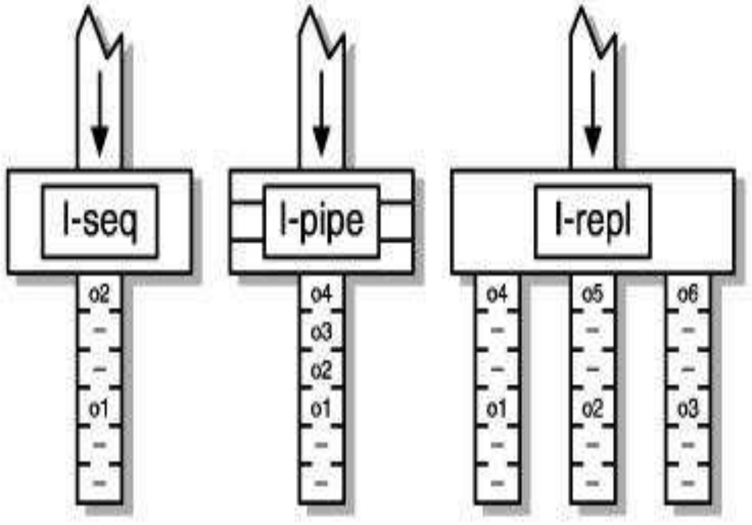
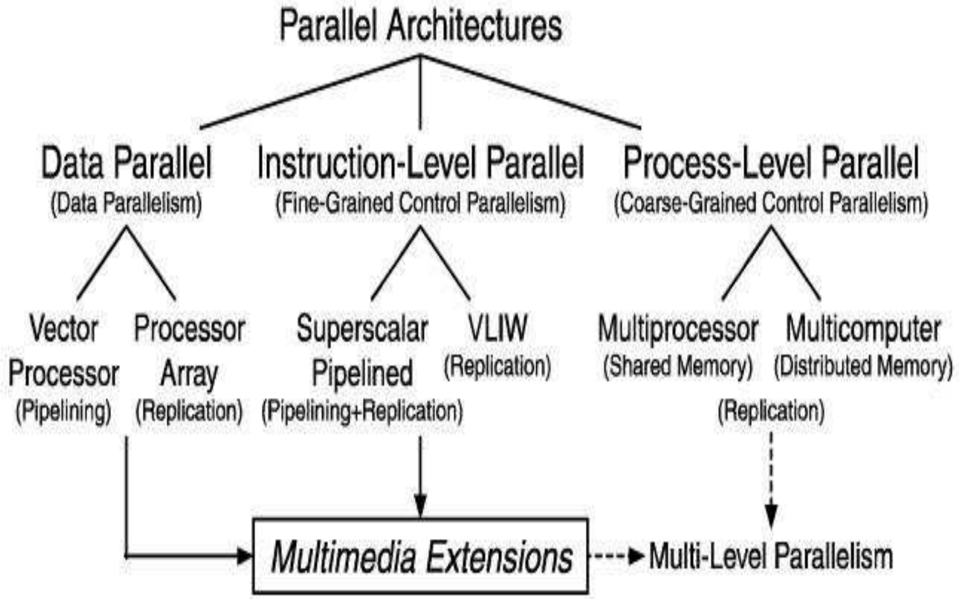
Vector Processors

Abhishek Kulkarni
Girish Subramanian

Classification of Parallel Architectures





Hennessy and Patterson 1990; Sima, Fountain, and Kacsuk 1997

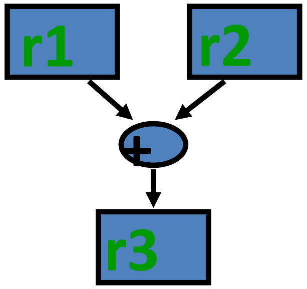
Why Vector Processors?

- Difficulties in exploiting ILP
  - Deeper the pipeline, more complex circuitry required (reorder buffer, register renaming etc.)
  - Deep pipeline implies more instructions in-flight (partially executed) hence more control hazards, data hazards etc.
  - Even with VLIW complex circuitry is involved and also increases compiler complexity.
- Cache Hit Rate
  - Scalar processors depend upon cache hit for performance. Scientific applications have very large data sets with poor memory locality.

Vector Processing Model

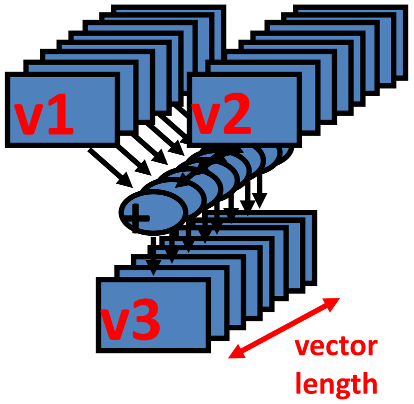
- Vector processors have high-level operations that work on linear arrays of numbers: "vectors"

SCALAR
(1 operation)



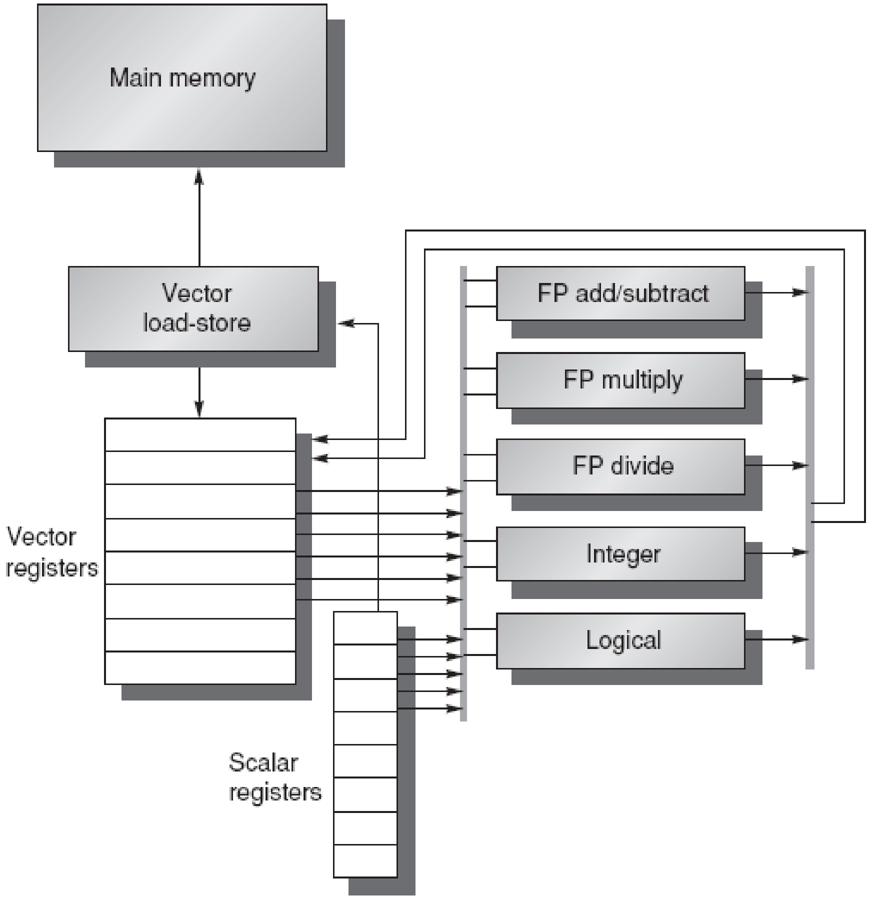
`add r3, r1, r2`

VECTOR
(N operations)



`add.vv v3, v1, v2`

Basic Vector Processor Architecture



Components of vector processors

a. Vector Registers
b. Vector Functional Units
c. Vector Load-Store Units
d. Scalar Registers

Styles of Vector Architecture

a. memory-memory vector processors : all vector operations are memory to memory
b. <u>Vector-register processors : all vector operations between vector registers.</u>

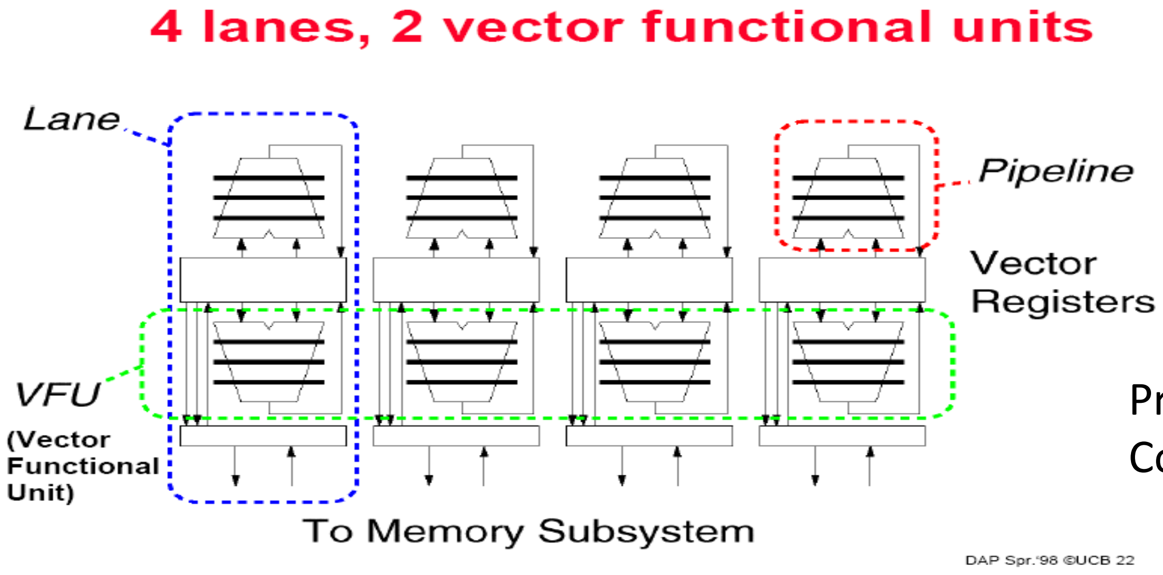
Appendix F

Vector Registers

- Consists a fixed number of vector register. (typically 8-32)
- Each register is an array of elements, each holding 64-128 64bit elements
- Has at least 2 read and 1 write ports.
- Example : Cray X1 has 32 vector registers each having 64 bit elements.
- Types
  - General Purpose registers
  - Flag Registers
  - Control registers

Vector Functional Units

- Fully pipelined, start new operation every clock.
- Typically 2-8 Functional units.
- Multiple parallel execution units called “lanes”



Professor David A. Patterson
Computer Science 252, Spring 1998

Vector Load Store Units

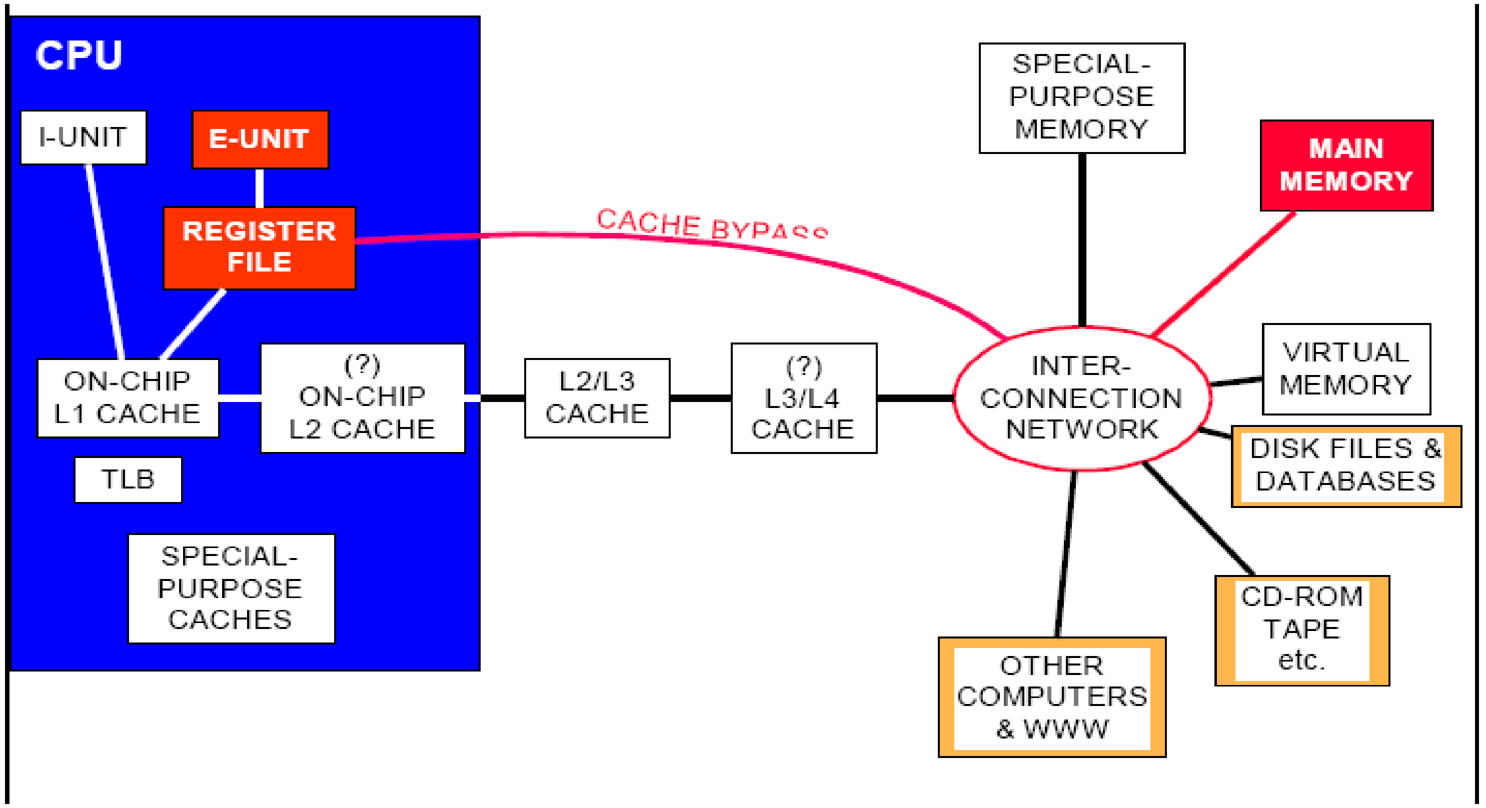
- Fully pipelined unit to load or store a vector; may have multiple LSUs.
- Uses the advantage of memory bank
  - support multiple loads/stores per cycle
  - multiple banks & address banks independently
  - support non-sequential accesses (see soon)
- Example

Memory architecture for Vector Processors

	Bank							
Cycle no.	0	1	2	3	4	5	6	7
0		136						
1		busy	144					
2		busy	busy	152				
3		busy	busy	busy	160			
4		busy	busy	busy	busy	168		
5		busy	busy	busy	busy	busy	176	
6			busy	busy	busy	busy	busy	184
7	192			busy	busy	busy	busy	busy
8	busy	200			busy	busy	busy	busy
9	busy	busy	208			busy	busy	busy
10	busy	busy	busy	216			busy	busy
11	busy	busy	busy	busy	224			busy
12	busy	busy	busy	busy	busy	232		
13		busy	busy	busy	busy	busy	240	
14			busy	busy	busy	busy	busy	248
15	256			busy	busy	busy	busy	busy
16	busy	264			busy	busy	busy	busy

1 fetch per cycle

Cache By Passing



- Do not depend upon cache.
- Scalar Processors have to depend on cache , hence occur cost while a cache-line miss occurs
- Good for Scientific applications

18-548/15-548 Memory System Architecture
Philip Koopman
November 4, 1998

Scalar Registers

- Typically Vector Processors have
  - 32 general purpose registers
  - 32 floating point registers
- Provide data as input to Vector Functional Units.

Example (daxpy)

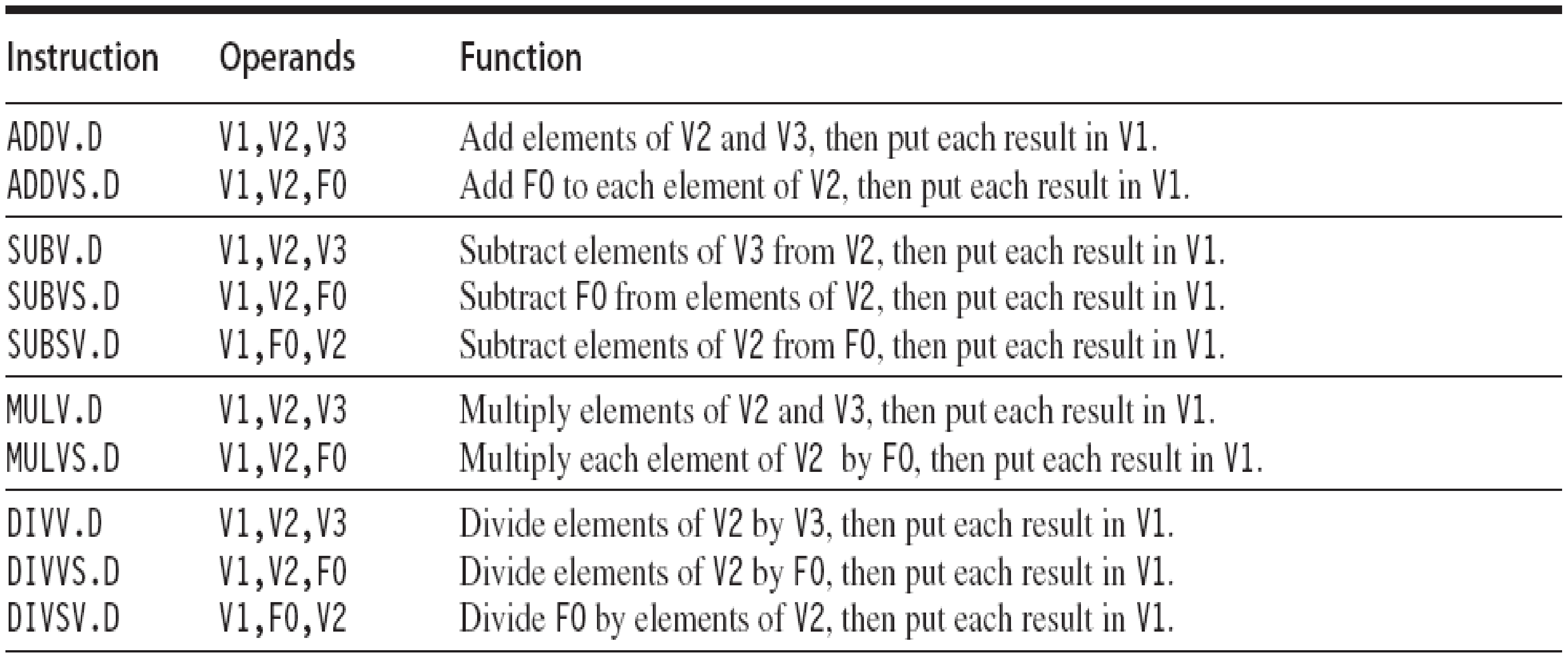
```
       L.D      F0,a          ;load scalar a
       DADDIU   R4,Rx,#512    ;last address to load
Loop:  L.D      F2,0(Rx)      ;load X(i)
       MUL.D    F2,F2,F0      ;a × X(i)
       L.D      F4,0(Ry)      ;load Y(i)
       ADD.D    F4,F4,F2      ;a × X(i) + Y(i)
       S.D      0(Ry),F4      ;store into Y(i)
       DADDIU   Rx,Rx,#8      ;increment index to X
       DADDIU   Ry,Ry,#8      ;increment index to Y
       DSUBU    R20,R4,Rx     ;compute bound
       BNEZ     R20,Loop      ;check if done
```

A Sample MIPS CODE

```
L.D       F0,a          ;load scalar a
LV        V1,Rx         ;load vector X
MULVS.D   V2,V1,F0      ;vector-scalar multiply
LV        V3,Ry         ;load vector Y
ADDV.D    V4,V2,V3      ;add
SV        Ry,V4         ;store the result
```

A Sample Code in VMIPS

Example Vector Instruction



Instruction	Operands	Function
ADDV.D	V1,V2,V3	Add elements of V2 and V3, then put each result in V1.
ADDVS.D	V1,V2,F0	Add F0 to each element of V2, then put each result in V1.
SUBV.D	V1,V2,V3	Subtract elements of V3 from V2, then put each result in V1.
SUBVS.D	V1,V2,F0	Subtract F0 from elements of V2, then put each result in V1.
SUBSV.D	V1,F0,V2	Subtract elements of V2 from F0, then put each result in V1.
MULV.D	V1,V2,V3	Multiply elements of V2 and V3, then put each result in V1.
MULVS.D	V1,V2,F0	Multiply each element of V2 by F0, then put each result in V1.
DIVV.D	V1,V2,V3	Divide elements of V2 by V3, then put each result in V1.
DIVVS.D	V1,V2,F0	Divide elements of V2 by F0, then put each result in V1.
DIVSV.D	V1,F0,V2	Divide F0 by elements of V2, then put each result in V1.

Properties of Vector Instructions

- Single Instruction implies lot of operations.
  - Hence reduce the number of instruction fetch and decode
- Each operation is independent of each other
  - Simple design
  - Multiple Operations can be run in parallel
- Data hazards has to be checked for each vector operation and not each operation
- Reduces Control hazards by reducing branches
- Knows memory access pattern

Vector Execution Time

- Time taken by each vector operation depends on – Vector Length, Data and Structural hazards
- Each operation has a startup time (pipelining latency)
- Startup time gets amortized as vector length tends to infinity. (One of the metrics for vector processors)

Convoy and Chime

- Convoy – A set of vector instruction that could potentially begin execution together in one clock period.
- Chime – unit of time to execute one convoy

```
LV       V1,Rx    ;load vector X
MULVS.D  V2,V1,F0 ;scaling vec.
LV       V3,Ry    ;load vector Y
ADDV.D   V4,V2,V3 ;add
SV       Ry,V4    ;store result
```

```
1. LV
2. MULVS.D LV
3. ADDV.D
4. SV
```

m-convoy take m-chimes (when startup time = 0)

4 convoy 4 chimes = (4 x 64 clock cycles)

4 clock cycle for 1 result

Startup overhead

Unit	Start-up overhead (cycles)
Load and store unit	12
Multiply unit	7
Add unit	6

Figure F.4 Start-up overhead.

Convoy	Starting time	First-result time	Last-result time
1. `LV`	0	12	$11 + n$
2. `MULVS.D LV`	$12 + n$	$12 + n + 12$	$23 + 2n$
3. `ADDV.D`	$24 + 2n$	$24 + 2n + 6$	$29 + 3n$
4. `SV`	$30 + 3n$	$30 + 3n + 12$	$41 + 4n$

Figure F.5 Starting times and first- and last-result times for convoys 1 through 4. The vector length is n.

$4 + (42/64) = 4.65$ clock cycles per result

Vector Length

- Consider a operation as shown below :

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

- Problem occurs when n is not equal length of the vector registers (64 in case of VMIPS)
- VLR – Vector Length Registers can be used when value of n is not known.

Strip mining

- Continuing the previous example. Problem may occur when size of ‘n’ > MVL (Maximum Vector Length)
- Strip mining generates code such that each vector operation is less than or equal to 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
```

Vector execution time with Strip mining

- Factors
  - Number of convoys in the loop = T_{chime}
  - Overhead for each strip-mined convoy = T_{loop} + T_{start}

T_{loop} = cost of executing the scalar code in loop.

T_{start} = vector startup cost.

$$T_n = \left\lceil \frac{n}{MVL} \right\rceil \times (T_{loop} + T_{start}) + n \times T_{chime}$$

Example

$$T_n = \left\lceil \frac{n}{MVL} \right\rceil \times (T_{loop} + T_{start}) + n \times T_{chime}$$

$$T_{200} = 4 \times (15 + T_{start}) + 200 \times 3$$

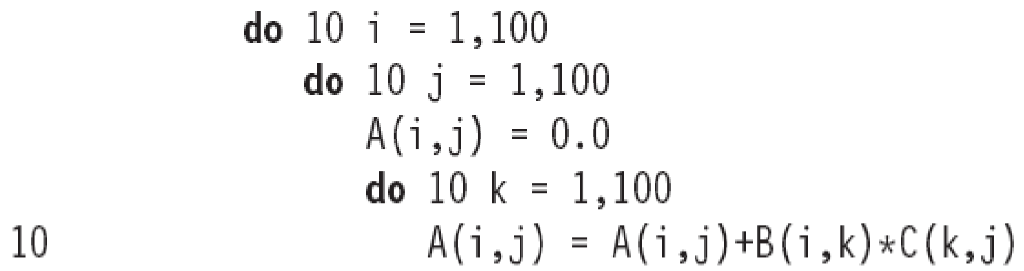
$$T_{200} = 60 + (4 \times T_{start}) + 600 = 660 + (4 \times T_{start})$$

```
       DADDUI   R2,R0,#1600  ;total # bytes in vector
       DADDU    R2,R2,Ra     ;address of the end of A vector
       DADDUI   R1,R0,#8     ;loads length of 1st segment
       MTC1     VLR,R1       ;load vector length in VLR
       DADDUI   R1,R0,#64    ;length in bytes of 1st segment
       DADDUI   R3,R0,#64    ;vector length of other segments
Loop:  LV       V1,Rb        ;load B
       MULVS.D  V2,V1,Fs     ;vector * scalar
       SV       Ra,V2        ;store A
       DADDU    Ra,Ra,R1     ;address of next segment of A
       DADDU    Rb,Rb,R1     ;address of next segment of B
       DADDUI   R1,R0,#512   ;load byte offset next segment
       MTC1     VLR,R3       ;set length to 64 elements
       DSUBU    R4,R2,Ra     ;at the end of A?
       BNEZ     R4,Loop      ;if not, go back
```

Number of Convoy = 3
Number of chimes = 3
n = 200
MVL = 64

Stride

- Consider a simple matrix multiplication program.



```
                  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)
```

- At each iteration we access the i th column of B and k th column on C.
- Stride = distance separating the elements that are to be merged into a single vector.

Stride (contd)

- Two types of addressing possible with Strides
  - Unit Stride
  - Non-Unit (constant) stride
- Example - LVWS V1, (R1,R2)

  R1 = base address , R2 = stride ,

  V1[i] = R1 + R2 X i
- One more mechanism for addressing is Indexed. (vector equivalent of register indirect)

Outline

- Enhancing vector performance
- Performance of vector processors
- Programming vector computers
  - Compiler vectorization
- Advantages
- Future of vector processors

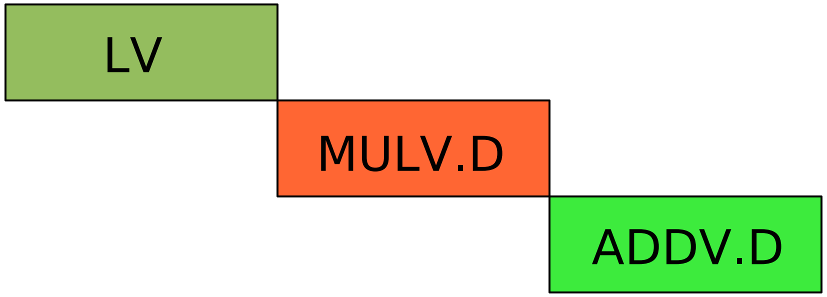
Enhancing vector performance

- Vector Chaining
- Conditionally Executed Statements
- Sparse Matrices
- Multiple Lanes
- Pipelined instruction Start-Up

Vector Chaining

- Forwarding extended to vector registers
- Eliminates data dependences by register bypassing

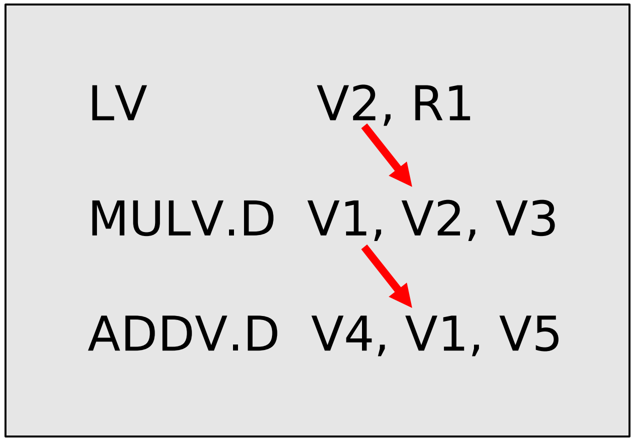
```
LV          V2, R1
MULV.D  V1, V2, V3
ADDV.D  V4, V1, V5
```



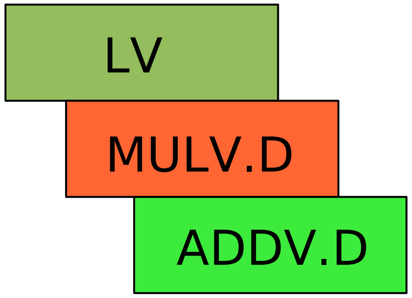
$VL*3 + LV_{startup} + ADDV_{startup} + MULV_{startup}$

Vector Chaining

- Forwarding extended to vector registers
- Eliminates data dependences by register bypassing



```
LV        V2, R1
MULV.D    V1, V2, V3
ADDV.D    V4, V1, V5
```



$VL + LV_{startup} + ADDV_{startup} + MULV_{startup}$

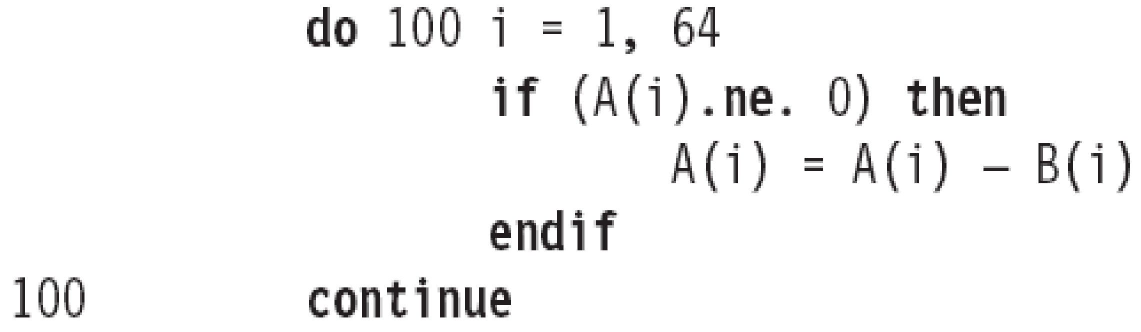
Vector Chaining

- Flexible chaining
  - Can chain any two vector instructions, if there is no structural hazard
  - Simultaneous access to the same vector register
- Reduces the number of chimes (How?)
- No convoy can contain a structural hazard

Conditionally Executed Statements

- Inhibitors for effective vectorization
  - Presence of conditionals
  - Use of sparse matrices
- Branch statements introduce control dependences

Can this loop be vectorized?



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

Vector-mask register

- Boolean vector of length MVL to control the execution of a vector instruction
- Vector instructions operate only on elements defined by VM
- Vector-mask can be set or unset based on the result of a condition
- Disadvantages
  - Instructions can take time even if their mask entry is set to 0
  - Some processors just disable the stores, but execute the actual instruction

Vector-mask example

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

```
LV          V1,Ra        ;load vector A into V1
LV          V2,Rb        ;load vector B
L.D         F0,#0        ;load FP zero into F0
SNEVS.D     V1,F0        ;sets VM(i) to 1 if V1(i)!=F0
SUBV.D      V1,V1,V2     ;subtract under vector mask
CVM                      ;set the vector mask to all 1s
SV          Ra,V1        ;store the result in A
```

Sparse Matrices

- Vector elements stored compactly, Indirect accesses
- Indexed Load (Gather) / Indexed Store (Scatter)
- Sparse vector sum

on arrays A & C

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

K and M are index vectors

```
LV        Vk,Rk          ;load K
LVI       Va,(Ra+Vk)     ;load A(K(I))
LV        Vm,Rm          ;load M
LVI       Vc,(Rc+Vm)     ;load C(M(I))
ADDV.D    Va,Va,Vc       ;add them
SVI       (Ra+Vk),Va     ;store A(K(I))
```

Scatter Gather example

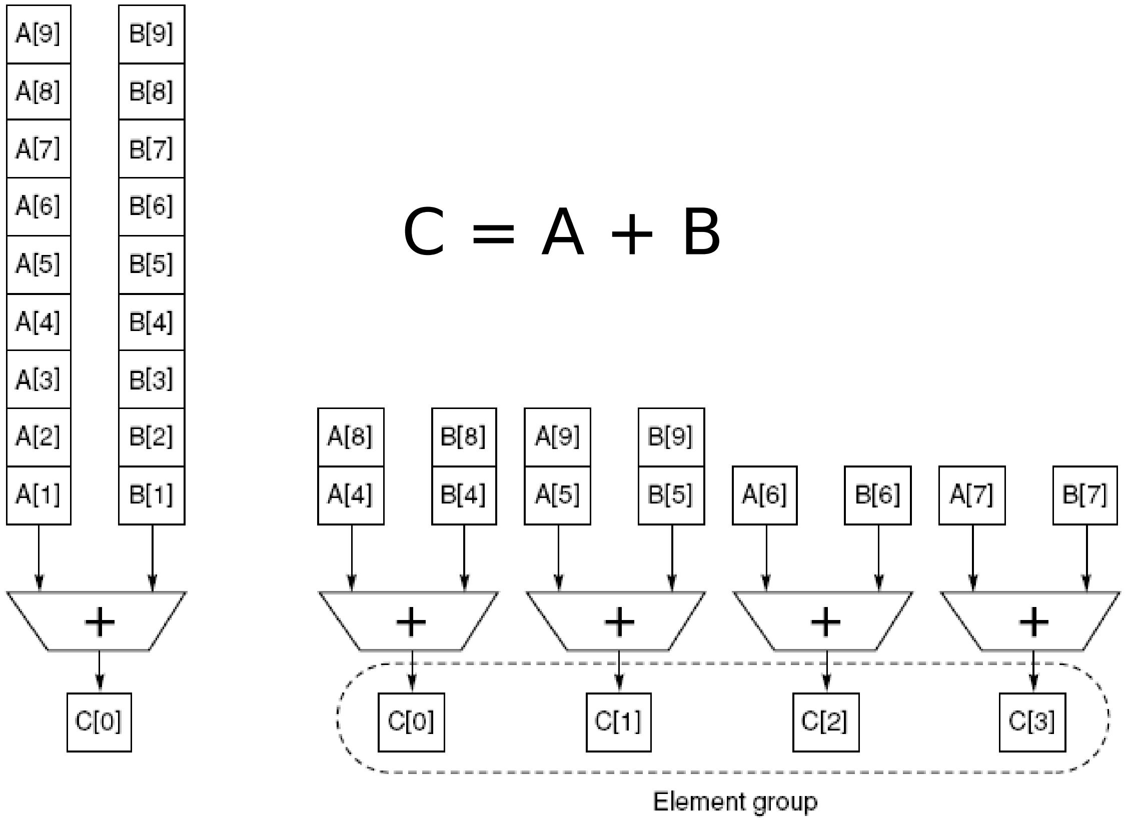
```
            do 100 i = 1, 64
                    if (A(i).ne. 0) then
                            A(i) = A(i) – B(i)
                    endif
100         continue
```

```
 LV          V1,Ra         ;load vector A into V1
 L.D         F0,#0         ;load FP zero into F0
 SNEVS.D     V1,F0         ;sets the VM to 1 if V1(i)!=F0
 CVI         V2,#8         ;generates indices in V2
 POP         R1,VM         ;find the number of 1's in VM
 MTC1        VLR,R1        ;load vector-length register
 CVM                       ;clears the mask
 LVI         V3,(Ra+V2)    ;load the nonzero A elements
 LVI         V4,(Rb+V2)    ;load corresponding B elements
 SUBV.D      V3,V3,V4      ;do the subtract
 SVI         (Ra+V2),V3    ;store A back
```

Multiple Lanes

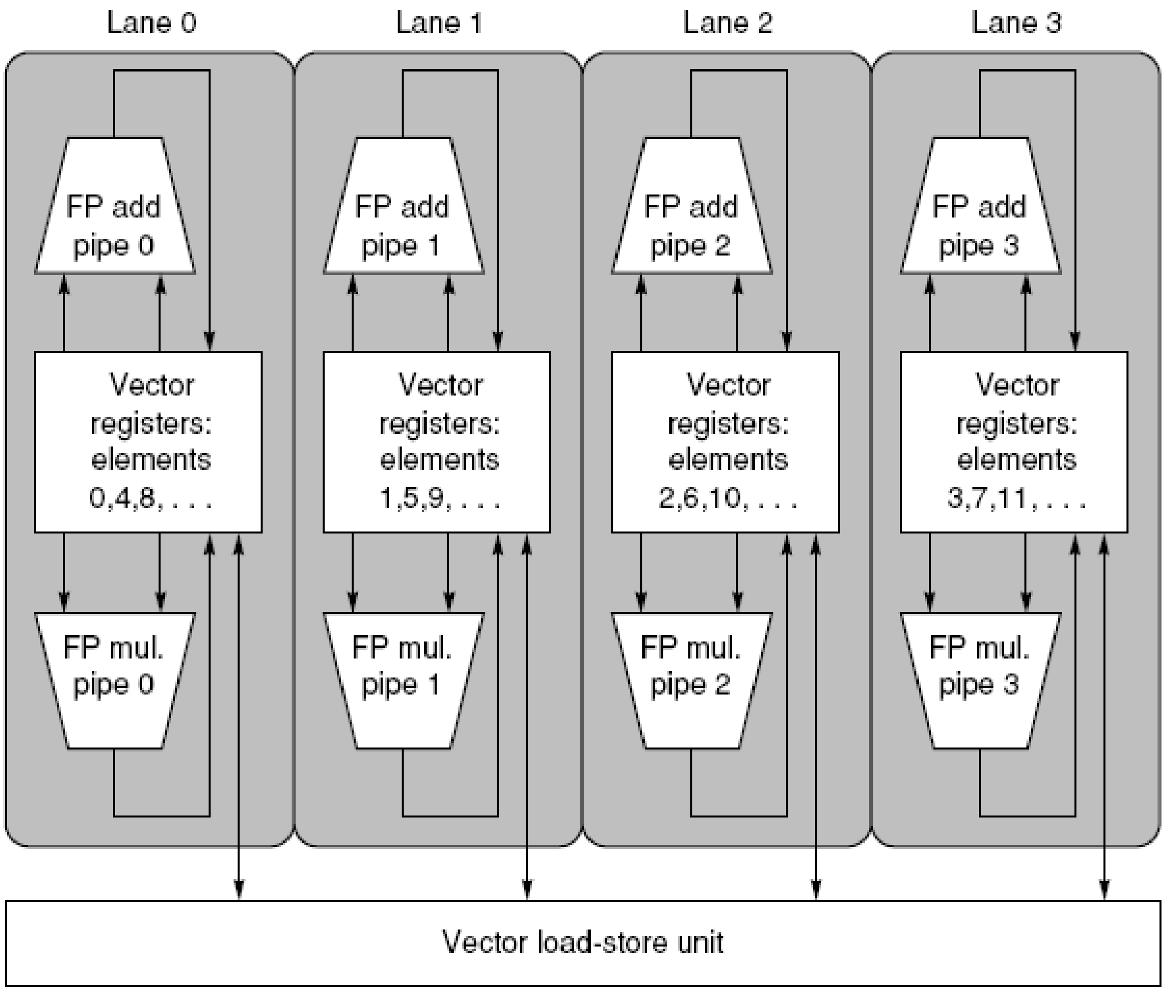
- Lanes are multiple parallel pipelines
- Reduce structural hazards

C = A + B



Single add pipeline Four add pipelines

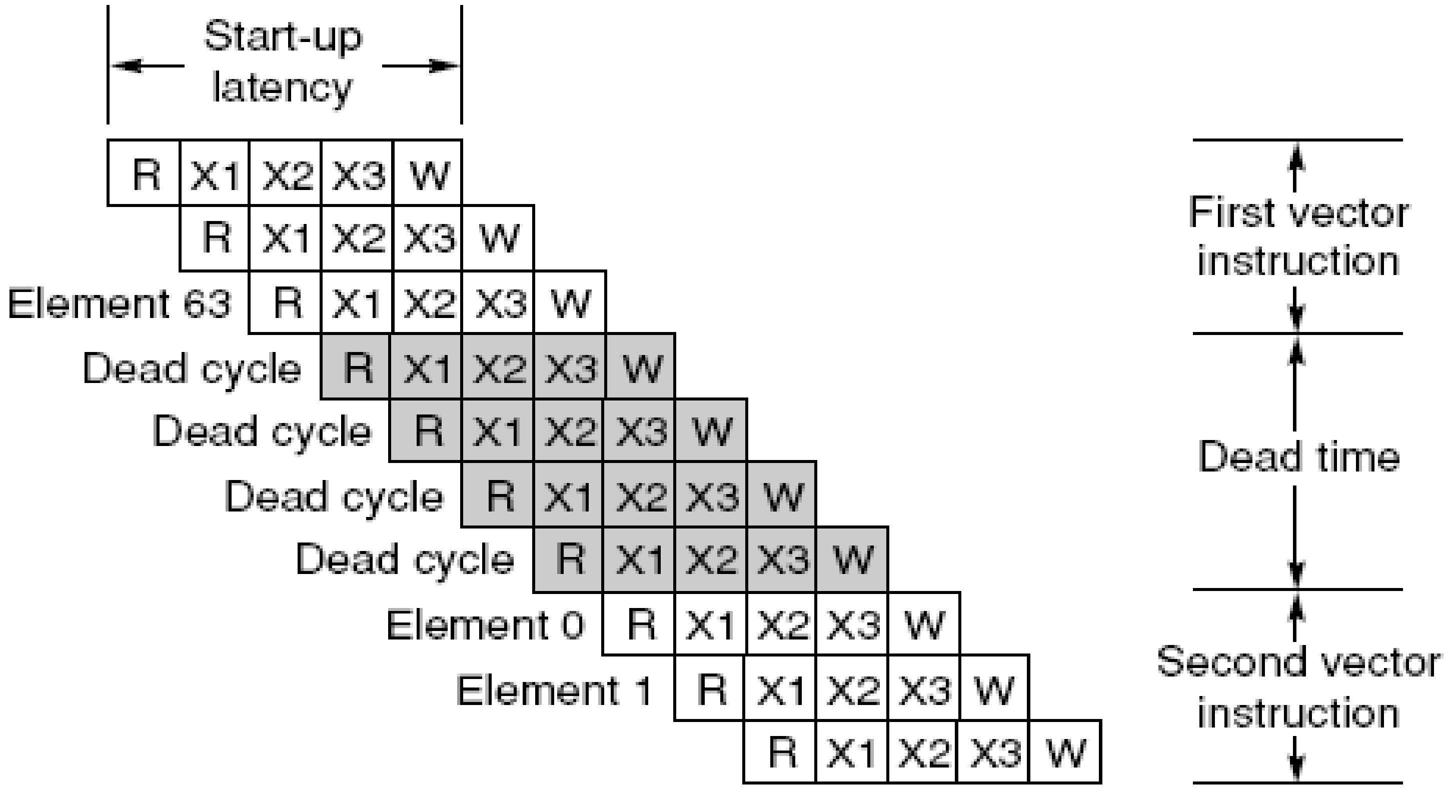
Vector unit with four lanes


Lane 0
Lane 1
Lane 2
Lane 3
FP add pipe 0
FP add pipe 1
FP add pipe 2
FP add pipe 3
Vector registers: elements 0,4,8, . . .
Vector registers: elements 1,5,9, . . .
Vector registers: elements 2,6,10, . . .
Vector registers: elements 3,7,11, . . .
FP mul. pipe 0
FP mul. pipe 1
FP mul. pipe 2
FP mul. pipe 3
Vector load-store unit

Pipelined Instruction Start-Up

- Chaining does not eliminate the start-up time for a vector instruction
- Some recovery time (dead time) required between two vector instructions
- Allow start of one instruction to overlap with the completion of the preceding instruction

Pipelined Instruction Start-Up



Start-up latency and dead time for a single vector pipeline

Performance measurements

- Measure the execution time of a vector loop
- Consider the start-up cost and the sustained rate of the operation
- Behavior of vector pipelines and instructions characterized by
  - R_∞ MLOPS rate on an infinite-length vector
  - $N_{1/2}$ Vector length needed to reach half of R_∞

R-infinity n-half vector model

- For a vector loop with n elements:

$$T_n = \left\lceil \frac{n}{MVL} \right\rceil \times (T_{loop} + T_{start}) + n \times T_{chime}$$

- Calculate asymptotic performance R∞

$$R_\infty = \lim_{n \to \infty} \left(\frac{\text{Operations per iteration} \times \text{Clock rate}}{\text{Clock cycles per iteration}} \right)$$

- Compute N½ to find performance r

$$r = \frac{qn}{T} = \frac{n}{t} = \frac{r_\infty}{(1 + n_{\frac{1}{2}}/n)}$$

R-infinity n-half vector model

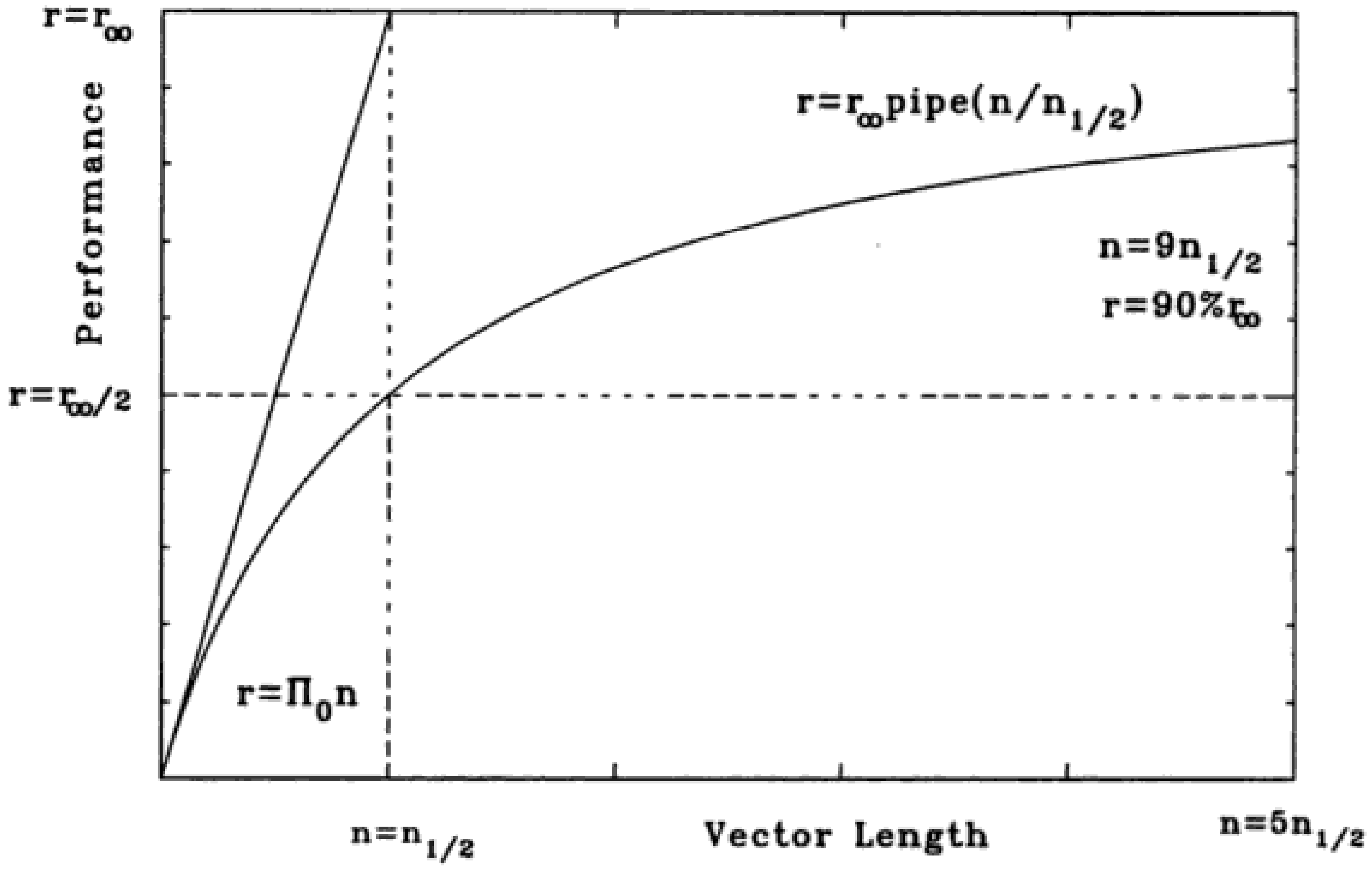


Figure 3.2: Performance against vector (or Fortran DO-loop) length for a pipelined arithmetic unit, showing the geometric definition of the $(r_\infty, n_{\frac{1}{2}})$ parameters.

Source: The Science of Computer Benchmarking By Roger W. Hockney

Programming vector computers

- Two main approaches
  - Writing data-parallel programs in native languages
    - UPC, X10, Chapel
  - Relying on compilers
    - Automatic
    - Hinted

Compiler Vectorization

- Why compiler cannot vectorize loops?
  - Branches
  - Recurrences
  - System calls
  - Subscript ambiguities
- Techniques adopted
  - Force maximum work in inner loop
  - Eliminate false dependences
  - Use vectorization directives

Compiler Vectorization

- Vectorization Directives
  - To aid the compiler in vectorizing a particular section of code

```
!DEC$ VECTOR ALWAYS
do i = 1, 100, 2
a(i) = b(i)
enddo
```

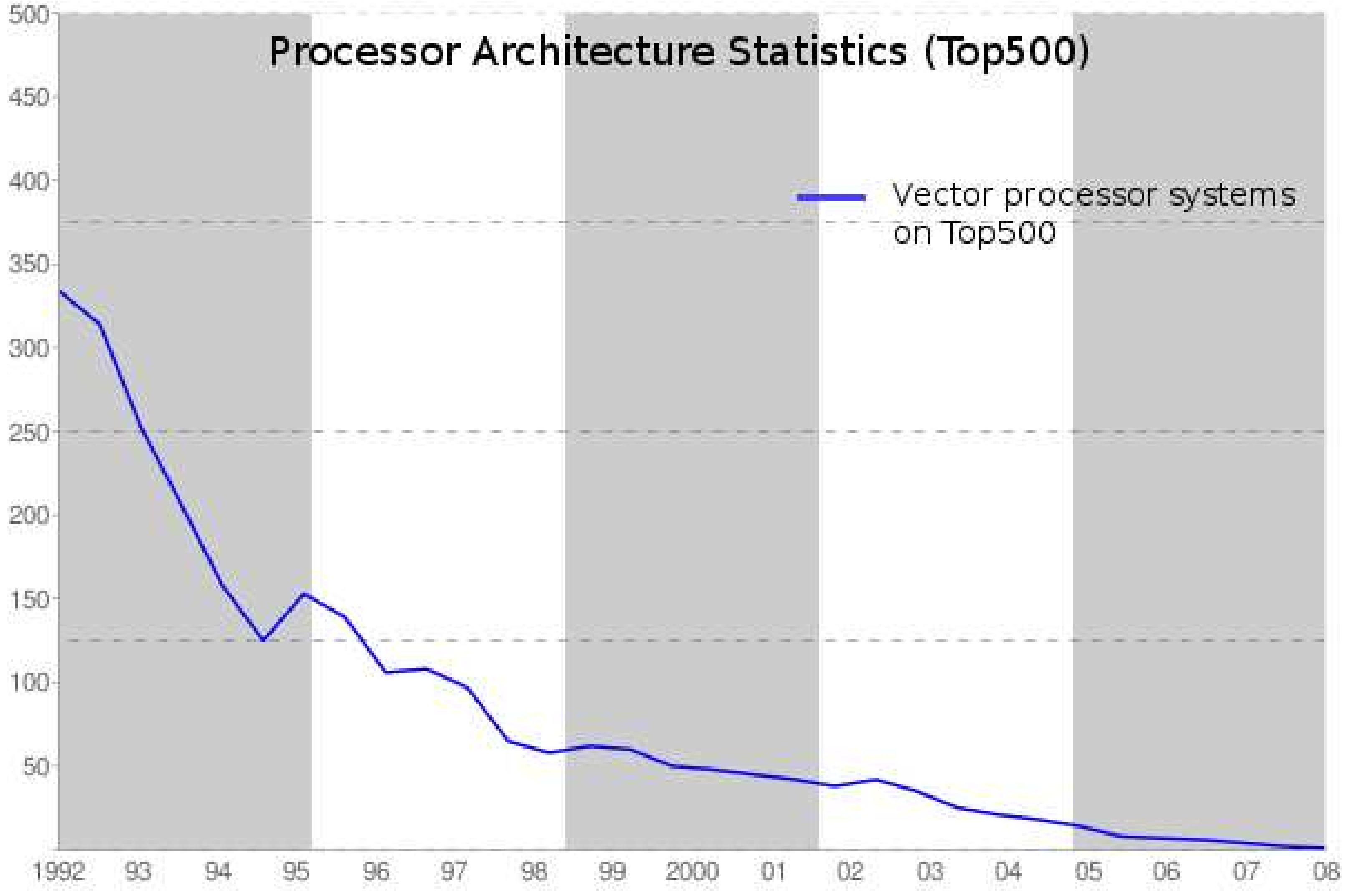
```
!DEC$ NOVECTOR
do i = 1, 100
a(i) = b(i) + c(i)
enddo
```

- Autovectorization
  - Dependence analysis
  - Finding sufficient parallelism
  - Loop unrolling, fusion/jamming

Advantages

- Vector supercomputers offer greater inherent parallelism versus the limited issue of superscalar microprocessors
- Very high memory bandwidths
- Smaller program size reduces complexity
- Low power consumption
- Work exceptionally well for a certain set of scientific applications

Decline of vector processors?



Questions?