 2-bit	512 × 2-bit	16K
TLB Entries	128/128	280/288	120 unified	128/128	32I / 64D	128I/65D	64 unified	64I/64D	128I
Memory B/W	2.66GB/s	2.1GB/s	1.54GB/s	1.6GB/s	1.06GB/s	3.2GB/s	539 MB/s	1.9GB/s	4.8
Package	CPGA-588	PGA-462	LGA-544	SCC-1088	PGA-370	PGA-423	CPGA-527	CLGA-787	1368
IC Process	0.18µ 6M	0.18µ 6M	0.25µ 2M	0.22µ 6m	0.18µ 6M	0.18µ 6M	0.25µ 4M	0.29µ 6M	0.1
Die Size	115mm^2	117mm^2	477mm^2	163mm^2	106mm^2	217mm^2	204mm^2	126 mm^2	21
Transistors	15.4 million	37 million	130 million	23 million	24 million	42 million	7.2 million	3.8 million	29
Est mfg cost*	$160	$62	$330	$110	$39	$110	$125	$70	$
Power(Max)	75W*	76W	60W*	36W*	30W	55W(TDP)	25W*	20W*	6
Availability	1Q01	4Q00	3Q00	4Q00	2Q00	4Q00	2Q00	3Q0	4

- Max issue: 4 instructions (many CPUs)
  Max rename registers: 128 (Pentium 4)
  Max BHT: 4K x 9 (Alpha 21264B), 16Kx2 (Ultra III)
  Max Window Size (OOO): 126 intructions (Pent. 4)
  Max Pipeline: 22/24 stages (Pentium 4)

Source: Microprocessor Report, www.MPRonline.com

SPEC 2000 Performance 3/2001 ***Source: Microprocessor Report, www.MPRonline.com***



cessor	Alpha 21264B	AMD Athlon	HP PA-8600	IBM Power 3-II	Intel PIII	Intel P4	MIPS R12000	Sun Ultra-II	Sun Ultra-III
tem or therboard	Alpha ES40 Model 6	AMD GA-7ZM	HP9000 j6000	RS/6000 44P-170	Dell Prec. 420	Intel D850GB	SGI 2200	Sun Enterprs 450	Sun Blade 100
k Rate	833MHz	1.2GHz	552MHz	450MHz	1GHz	1.5GHz	400MHz	480MHz	900MHz
rnal Cache	8MB	None	None	8MB	None	None	8MB	8MB	8MB
.gzip	392	n/a	376	230	545	553	226	165	349
.vpr	452	n/a	421	285	354	298	384	212	383
.gcc	617	n/a	577	350	401	588	313	232	500
.mcf	441	n/a	384	498	276	473	563	356	474
.crafty	694	n/a	472	304	523	497	334	175	439
.parser	360	n/a	361	171	362	472	283	211	412
.eon	645	n/a	395	280	615	650	360	209	465
.perlbmk	526	n/a	406	215	614	703	246	247	457
.gap	365	n/a	229	256	443	708	204	171	300
.vortex	673	n/a	764	312	717	735	294	304	581
.bzip2	560	n/a	349	258	396	420	334	237	500
.twolf	658	n/a	479	414	394	403	451	243	473
Cint_base2000	518	n/a	417	286	454	524	320	225	438
.wupside	529	360	340	360	416	759	280	284	497
.swim	1,156	506	761	279	493	1,244	300	285	752
.mgrid	580	272	462	319	274	558	231	226	377
.applu	424	298	563	327	280	641	237	150	221
.mesa	713	302	300	330	541	553	289	273	469
.galgel	558	468	569	429	335	537	989	735	1,266
.art	1,540	213	419	969	410	514	995	920	990
.equake	231	236	347	560	249	739	222	149	211
.facerec	822	411	258	257	307	451	411	459	718
.ammp	488	221	376	326	294	366	373	313	421
.lucas	731	237	370	284	349	764	259	205	204
.fma3d	528	365	302	340	297	427	192	207	302
.sixtrack	340	256	286	234	170	257	199	159	273
.aspi	553	278	523	349	371	427	252	189	340
Cfp_base2000	590	304	400	356	329	549	319	274	427

Conclusion

- **1985-2000: 1000X performance**
  - **Moore's Law transistors/chip => Moore's Law for Performance/MPU**
- **Hennessy: industry been following a roadmap of ideas known in 1985 to exploit Instruction Level Parallelism and (real) Moore's Law to get 1.55X/year**
  - **Caches, Pipelining, Superscalar, Branch Prediction, Out-of-order execution,**
- **ILP limits: To make performance progress in future need to have explicit parallelism from programmer vs. implicit parallelism of ILP exploited by compiler, HW?**
  - **Otherwise drop to old rate of 1.3X per year?**
  - **Less than 1.3X because of processor-memory performance gap?**
- **Impact on you: if you care about performance, better think about explicitly parallel algorithms vs. rely on ILP?**