CS252
Graduate Computer Architecture
Lecture 10

Vector Processing (Continued)
Branch Prediction

October 4, 2000

Prof. John Kubiatowicz

Review: Vector Processing

- Vector Processing represents an alternative to complicated superscalar processors.
- Primitive operations on large vectors of data
- Load/store architecture:
  - Data loaded into vector registers; computation is register to register.
  - Memory system can take advantage of predictable access patterns:
    - » Unit stride, Non-unit stride, indexed
- Vector processors exploit large amounts of parallelism without data and control hazards:
  - Every element is handled independently and possibly in parallel
  - Same effect as scalar loop without the control hazards or complexity of tomasulo-style hardware
- Hardware parallelism can be varied across a wide range by changing number of *vector lanes* in each vector functional unit.

Review: ILP? Wherefore art thou?

- There is a fair amount of ILP available, but branches get in the way
  - Better branch prediction techniques? Probably not much room to go still: prediction rates already up in the 93% and above
  - Fundamental new programming model?
- Vector model accommodates long memory latency, doesn't rely on caches as does Out-Of-Order, superscalar/VLIW designs:
  - No branch prediction! Loops are implicit in model
  - Much easier for hardware: more powerful instructions, more predictable memory accesses, fewer hazards, fewer branches, fewer mispredicted branches, ...
  - But, what % of computation is vectorizable?
  - Is vector a good match to new apps such as multimedia, DSP?
- Right answer? Both? Neither? (my favorite)

Review: "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

Vector Example with dependency

```
/* Multiply a[m][k] * b[k][n] to get c[m][n] */
for (i=1; i<m; i++)
{
    for (j=1; j<n; j++)
    {
        sum = 0;
        for (t=1; t<k; t++)
        {
            sum += a[i][t] * b[t][j];
        }
        c[i][j] = sum;
    }
}
```


Straightforward Solution: Use scalar processor

- This type of operation is called a reduction
- Grab one element at a time from a vector register and send to the scalar unit?
  - Usually bad, since path between scalar processor and vector processor not usually optimized all that well
- Alternative: Special operation in vector processor
  - shift all elements left vector length elements or collapse into a compact vector all elements not masked
  - Supported directly by some vector processors
  - Usually not as efficient as normal vector operations
    - » (Number of cycles probably logarithmic in number of bits!)

Novel Matrix Multiply Solution

- You don't need to do reductions for matrix multiply
- You can calculate multiple independent sums within one vector register
- You can vectorize the j loop to perform 32 dot-products at the same time

- (Assume Maximul Vector Length is 32)
- Show it in C source code, but can imagine the assembly vector instructions from it

Optimized Vector Example

```
/* Multiply a[m][k] * b[k][n] to get c[m][n] */
for (i=1; i<m; i++){
  for (j=1; j<n; j+=32){/* Step j 32 at a time. */
    sum[0:31] = 0; /* Init vector reg to zeros. */
    for (t=1; t<k; t++) {
      a_scalar = a[i][t]; /* Get scalar */
      b_vector[0:31] = b[t][j:j+31]; /* Get vector */

      /* Do a vector-scalar multiply. */
      prod[0:31] = b_vector[0:31]*a_scalar;

      /* Vector-vector add into results. */
      sum[0:31] += prod[0:31];
    }
    /* Unit-stride store of vector of results. */
    c[i][j:j+31] = sum[0:31];
  }
}
```

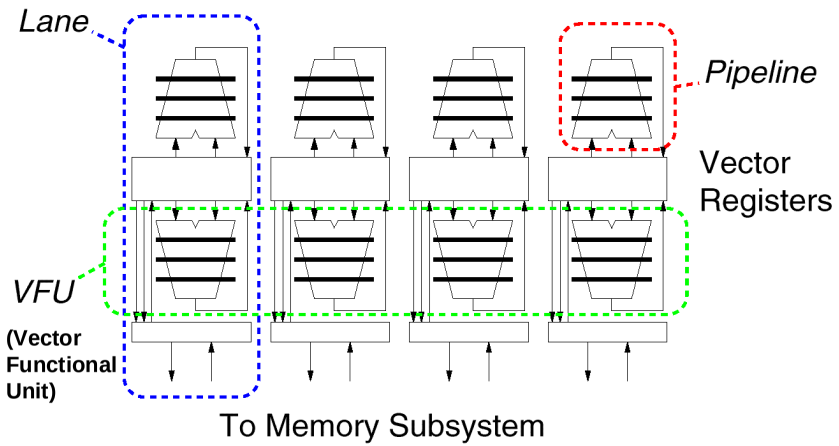
Novel, Step #2

- It's actually better to interchange the i and j loops, so that you only change vector length once during the whole matrix multiply
- To get the absolute fastest code you have to do a little register blocking of the innermost loop.

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 convoys, 1 lane, VL=64
=> 4 x 64 = 256 clocks
(or 4 clocks per result)

Hardware Vector Length

- What to do when software vector length doesn't exactly match hardware vector length?
- *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
```


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 multiply	7
Vector add	6

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

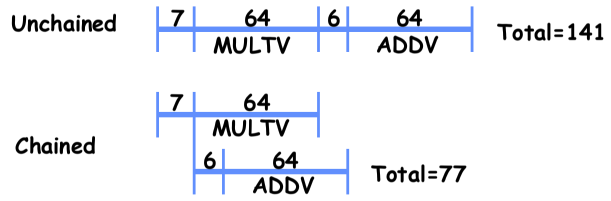
Convoy	*Start*	*1st result*	*last result*	
1. LV	0	12	11+n (=12+n-1)	
2. MULV, LV	12+n	12+n+7	18+2n	*Multiply startup*
	12+n+1	12+n+13	24+2n	*Load start-up*
3. ADDV	25+2n	25+2n+6	30+3n	*Wait convoy 2*
4. SV	31+3n	31+3n+12	42+4n	*Wait convoy 3*

Vector Opt #1: Chaining

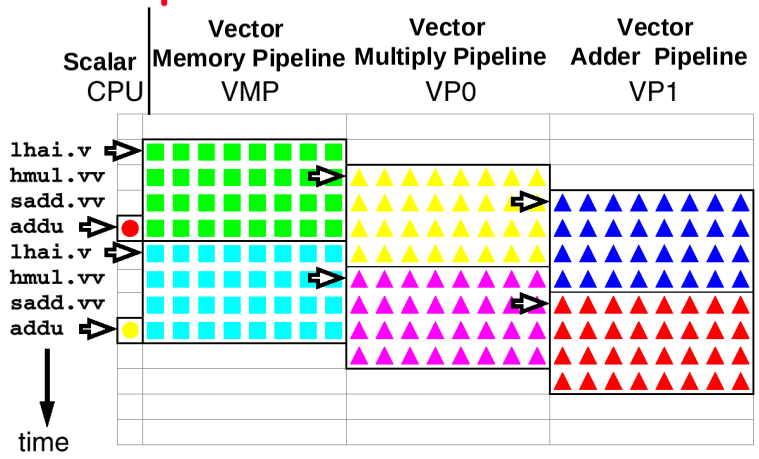
- Suppose:

```
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

Operations

Instruction issue

Vector Stride

- Suppose adjacent elements not sequential in memory

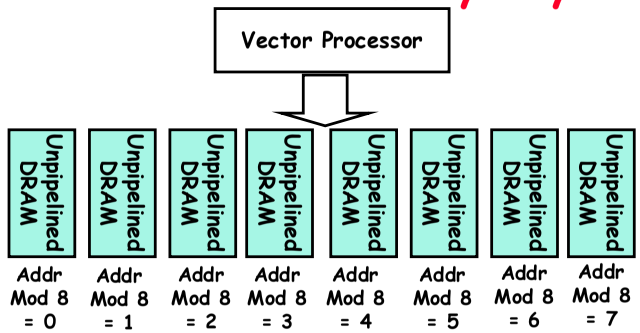
```
do 10 i = 1,100
      do 10 j = 1,100
            A(i,j) = 0.0
                  do 10 k = 1,100
10                      A(i,j) = A(i,j)+B(i,k)*C(k,j)
```

- 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
- Think of addresses per vector element

Memory operations

- Load/store operations move groups of data between registers and memory
- Three types of addressing
  - Unit stride
    - » Contiguous block of information in memory
    - » Fastest: always possible to optimize this
  - Non-unit (constant) stride
    - » Harder to optimize memory system for all possible strides
    - » Prime number of data banks makes it easier to support different strides at full bandwidth
  - Indexed (gather-scatter)
    - » Vector equivalent of register indirect
    - » Good for sparse arrays of data
    - » Increases number of programs that vectorize

Interleaved Memory Layout



- Great for unit stride:
  - Contiguous elements in different DRAMs
  - Startup time for vector operation is latency of single read
- What about non-unit stride?
  - Above good for strides that are relatively prime to 8
  - Bad for: 2, 4
  - Better: prime number of banks…!

How to get full bandwidth for Unit Stride?

- Memory system must sustain (# lanes x word) /clock
- 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
- If desired throughput greater than one word per cycle
  - Either more banks (start multiple requests simultaneously)
  - Or wider DRAMS. Only good for unit stride or large data types
- More banks/weird numbers of banks good to support more strides at full bandwidth
  - Will read paper on how to do prime number of banks efficiently

Vector Opt #2: 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)

Vector Opt #3: 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?

CS 252 Administrivia

- Exam: Wednesday 10/18
  Location: TBA
  TIME: 5:30 - 8:30
- This info is on the Lecture page (has been)
- Meet at LaVal's afterwards for Pizza and Beverages

- Assignment up now
  - Due on Friday Oct 13th!
  - Done in pairs. Put both names on papers.
  - Make sure you have partners! Feel free to use mailing list for this.

Parallelism and Power

- If code is vectorizable, then simple hardware, more energy efficient than Out-of-order machines.
- Can decrease power by lowering frequency so that voltage can be lowered, then duplicating hardware to make up for slower clock:

$$\text{Power} \propto CV^2 f$$

$$\left\{ \begin{array}{c} f = \frac{1}{n} f_0 \\ Lanes = n \times Lanes_0 \\ V = \delta V_0;\ \delta < 1 \end{array} \right\} \Rightarrow \left\{ \begin{array}{l} \text{Performance Constant} \\ \text{Power Change}: \delta^2 < 1 \end{array} \right\}$$

- Note that V_o can be made as small as permissible within process constraints by simply increasing "n"

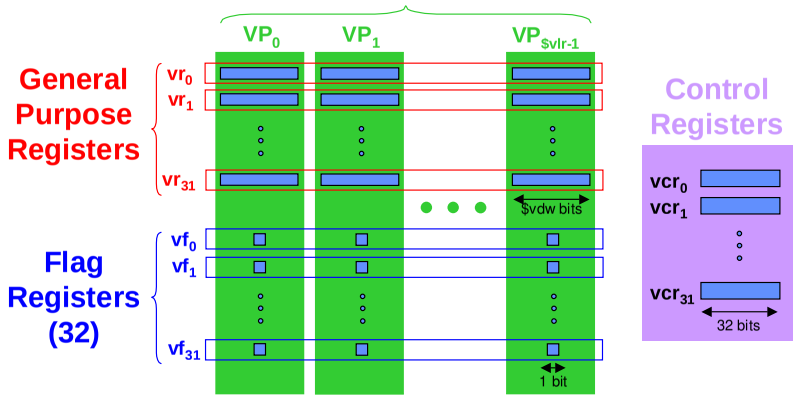
Vector Options

- 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, 16 vector regs each with 32 elements
  - 1 instruction updates 32 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 32 "virtual processors" (VPs) each with 16 scalar registers! (~ multithreaded processor)
  - 1 instruction updates 1 scalar register in 64 VPs
- Hardware identical, just 2 compiler perspectives

Virtual Processor Vector Model: Treat like SIMD multiprocessor

- Vector operations are SIMD (single instruction multiple data) operations
  - Each virtual processor has as many scalar "registers" as there are vector registers
  - There are as many virtual processors as current vector length.
  - Each element is computed by a virtual processor (VP)

Vector Architectural State



Designing a Vector Processor

- Changes to scalar
- How Pick Vector Length?
- How Pick Number of Vector Registers?
- Context switch overhead
- Exception handling
- Masking and Flag Instructions

Changes to scalar processor to run vector instructions

- Decode vector instructions
- Send scalar registers to vector unit (vector-scalar ops)
- Synchronization for results back from vector register, including exceptions
- Things that don't run in vector don't have high ILP, so can make scalar CPU simple

How Pick Vector Length?

- Longer good because:
  1) Hide vector startup
  2) lower instruction bandwidth
  3) tiled access to memory reduce scalar processor memory bandwidth needs
  4) if know max length of app. is < max vector length, no strip mining overhead
  5) Better spatial locality for memory access
- Longer not much help because:
  1) diminishing returns on overhead savings as keep doubling number of element
  2) need natural app. vector length to match physical register length, or no help (lots of short vectors in modern codes!)

How Pick Number of Vector Registers?

- More Vector Registers:
  1) Reduces vector register "spills" (save/restore)
     - » 20% reduction to 16 registers for su2cor and tomcatv
     - » 40% reduction to 32 registers for tomcatv
     - » others 10%-15%
  2) Aggressive scheduling of vector instructinons: better compiling to take advantage of ILP
- Fewer:
  1) Fewer bits in instruction format (usually 3 fields)
  2) Easier implementation

Context switch overhead: Huge amounts of state!

- Extra dirty bit per processor
  - If vector registers not written, don't need to save on context switch
- Extra valid bit per vector register, cleared on process start
  - Don't need to restore on context switch until needed

Exception handling: External Interrupts?

- If external exception, can just put pseudo-op into pipeline and wait for all vector ops to complete
  - Alternatively, can wait for scalar unit to complete and begin working on exception code assuming that vector unit will not cause exception and interrupt code does not use vector unit

Exception handling: Arithmetic Exceptions

- Arithmetic traps harder
- Precise interrupts => large performance loss!
- Alternative model: arithmetic exceptions set vector flag registers, 1 flag bit per element
- Software inserts trap barrier instructions from SW to check the flag bits as needed
- IEEE Floating Point requires 5 flag bits

Exception handling: Page Faults

- Page Faults must be precise
- Instruction Page Faults not a problem
  - Could just wait for active instructions to drain
  - Also, scalar core runs page-fault code anyway
- Data Page Faults harder
- Option 1: Save/restore internal vector unit state
  - Freeze pipeline, dump vector state
  - perform needed ops
  - Restore state and continue vector pipeline

Exception handling: Page Faults

- Option 2: expand memory pipeline to check addresses before send to memory + memory buffer between address check and registers
  - multiple queues to transfer from memory buffer to registers; check last address in queues before load 1st element from buffer.
  - Per Address Instruction Queue (PAIQ) which sends to TLB and memory while in parallel go to Address Check Instruction Queue (ACIQ)
  - When passes checks, instruction goes to Committed Instruction Queue (CIQ) to be there when data returns.
  - On page fault, only save intructions in PAIQ and ACIQ

Masking and Flag Instructions

- Flag have multiple uses (conditional, arithmetic exceptions)
- Alternative is conditional move/merge
- Clear that fully masked is much more effiecient that with conditional moves
  - Not perform extra instructions, avoid exceptions
- Downside is:

1) extra bits in instruction to specify the flag regsiter
2) extra interlock early in the pipeline for RAW hazards on Flag registers

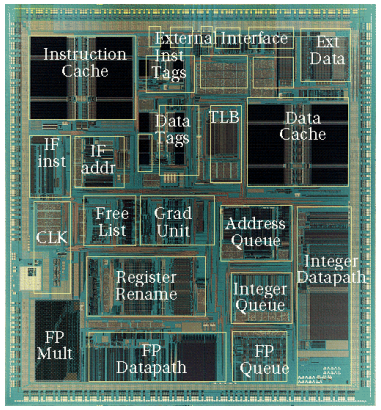
Flag Instruction Ops

- Do in scalar processor vs. in vector unit with vector ops?
- Disadvantages to using scalar processor to do flag calculations (as in Cray):

1) if MVL > word size => multiple instructions; also limits MVL in future
2) scalar exposes memory latency
3) vector produces flag bits 1/clock, but scalar consumes at 64 per clock, so cannot chain together

- Proposal: separate Vector Flag Functional Units and instructions in VU

MIPS R10000 vs. T0



*See http://www.icsi.berkeley.edu/real/spert/t0-intro.html

Vectors Are Inexpensive

Scalar

- N ops per cycle
  $\Rightarrow O(N^2)$ circuitry
- HP PA-8000
  - 4-way issue
  - reorder buffer: 850K transistors
    - incl. 6,720 5-bit register number comparators

Vector

- N ops per cycle
  $\Rightarrow O(N + \varepsilon N^2)$ circuitry
- T0 vector micro
  - 24 ops per cycle
  - 730K transistors total
    - only 23 5-bit register number comparators
  - No floating point

Vectors Lower Power

Single-issue Scalar

- One instruction fetch, decode, dispatch per operation
- Arbitrary register accesses, adds area and power
- Loop unrolling and software pipelining for high performance increases instruction cache footprint
- All data passes through cache; waste power if no temporal locality
- One TLB lookup per load or store
- Off-chip access in whole cache lines

Vector

- One inst fetch, decode, dispatch per vector
- Structured register accesses
- Smaller code for high performance, less power in instruction cache misses
- Bypass cache
- One TLB lookup per group of loads or stores
- Move only necessary data across chip boundary

Superscalar Energy Efficiency Even Worse

Superscalar

- Control logic grows quad-ratically with issue width
- Control logic consumes energy regardless of available parallelism
- Speculation to increase visible parallelism wastes energy

Vector

- Control logic grows linearly with issue width
- Vector unit switches off when not in use
- Vector instructions expose parallelism without speculation
- Software control of speculation when desired:
  - Whether to use vector mask or compress/expand for conditionals

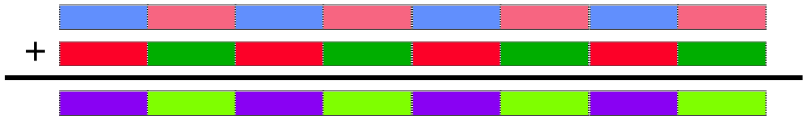
Vector Applications

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

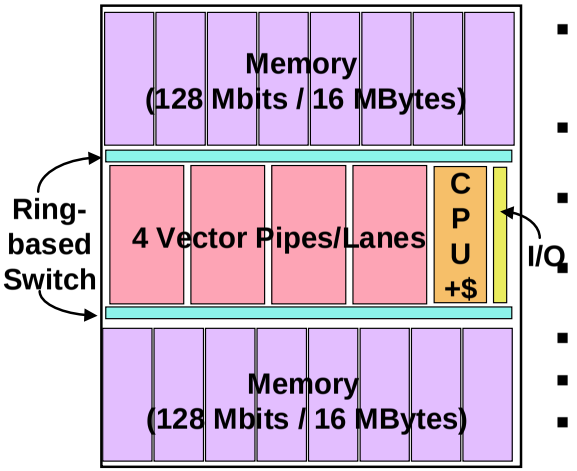
MMX Instructions

- 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

New Architecture Directions

- "…media processing will become the dominant force in computer arch. & microprocessor design."
- "... new media-rich applications... involve significant real-time processing of continuous media streams, and make heavy use of vectors of packed 8-, 16-, and 32-bit integer and Fl. Pt."
- Needs include high memory BW, high network BW, continuous media data types, real-time response, fine grain parallelism
  - "How Multimedia Workloads Will Change Processor Design", Diefendorff & Dubey, *IEEE Computer* (9/97)

Return of vectors: Tentative VIRAM-1 Floorplan

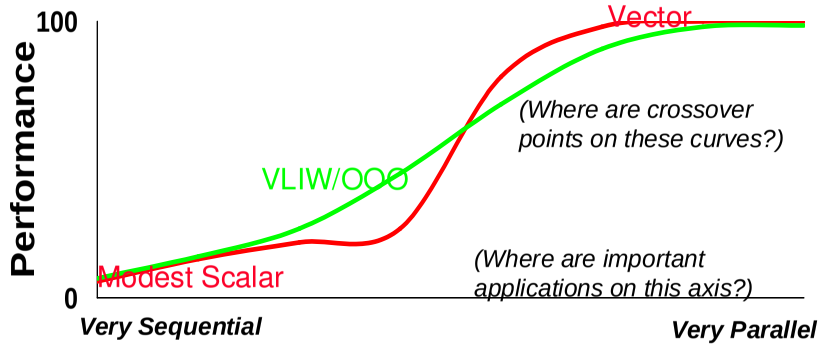


- 0.18 µm DRAM 32 MB in 16 banks x 256b, 128 subbanks
- 0.25 µm, 5 Metal Logic
- - 200 MHz MIPS, 16K I$, 16K D$
- - 4 200 MHz FP/int. vector units
- die: - 16x16 mm
- xtors: - 270M
- power: -2 Watts

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%	

VLIW/Out-of-Order vs. Modest Scalar+Vector



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.

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

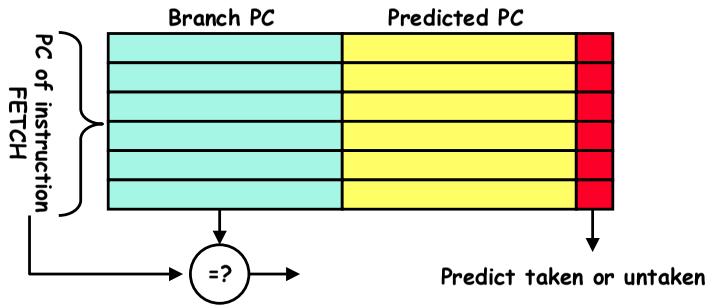
- Prediction has become essential to getting good performance from scalar instruction streams.
- We will discuss predicting branches, data dependencies, actual data, and results of groups of instructions:
  - At what point does computation become a probabilistic operation + verification?
  - We are pretty 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.

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)



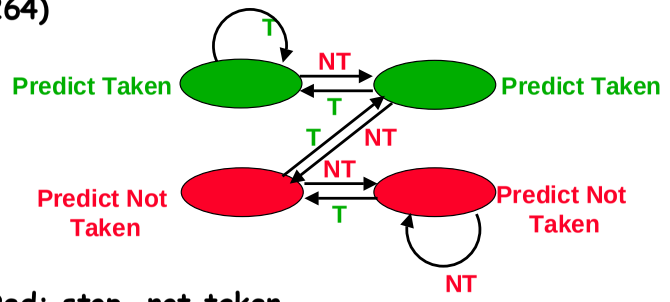
- Return instruction addresses predicted with stack

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

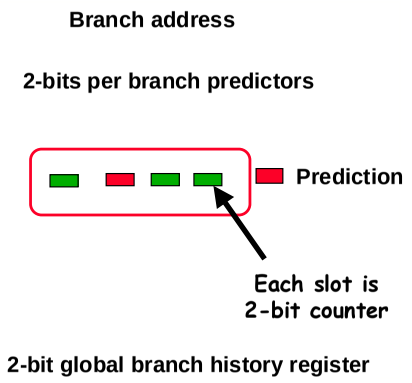
- Hypothesis: recent branches are correlated; that is, behavior of recently executed branches affects prediction of current branch
- Two possibilities; Current branch depends on:
  - Last m most recently executed branches anywhere in program Produces a "GA" (for "global address") in the Yeh and Patt 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.

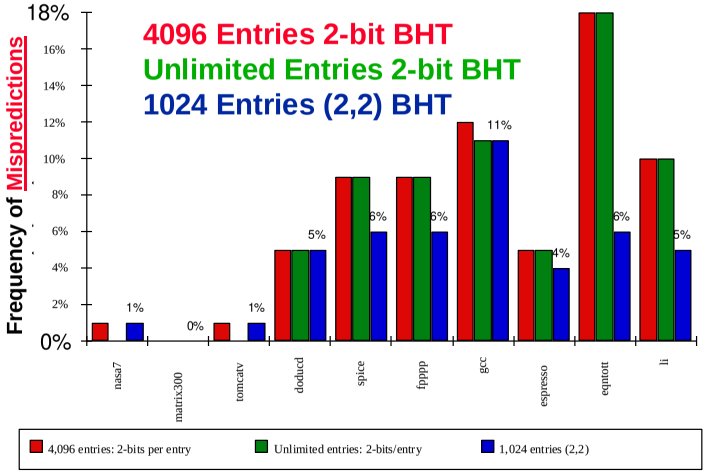
(2,2) GAs predictor

- 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

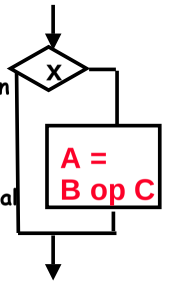
- Real programs + OS more like gcc
- Small benefits beyond benchmarks for correlation? problems with branch aliases?

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
  - This transformation is called "if-conversion"



- 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

Summary #1
Vector Processing

- Vector is alternative model for exploiting ILP
- If code is vectorizable, then simpler hardware, more energy efficient, and better real-time model than Out-of-order machines
- Design issues include number of lanes, number of functional units, number of vector registers, length of vector registers, exception handling, conditional operations
- Will multimedia popularity revive vector architectures?

Summary #2
Dynamic Branch Prediction

- 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