Lecture 8: Vector Processing, Branch Prediction, Dependence Speculation

Prof. John Kubiatowicz
Computer Science 252
Fall 1998

Review

- Precise exceptions/Speculation: Out-of-order execution, In-order commit (reorder buffer)
- Explicit Renaming: more physical registers than needed by ISA. Uses a translation table
- Memory Disambiguation: Detecting RAW hazards that occur through the memory interface.
  - Simplistic approach: wait until addresses for all previous stores are ready before starting load.
- Superscalar and VLIW: CPI < 1 (IPC > 1)
  - Dynamic issue vs. Static issue
  - More instructions issue at same time => larger hazard penalty
  - Limitation is often number of instructions that you can successfully fetch and decode per cycle ⇒ "Flynn barrier"
- SW Pipelining
  - Symbolic Loop Unrolling to get most from pipeline with little code expansion, little overhead

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 mechanims with increasing HW budgets?
- Do we need to invent new HW/SW mechanisms to keep on processor performance curve?
  - Intel MMX
  - Motorola AltaVec
  - Supersparc Multimedia ops, etc.

Limits to ILP

Initial HW Model here; MIPS compilers.

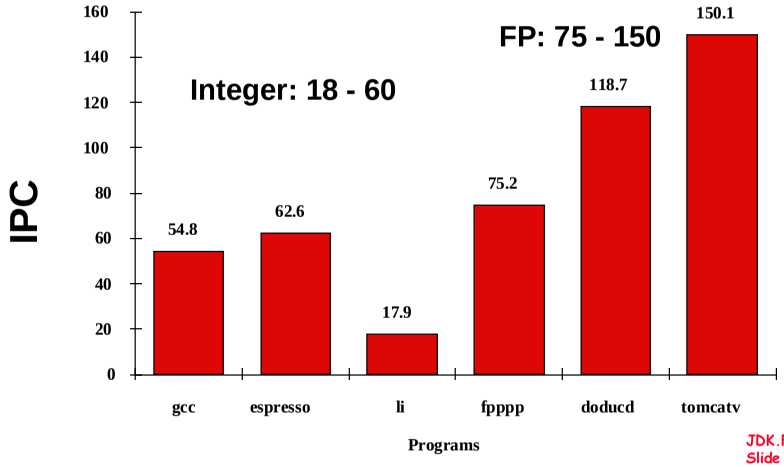
Assumptions for ideal/perfect machine to start:

1. *Register renaming*–infinite virtual registers 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; unlimited number of instructions issued per clock cycle

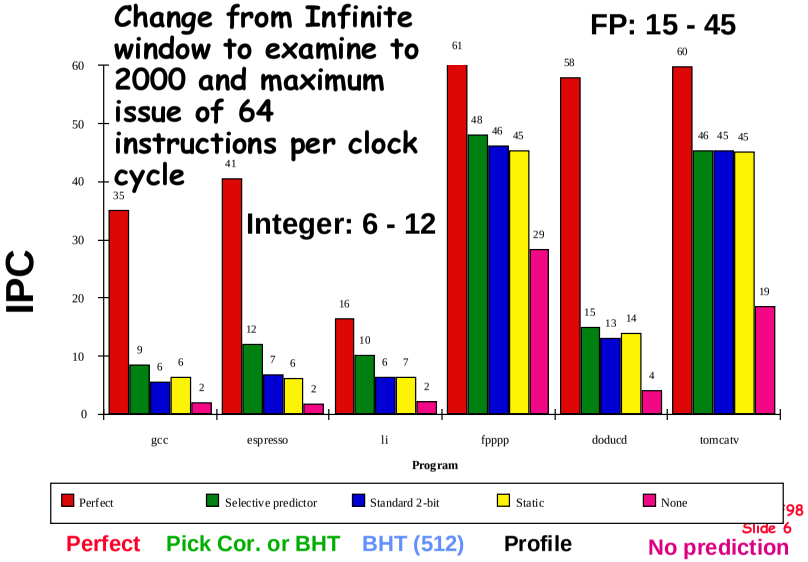
Upper Limit to ILP: Ideal Machine

(Figure 4.38, page 319)



More Realistic HW: Branch Impact

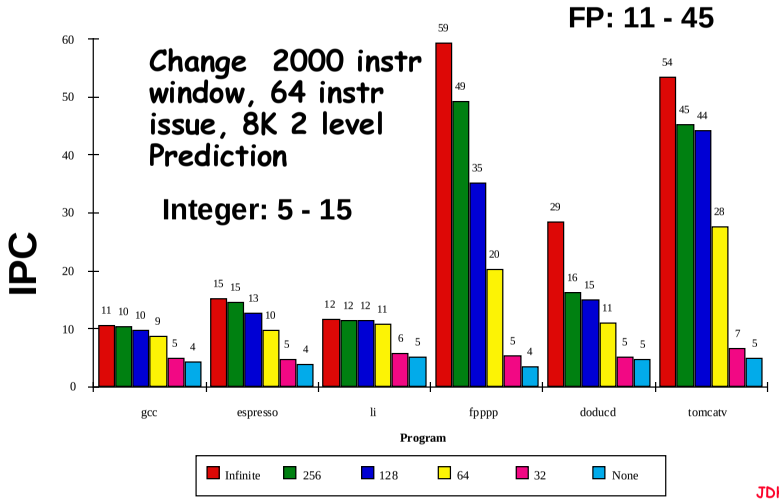
Figure 4.40, Page 323



Perfect | Pick Cor. or BHT | BHT (512) | Profile | No prediction

More Realistic HW: Register Impact

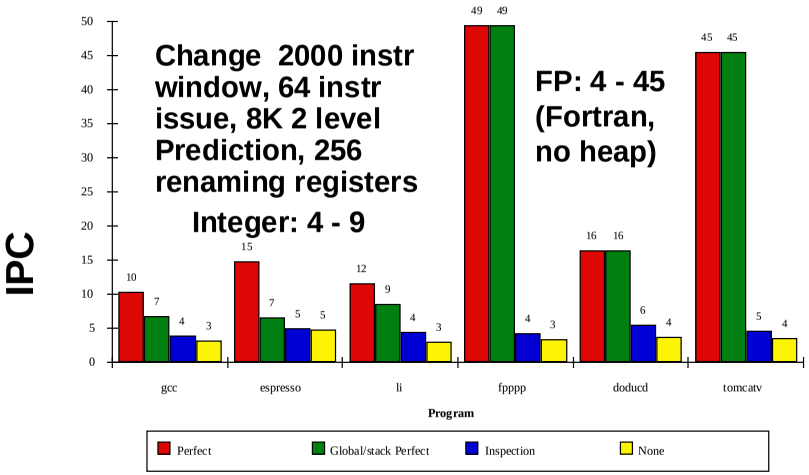
Figure 4.44, Page 328



Infinite | 256 | 128 | 64 | 32 | None

More Realistic HW: Alias Impact

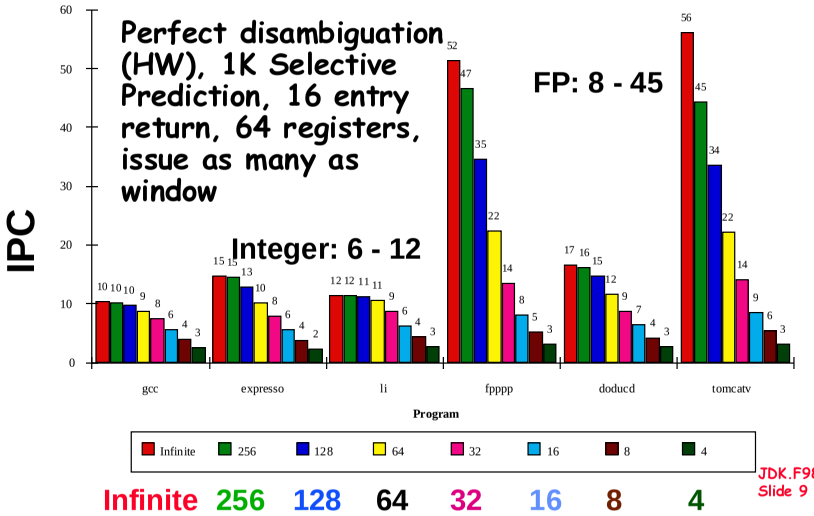
Figure 4.46, Page 330



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

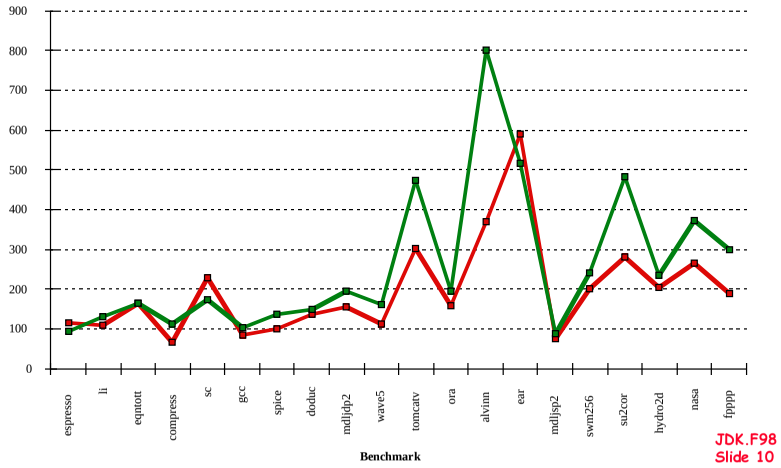
Realistic HW for '9X: Window Impact

(Figure 4.48, Page 332)



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)



Problems with scalar approach to ILP extraction

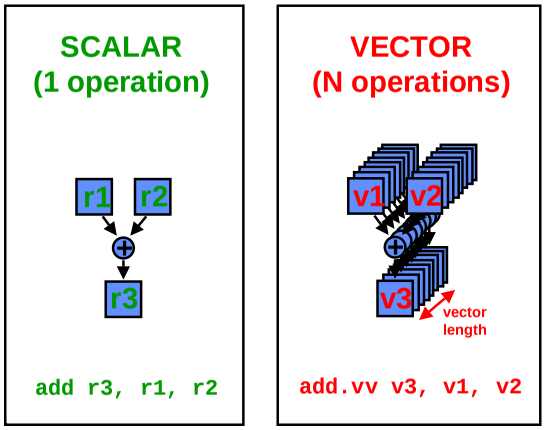
- Limits to conventional exploitation of ILP:
  1) *pipelined clock rate*: at some point, each increase in clock rate has corresponding CPI increase (branches, other hazards)
  2) *instruction fetch and decode*: hard to fetch and decode more instructions per clock cycle
  3) *cache hit rate*: some long-running (scientific) programs have very large data sets accessed with poor locality; others have continuous data streams (multimedia) and hence poor locality
  4) *power*: out-of-order, speculative execution has serious costs in terms of power consumption.
  5) *complexity*: Modern ILP processors are getting extremely complex.

Cost-performance of simple vs. OOO

MIPS MPUs	R5000	R10000	10k/5k
• Clock Rate	200 MHz	195 MHz	1.0x
• On-Chip Caches	32K/32K	32K/32K	1.0x
• Instructions/Cycle	1(+ FP)	4	4.0x
• Pipe stages	5	5-7	1.2x
• Model	In-order	Out-of-order	---
• Die Size (mm^2)	84	298	3.5x
- without cache, TLB	32	205	6.3x
• Development (man yr.)	60	300	5.0x
• SPECint_base95	5.7	8.8	1.6x

Alternative Model: Vector Processing

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



Properties of Vector Processors

- Each result independent of previous result
  => long pipeline, compiler ensures no dependencies
  => high clock rate
- Vector instructions access memory with known pattern
  => highly interleaved memory
  => amortize memory latency of over - 64 elements
  => no (data) caches required! (Do use instruction cache)
- Reduces branches and branch problems in pipelines
- Single vector instruction implies lots of work (- loop)
  => fewer instruction fetches

Operation & Instruction Count: RISC v. Vector Processor

(from F. Quintana, U. Barcelona.)

Spec92fp	Operations (Millions)			Instructions (M)		
Program	RISC	Vector	R / V	RISC	Vector	R / V
swim256	115	95	1.1x	115	0.8	142x
hydro2d	58	40	1.4x	58	0.8	71x
nasa7	69	41	1.7x	69	2.2	31x
su2cor	51	35	1.4x	51	1.8	29x
tomcatv	15	10	1.4x	15	1.3	11x
wave5	27	25	1.1x	27	7.2	4x
mdljdp2	32	52	0.6x	32	15.8	2x

Vector reduces ops by 1.2X, instructions by 20X

Styles of Vector Architectures

- *memory-memory vector processors*: all vector operations are memory to memory
- *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
  - We assume vector-register for rest of lectures

Components of Vector Processor

- *Vector Register*: fixed length bank holding a single vector
  - has at least 2 read and 1 write ports
  - typically 8-32 vector registers, each holding 64-128 64-bit elements
- *Vector Functional Units (FUs)*: fully pipelined, start new operation every clock
  - typically 4 to 8 FUs: FP add, FP mult, FP reciprocal (1/X), integer add, logical, shift; may have multiple of same unit
- *Vector Load-Store Units (LSUs)*: fully pipelined unit to load or store a vector; may have multiple LSUs
- *Scalar registers*: single element for FP scalar or address
- Cross-bar to connect FUs , LSUs, registers

"DLXV" Vector Instructions

	Instr.	Operands	Operation	Comment
•	ADDV	V1,V2,V3	V1=V2+V3	vector + vector
•	ADDSV	V1,F0,V2	V1=F0+V2	scalar + vector
•	MULTV	V1,V2,V3	V1=V2xV3	vector x vector
•	MULSV	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]	indir.("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

Memory operations

- Load/store operations move groups of data between registers and memory
- Three types of addressing
  - Unit stride
    - » Fastest
  - Non-unit (constant) stride
  - Indexed (gather-scatter)
    - » Vector equivalent of register indirect
    - » Good for sparse arrays of data
    - » Increases number of programs that vectorize

DAXPY (Y = a * X + Y)

Assuming vectors X, Y are length 64

Scalar vs. Vector →

```
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
```

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

578 (2+9*64) vs.
321 (1+5*64) ops (1.8X)

578 (2+9*64) vs.
6 instructions (96X)

64 operation vectors + no loop overhead

also 64X fewer pipeline hazards

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	8	2 L, 1 S
Cray YMP	1988	166 MHz	8	64	8	2 L, 1 S
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 (MFLOPS)

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)

CS 252 Administrivia

- Reading assignment for Friday:
  - Young et al, "A Comparative Analysis of Schemes for Correlated Branch Prediction"
  - Moshovos et all, "Dynamic Speculation and Synchronization of Data Dependences."
  - Chrysos and Emer, "Memory Dependence Prediction using Store Sets"
  - One paragraph for the first and one for the second two.

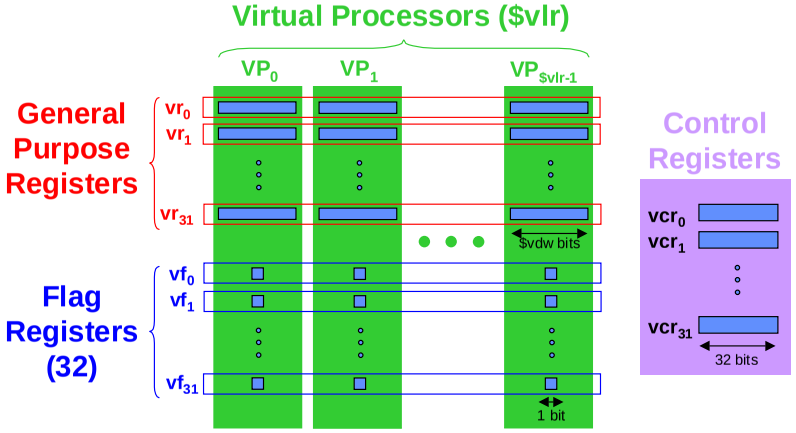
Vector Surprise

- Use vectors for inner loop parallelism (no surprise)
  - One dimension of array: A[0, 0], A[0, 1], A[0, 2], ...
  - think of machine as, say, 32 vector regs each with 64 elements
  - 1 instruction updates 64 elements of 1 vector register
- and for outer loop parallelism!
  - 1 element from each column: A[0,0], A[1,0], A[2,0], ...
  - think of machine as 64 "virtual processors" (VPs) each with 32 scalar registers! (- multithreaded processor)
  - 1 instruction updates 1 scalar register in 64 VPs
- Hardware identical, just 2 compiler perspectives

Virtual Processor Vector Model

- Vector operations are SIMD (single instruction multiple data)operations
- Each element is computed by a virtual processor (VP)
- Number of VPs given by vector length
  - vector control register

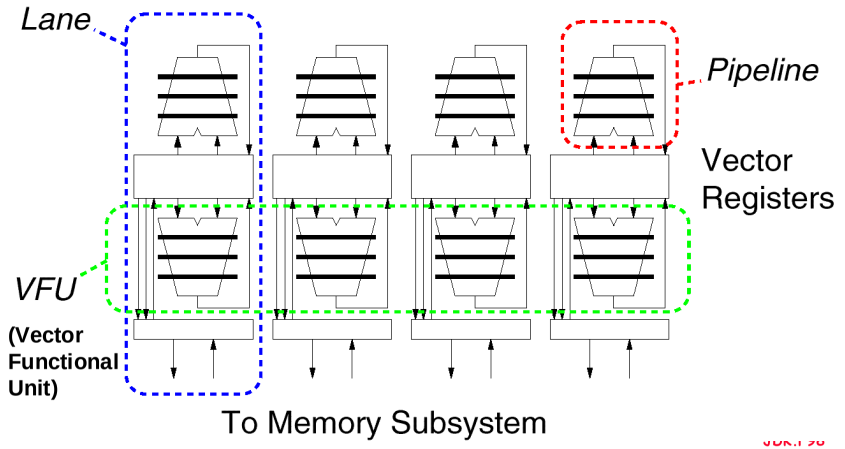
Vector Architectural State



Vector Implementation

- Vector register file
  - Each register is an array of elements
  - Size of each register determines maximum vector length
  - Vector length register determines vector length for a particular operation
- Multiple parallel execution units = "lanes" (sometimes called "pipelines" or "pipes")

Vector Terminology: 4 lanes, 2 vector functional units



Vector Execution Time

- Time = f(vector length, data dependicies, struct. hazards)
- *Initiation rate*: rate that FU consumes vector elements (= number of lanes; usually 1 or 2 on Cray T-90)
- *Convoy*: set of vector instructions that can begin execution in same clock (no struct. or data 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; good approximization for long vectors)

```
1: LV    V1,Rx     ;load vector X
2: MULV  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, 1 lane, VL=64
=> 4 x 64 = 256 clocks
(or 4 clocks per result)

DLXV Start-up Time

- *Start-up time*: pipeline latency time (depth of FU pipeline); another sources of overhead

Operation	Start-up penalty (from CRAY-1)
Vector load/store	12
Vector multply	7
Vector add	6

Assume convoys don't overlap; vector length = n:

Convoy	*Start*	*1st result*	*last result*	
1. LV	0	12	11+n (12+n-1)	
2. MULV, LV	12+n	12+n+7	18+2n	*Multiply startup*
	12+n+1	12+n+13	24+2n	*Load start-up*
3. ADDV	25+2n	25+2n+6	30+3n	*Wait convoy 2*
4. SV	31+3n	31+3n+12	42+4n	*Wait convoy 3*

Why startup time for each vector instruction?

- Why not overlap startup time of back-to-back vector instructions?
- Cray machines built from many ECL chips operating at high clock rates; hard to do?
- Berkeley vector design ("T0") didn't know it wasn't supposed to do overlap, so no startup times for functional units (except load)

Vector Load/Store Units & Memories

- Start-up overheads usually longer fo LSUs
- Memory system must sustain (# lanes x word) /clock
- 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 (see soon)
- Note: No. memory banks > memory latency to avoid stalls
  - *m* banks => *m* words per memory lantecy / clocks
  - if *m* < /, then gap in memory pipeline:

clock:	0	…	/	/+1	/+2	…	/+*m*- 1	l+m	…	2 /
word:	--	…	0	1	2	…	*m*-1	--	…	*m*

  - may have 1024 banks in SRAM

Vector Length

- What to do when vector length is not exactly 64?
- *vector-length register* (VLR) controls the length of any vector operation, including a vector load or store. (cannot be > the length of vector registers)

```
      do 10 i = 1, n
10    Y(i) = a * X(i) + Y(i)
```

- Don't know n until runtime!
  n > Max. Vector Length (MVL)?

Strip Mining

- Suppose Vector Length > Max. Vector Length (MVL)?
- *Strip mining*: generation of code such that each vector operation is done for a size S to the MVL
- 1st loop do short piece (n mod MVL), rest VL = MVL

```
low = 1
VL = (n mod MVL)  /*find the odd size piece*/
do 1 j = 0,(n / MVL)  /*outer loop*/
   do 10 i = low,low+VL-1  /*runs for length VL*/
         Y(i) = a*X(i) + Y(i)  /*main operation*/
10 continue
   low = low+VL  /*start of next vector*/
   VL = MVL  /*reset the length to max*/
1  continue
```


Common Vector Metrics

- R_∞: MFLOPS rate on an infinite-length vector
  - vector "speed of light"
  - Real problems do not have unlimited vector lengths, and the start-up penalties encountered in real problems will be larger
  - (R_n is the MFLOPS rate for a vector of length n)
- $N_{1/2}$: The vector length needed to reach one-half of R_∞
  - a good measure of the impact of start-up
- N_V: The vector length needed to make vector mode faster than scalar mode
  - measures both start-up and speed of scalars relative to vectors, quality of connection of scalar unit to vector unit

Vector Stride

- Suppose adjacent elements not sequential in memory

```
do 10 i = 1,100
     do 10 j = 1,100
          A(i,j) = 0.0
          do 10 k = 1,100
10                A(i,j) = A(i,j)+B(i,k)*C(k,j)
```

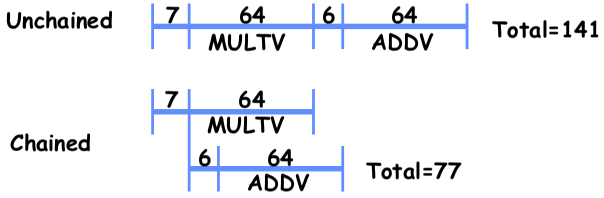
- Either B or C accesses not adjacent (800 bytes between)
- *stride*: distance separating elements that are to be merged into a single vector (caches do unit stride)
  => LVWS (load vector with stride) instruction
- Strides => can cause bank conflicts
  (e.g., stride = 32 and 16 banks)
- Think of address per vector element

Vector Opt #1: Chaining

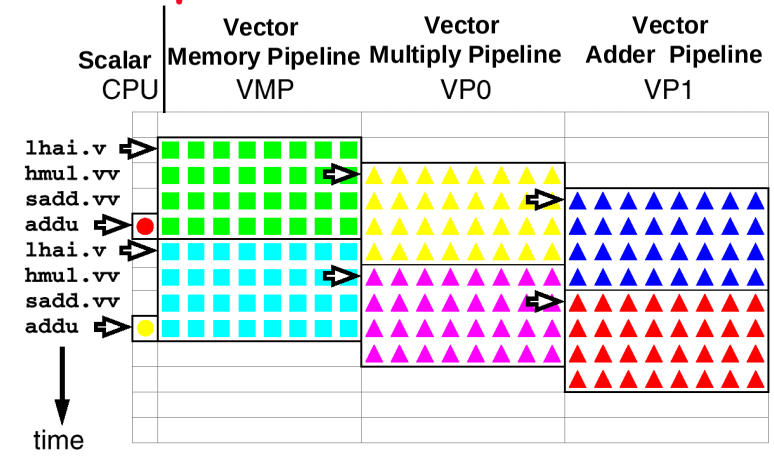
- Suppose:

```
MULV    V1,V2,V3
ADDV    V4,V1,V5   ; separate convoy?
```

- *chaining*: vector register (V1) is not as a single entity but as a group of individual registers, then pipeline forwarding can work on individual elements of a vector
- *Flexible chaining*: allow vector to chain to any other active vector operation => more read/write ports
- As long as enough HW, increases convoy size



Example Execution of Vector Code



8 lanes, vector length 32, chaining

Vector Opt #2: Conditional Execution

- Suppose:

```
    do 100 i = 1, 64
        if (A(i) .ne. 0) then
            A(i) = A(i) - B(i)
        endif
100 continue
```

- *vector-mask control* takes a Boolean vector: when *vector-mask register* is loaded from vector test, vector instructions operate only on vector elements whose corresponding entries in the vector-mask register are 1.
- Still requires clock even if result not stored; if still performs operation, what about divide by 0?

Vector Opt #3: Sparse Matrices

- Suppose:

```
        do    100 i = 1,n
100     A(K(i)) = A(K(i)) + C(M(i))
```

- *gather* (LVI) operation takes an *index vector* and fetches data from each address in the index vector
  - This produces a "dense" vector in the vector registers
- After these elements are operated on in dense form, the sparse vector can be stored in expanded form by a *scatter* store (SVI), using the same index vector
- Can't be figured out by compiler since can't know elements distinct, no dependencies
- Use CVI to create index 0, 1xm, 2xm, ..., 63xm

Sparse Matrix Example

- Cache (1993) vs. Vector (1988)

	IBM RS6000	Cray YMP
Clock	72 MHz	167 MHz
Cache	256 KB	0.25 KB
Linpack	140 MFLOPS	160 (1.1)
Sparse Matrix (Cholesky Blocked)	17 MFLOPS	125 (7.3)

- Cache: 1 address per cache block (32B to 64B)
- Vector: 1 address per element (4B)

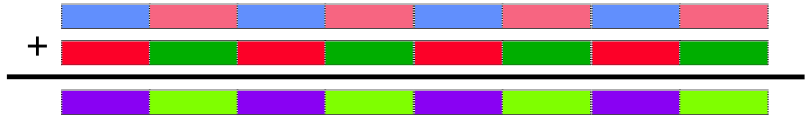
Applications

Limited to scientific computing?

- Multimedia Processing (compress., graphics, audio synth, image proc.)
- Standard benchmark kernels (Matrix Multiply, FFT, Convolution, Sort)
- Lossy Compression (JPEG, MPEG video and audio)
- Lossless Compression (Zero removal, RLE, Differencing, LZW)
- Cryptography (RSA, DES/IDEA, SHA/MD5)
- Speech and handwriting recognition
- Operating systems/Networking (memcpy, memset, parity, checksum)
- Databases (hash/join, data mining, image/video serving)
- Language run-time support (stdlib, garbage collection)
- even SPECint95

Vector for Multimedia?

- Intel MMX: 57 new 80x86 instructions (1st since 386)
  - similar to Intel 860, Mot. 88110, HP PA-71000LC, UltraSPARC
- 3 data types: 8 8-bit, 4 16-bit, 2 32-bit in 64bits
  - reuse 8 FP registers (FP and MMX cannot mix)
- - short vector: load, add, store 8 8-bit operands



- Claim: overall speedup 1.5 to 2X for 2D/3D graphics, audio, video, speech, comm., ...
  - use in drivers or added to library routines; no compiler

MMX Instructions

- Move 32b, 64b
- Add, Subtract in parallel: 8 8b, 4 16b, 2 32b
  - opt. signed/unsigned saturate (set to max) if overflow
- Shifts (sll,srl, sra), And, And Not, Or, Xor in parallel: 8 8b, 4 16b, 2 32b
- Multiply, Multiply-Add in parallel: 4 16b
- Compare = , > in parallel: 8 8b, 4 16b, 2 32b
  - sets field to 0s (false) or 1s (true); removes branches
- Pack/Unpack
  - Convert 32b<-> 16b, 16b <-> 8b
  - Pack saturates (set to max) if number is too large

Vectors and Variable Data Width

- Programmer thinks in terms of vectors of data of some width (8, 16, 32, or 64 bits)
- Good for multimedia; More elegant than MMX-style extensions
- Don't have to worry about how data stored in hardware
  - No need for explicit pack/unpack operations
- Just think of more virtual processors operating on narrow data
- Expand Maximum Vector Length with decreasing data width:
  64 x 64bit, 128 x 32 bit, 256 x 16 bit, 512 x 8 bit

Mediaprocessing: Vectorizable? Vector Lengths?

Kernel	Vector length
Matrix transpose/multiply	# vertices at once
DCT (video, communication)	image width
FFT (audio)	256-1024
Motion estimation (video)	image width, iw/16
Gamma correction (video)	image width
Haar transform (media mining)	image width
Median filter (image processing)	image width
Separable convolution (img. proc.)	image width

(from Pradeep Dubey - IBM, http://www.research.ibm.com/people/p/pradeep/tutor.html)

Compiler Vectorization on Cray XMP

Benchmark	%FP	%FP in vector	
ADM	23%	68%	
DYFESM	26%	95%	
FLO52	41%	100%	
MDG	28%	27%	
MG3D	31%	86%	
OCEAN	28%	58%	
QCD	14%	1%	
SPICE	16%	7%	(1% overall)
TRACK	9%	23%	
TRFD	22%	10%	

Vector Pitfalls

- Pitfall: Concentrating on peak performance and ignoring start-up overhead:
  N_V (length faster than scalar) > 100!
- Pitfall: Increasing vector performance, without comparable increases in scalar performance (Amdahl's Law)
  - failure of Cray competitor (ETA) from his former company
- Pitfall: Good processor vector performance without providing good memory bandwidth
  - MMX?

Vector Advantages

- Easy to get high performance; N operations:
  - are independent
  - use same functional unit
  - access disjoint registers
  - access registers in same order as previous instructions
  - access contiguous memory words or known pattern
  - can exploit large memory bandwidth
  - hide memory latency (and any other latency)
- Scalable: (get higher performance by adding HW resources)
- Compact: Describe N operations with 1 short instruction
- Predictable: performance vs. statistical performance (cache)
- Multimedia ready: N * 64b, 2N * 32b, 4N * 16b, 8N * 8b
- Mature, developed compiler technology
- Vector Disadvantage: Out of Fashion?
  - Hard to say. Many irregular loop structures seem to still be hard to vectorize automatically.
  - Theory of some researchers that SIMD model has great potential.

Vector Summary

- Vector model accommodates long memory latency, doesn't rely on caches as does Out-Of-Order, superscalar/VLIW designs
- Much easier for hardware: more powerful instructions, more predictable memory accesses, fewer hazards, fewer branches, fewer mispredicted branches, ...
- What % of computation is vectorizable?
- Is vector a good match to new apps such as multimedia, DSP?

Prediction: Branches, Dependencies, Data New era in computing?

- Prediction has become essential to getting good performance from scalar instruction streams.
- We will discuss predicting branches, data dependencies, actual data, and results of groups of instructions:
  - At what point does computation become a probabilistic operation + verification?
  - We are pretty close with control hazards already…
- Why does prediction work?
  - Underlying algorithm has regularities.
  - Data that is being operated on has regularities.
  - Instruction sequence has redundancies that are artifacts of way that humans/compilers think about problems.
- Prediction ⇒ Compressible information streams?

Dynamic Branch Prediction

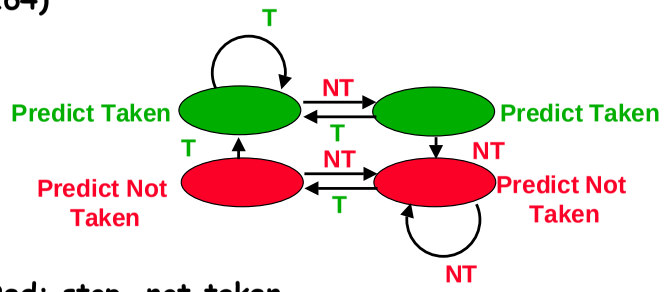
- Is dynamic branch prediction better than static branch prediction?
  - Seems to be. Still some debate to this effect
  - Josh Fisher had good paper on "Predicting Conditional Branch Directions from Previous Runs of a Program." ASPLOS '92. In general, good results if allowed to run program for lots of data sets.
    - » How would this information be stored for later use?
    - » Still some difference between best possible static prediction (using a run to predict itself) and weighted average over many different data sets
  - Paper by Young et all, "A Comparative Analysis of Schemes for Correlated Branch Prediction" notices that there are a small number of important branches in programs which have dynamic behavior.

Dynamic Branch Prediction

- Performance = *f*(accuracy, cost of misprediction)
- Branch History Table: 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
(Jim Smith, 1981)

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



- Red: stop, not taken
- Green: go, taken
- Adds *hysteresis* to decision making process

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

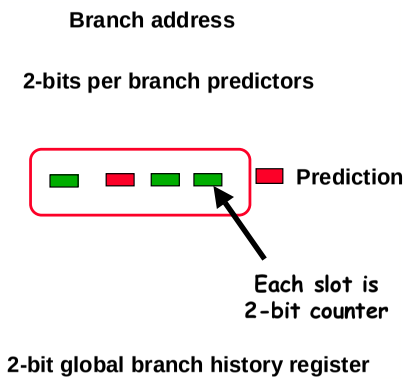
- Hypothesis: recent branches are correlated; that is, behavior of recently executed branches affects prediction of current branch
- Two possibilities; Current branch depends on:
  - Last m most recently executed branches anywhere in program
    Produces a "GA" (for "global address") in the Yeh and Patt classification (e.g. GAg)
  - Last m most recent outcomes of same branch.
    Produces a "PA" (for "per address") in same classification (e.g. PAg)
- Idea: record m most recently executed branches as taken or not taken, and use that pattern to select the proper branch history table entry
  - A single history table shared by all branches (appends a "g" at end), indexed by history value.
  - Address is used along with history to select table entry (appends a "p" at end of classification)
  - If only portion of address used, often appends an "s" to indicate "set-indexed" tables (I.e. GAs)

Correlating Branches

- For instance, consider global history, set-indexed BHT. That gives us a GAs history table.

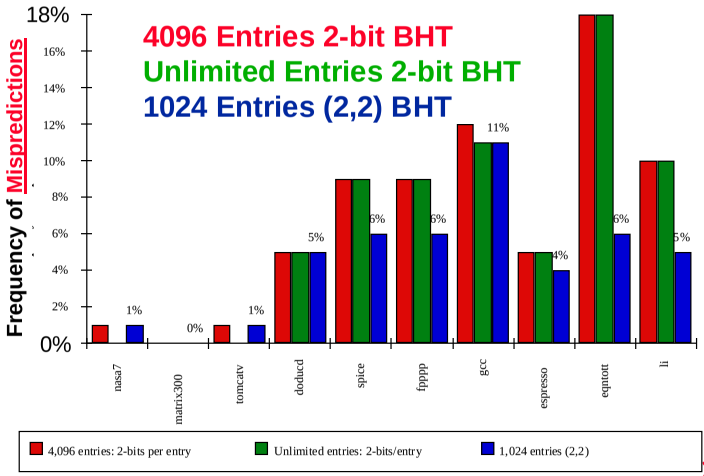
(2,2) GAs predictor

- First 2 means that we keep two bits of history
- Second means that we have 2 bit counters in each slot.
- Then behavior of recent branches selects between, say, four predictions of next branch, updating just that prediction
- Note that the original two-bit counter solution would be a (0,2) GAs predictor
- Note also that aliasing is possible here...



Accuracy of Different Schemes

(Figure 4.21, p. 272)



Re-evaluating Correlation

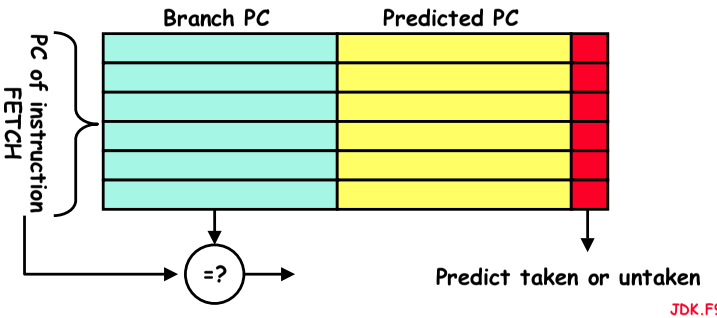
- Several of the SPEC benchmarks have less than a dozen branches responsible for 90% of taken branches:

program	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?

Need Address at 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)



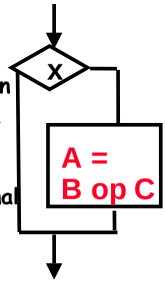
- Return instruction addresses predicted with stack

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.
  - IA-64: 64 1-bit condition fields selected so conditional execution of any instruction



- 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

Dynamic Branch Prediction Summary

- Prediction becoming important part of scalar execution.
  - Prediction is exploiting "information compressibility" in execution
- Branch History Table: 2 bits for loop accuracy
- Correlation: Recently executed branches correlated with next branch.
  - Either different branches (GA)
  - Or different executions of same branches (PA).
- Branch Target Buffer: include branch address & prediction
- Predicated Execution can reduce number of branches, number of mispredicted branches