The Best Way to Achieve Vector-Like Performance?

J. E. Smith

April 20, 1994

CRAY RESEARCH, INC.

Remember?

the SEMANTIC GAP

CRAY RESEARCH, INC.

The Semantic Gap

Premise: The semantic gap between HLL and machine languages should be closed via complex instructions.

Consider:

```
      Do 10 i=1,looplength
         a(i) = b(i) * x + c(i)
10    continue
```

CISC compilation:

```
      R1     <- looplength
      R2     <- 1
      R3     <- x
loop: a(R2)  <-  b(R2) * R3 + c(R2)
      if R2++ < R1 goto loop
```

The inner loop is just two instructions.

A problem: instructions have dependences coded in.

RISC/superscalar

Premise: Break program into elemental pieces and schedule to maximize independence.

(unroll twice)

```
        A1        <- looplength  .assume even
        S2        <- x           .load constant
        A4        <- addr(b)     .load
        A5        <- addr(c)     . address
        A6        <- addr(a)     .  pointers
loop:   S3        <- mem(A4)     .load b(i)
        S5        <- mem(A4 + 8) .load b(i+1)
        S4        <- mem(A5)     .load c(i)
        S7        <- S3 * S2     .b(i)*x
        S6        <- mem(A5 + 8) .load c(i+1)
        S8        <- S5 * S2     .b(i+1)*x
        A4        <- A4 + 16     .bump b pointer
        S9        <- S7 + S4     .c(i)+b(i)*x
        A5        <- A5 + 16     .bump c pointer
        S10       <- S8 + S6     .c(i+1)+b(i+1)*x
        A1        <- A1 - 2      .decrement counter
        mem(A6)   <- S9          .store a(i)
        mem(A6+8)<- S10          .store a(i+1)
        A6        <- A6 + 16     .bump a pointer
        BNEZ       A1,loop       .branch on A1 NEZ
```

RISC/superscalar (contd.)

A problem: at runtime, the hardware must still scan the instruction stream to find dependences (in data and resources).

- The wider the superscalar, the more difficult the problem becomes.

- Dependences are found at compile time, but thrown away after scheduling.

Another problem: data locality is hidden.
e.g. the address stream is:

b(1), b(2), c(1), c(2), a(1), a(2), b(3), b(4), c(3), c(4), a(3), a(4), etc.

So, just when you thought it was safe...

the

PARALLELISM GAP

The Parallelism Gap

Premise: The parallelism gap between HLL and the hardware can be closed by having an architecture that conveys instruction level parallelism to the hardware.

Example: VLIW

The compiler places independent instructions into long instruction words.

(preamble and postamble omitted)

```
loop:   S3  <- mem(A4)       | S37 <- S33 * S2 | A6 <- A6+32
        S5  <- mem(A5)       | S29 <- S27 + S25|
        mem(A6) <- S19       |                 |
        S13 <- mem(A4+8)     | S7  <- S3  * S2 |
        S15 <- mem(A5+8)     | S39 <- S37 + S35|
        mem(A6+8) <- S29     |                 |
        S23 <- mem(A4+16)    | S17 <- S13 * S2 |
        S25 <- mem(A5+16)    | S9  <- S7  + S5 |
        mem(A6+16) <- S39    |                 |
        S33 <- mem(A4+24)    | S27 <- S23  * S2| A4 <- A4+32
        S35 <- mem(A5+24)    | S19 <- S17 + S15| A5 <- A5+32
        mem(A6+24) <- S9     |                 | BNEZ  A1, loop
```

VLIW bridges parallelism gap to some extent...

Instructions packed in a LIW are independent, but there is no additional info about other dependences.

Also the locality problem is not dealt with any better than superscalar RISC.

Vectors

Parallelism is conveyed to hardware via vector instructions.

With vector registers, large numbers of dependences can be resolved at once.

(stripmine code not shown)

```
A2  <- address(a)   .load
A3  <- address(b)   . array
A4  <- address(c)   .  pointers
S1  <- x            .scalar x in register S1
V1  <- A3           .load b
V3  <- A4           .load c
V2  <- V1 * S1      .vector * scalar
V4  <- V2 + V3      .add in c
A2  <- V4           .store to a
```

This "CISC" does not close a semantic gap;
it closes a parallelism gap.

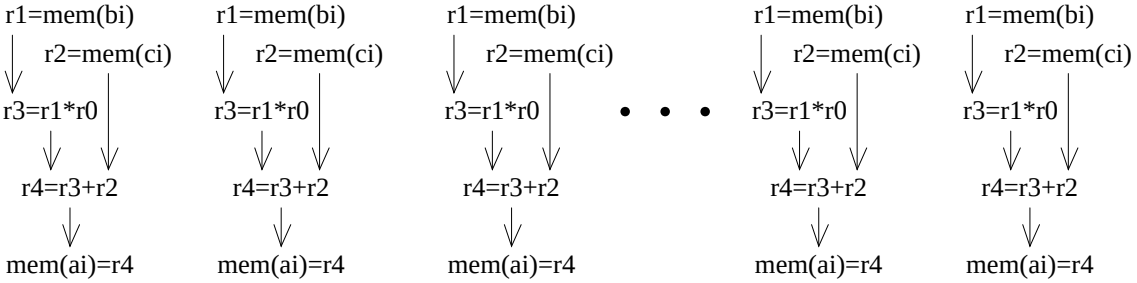
Also, data locality info is passed to hardware;

address stream:

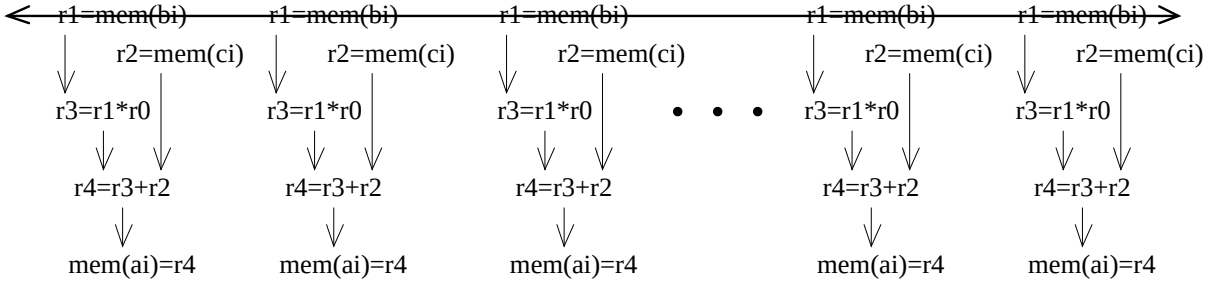
a(1),a(2),a(3),a(4),...b(1),b(2),b(3),....c(1),c(2),c(3),...

Passing ILP to Hardware

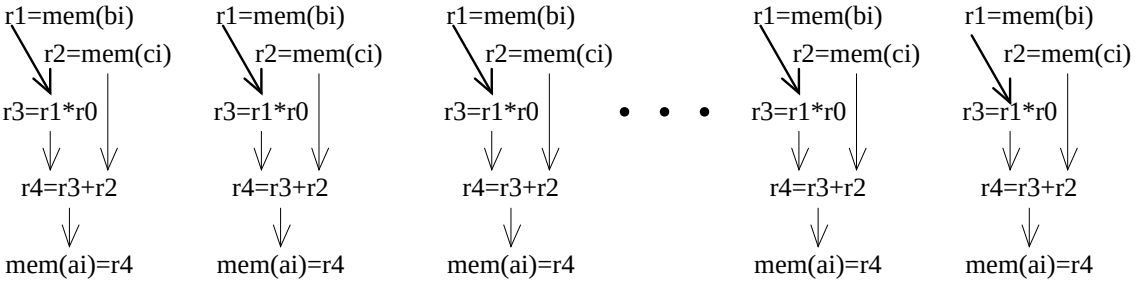
Large blocks of both data independence and dependence can be passed to hardware.



Block of independent operations in one vector inst.



Block of data dependences between two vector instructions



CRAY RESEARCH, INC.

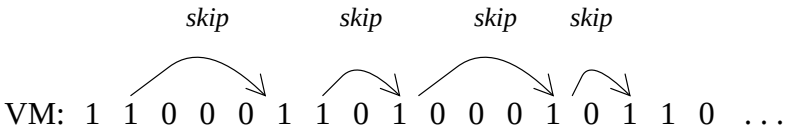
Passing ILP to Hardware

Consider an example of control parallelism:

```
        Do 20 i=1,looplength
           if (a(i).eq.b(i)) then
              c(i) = a(i) + e(i)
           endif
   20   continue
```

```
V1  <- a          .load a
V2  <- b          .load b
VM  <- V1==V2     .compare
V3  <- e;VM       .load e under mask
V4  <- V1+V3; VM  .add under mask
c   <- V4; VM     .store under mask
```

- With vectors, hardware sees the "big picture":



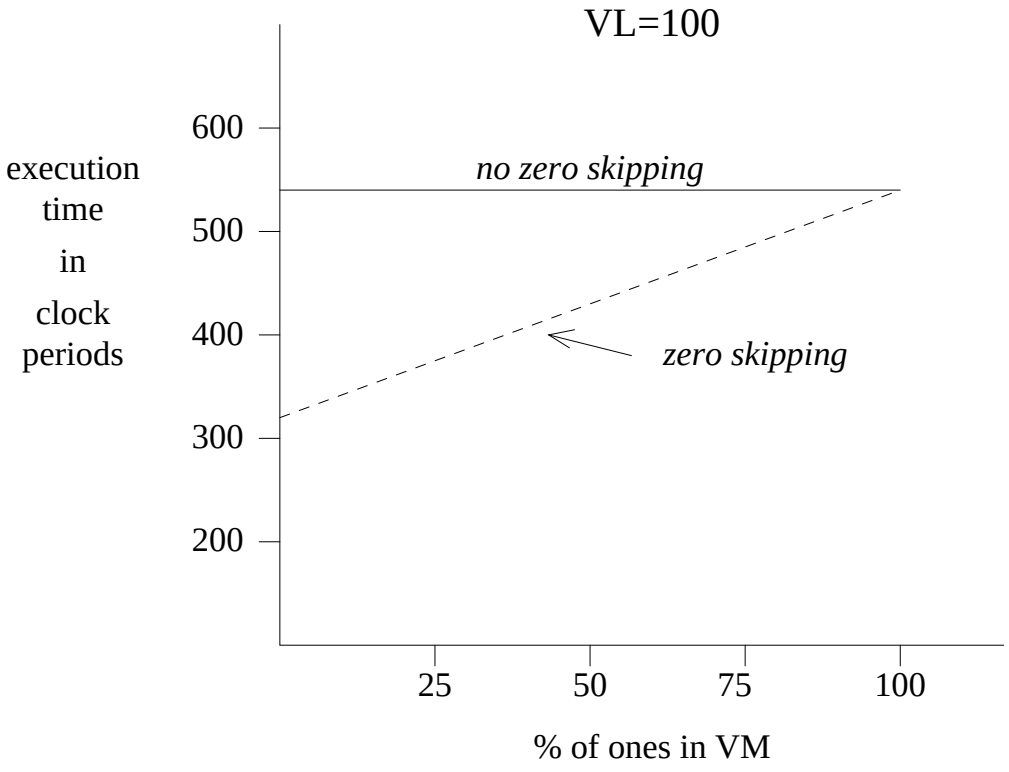
Consecutive zeros can be skipped
=> only useful operations are performed.

- Contrast with conditional moves: excess operations, safety problems
- Contrast with "guarded" scalar ops: consume issue bandwidth

CRAY RESEARCH, INC.

"Density-Time" Masked Operations

```
      Do 20 i=1,looplength
         if (a(i).eq.b(i)) then
            c(i) = a(i) + e(i)
         endif
20    continue
```



CRAY RESEARCH, INC.

Passing Address Patterns to Hardware

A simple example has already been given.

Also consider:

```
      Do 20 i = 1,length1
        Do 20 j=1,length2
          c(i,j) = a(j,i) + e(i,j)
   20   continue
```

```
A1  <- length1      .setup stride
V1  <- e,A1         .load with stride
```

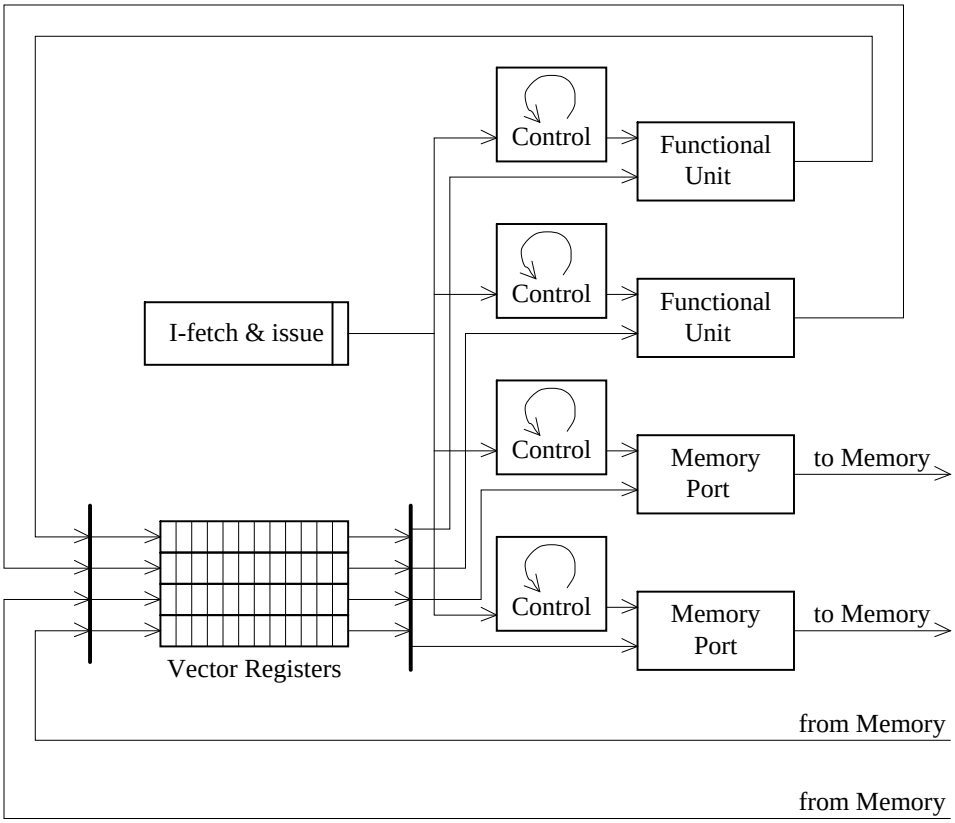
AND

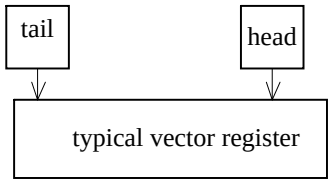
```
      Do 20 i = 1,length1
          c(i) = a(index(i)) + e(i)
   20   continue
```

```
V1  <- index,1      .load index vector
V2  <- a,V1         .gather with index vector
```

Modular, highly parallel control structure

Yet simple instruction issue logic.





Counter-based control allows very fast clock
(e.g. 2X normal scalar clock)

CRAY RESEARCH, INC.

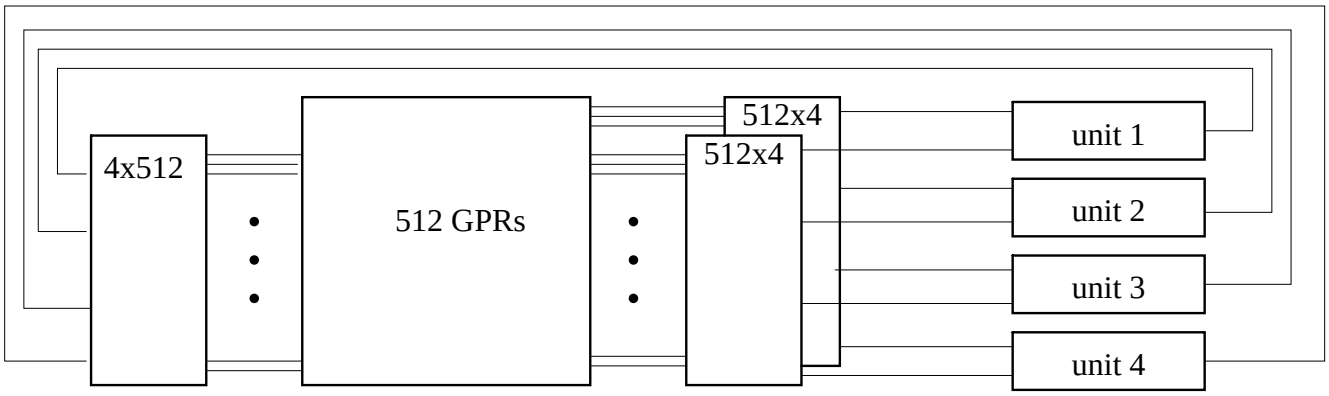
Simple support for a very large register space.

- Lots of registers: 1024 in CRI C90
- Simple reservation logic
  2 reservations per vector register (1 read port, 1 write port)
- Two-level addressing
  Vector register + element in register
- Simple port structure

Example: 512 GPRs vs 8x64 Vregs

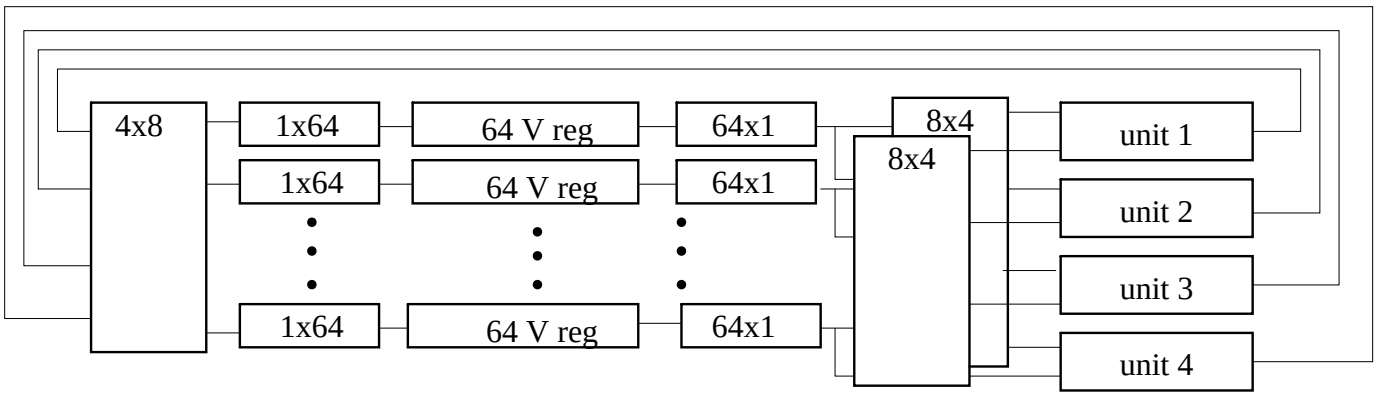
- Consider logical crosspoints required.

-with 512 general purpose registers:



2048 x-points 4096 x-points =total 6144 x-points

-with 8 64-element vector registers:



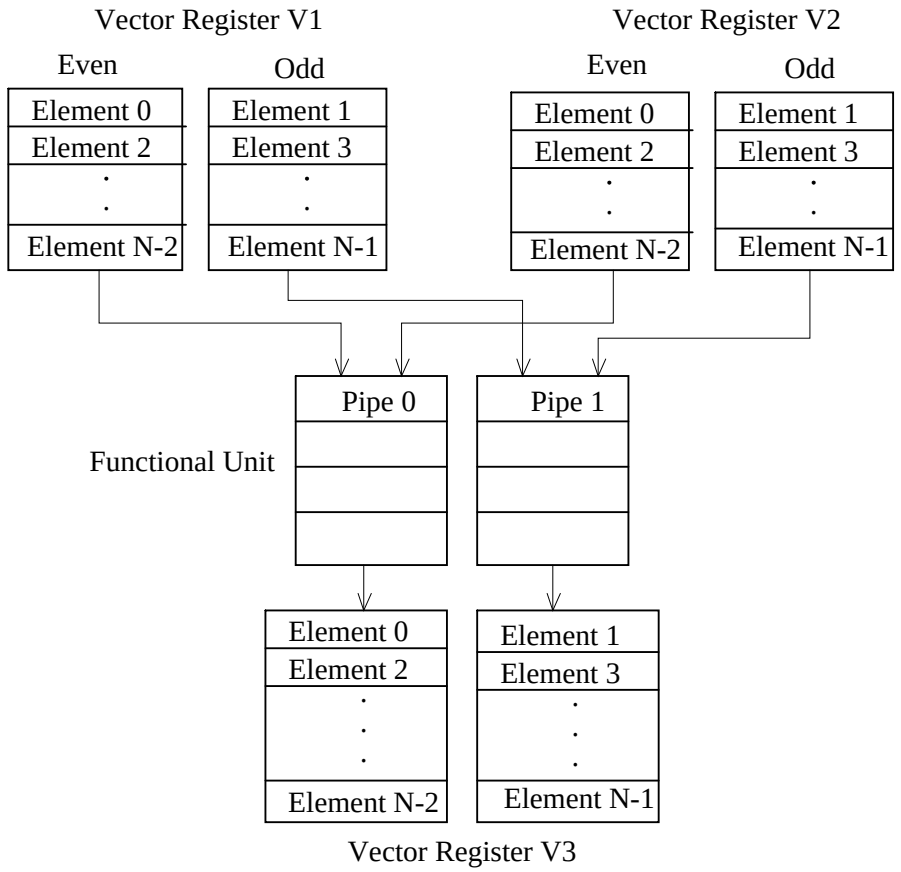
32 x-points 512 x-points 512 x-points 64 x-points = total 1120 x-points

- A factor of 5 fewer logical crosspoints.

CRAY RESEARCH, INC.

ILP can be simply increased

- Replicate pipes in the implementation

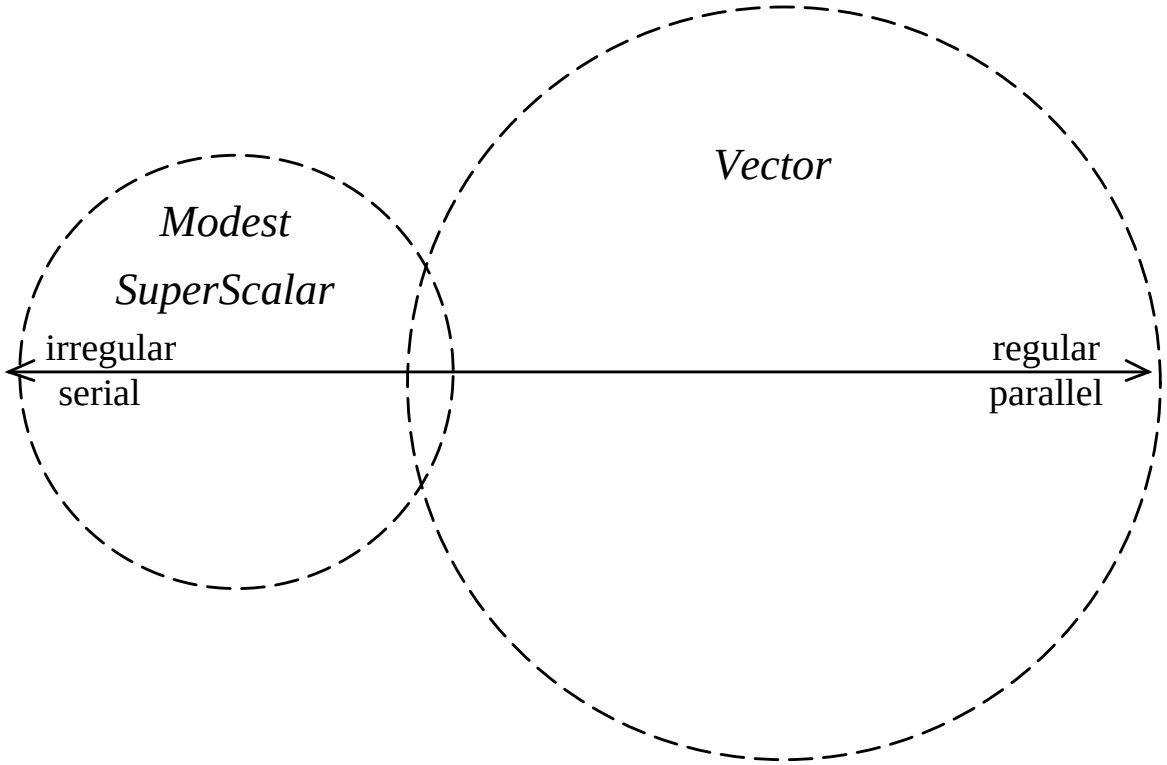


- Instruction issue logic is unchanged.
- Register crosspoints grow linearly.
- A multi-pipe vector implementation can sustain the equivalent of 20 or more instruction issues per clock.

CRAY RESEARCH, INC.

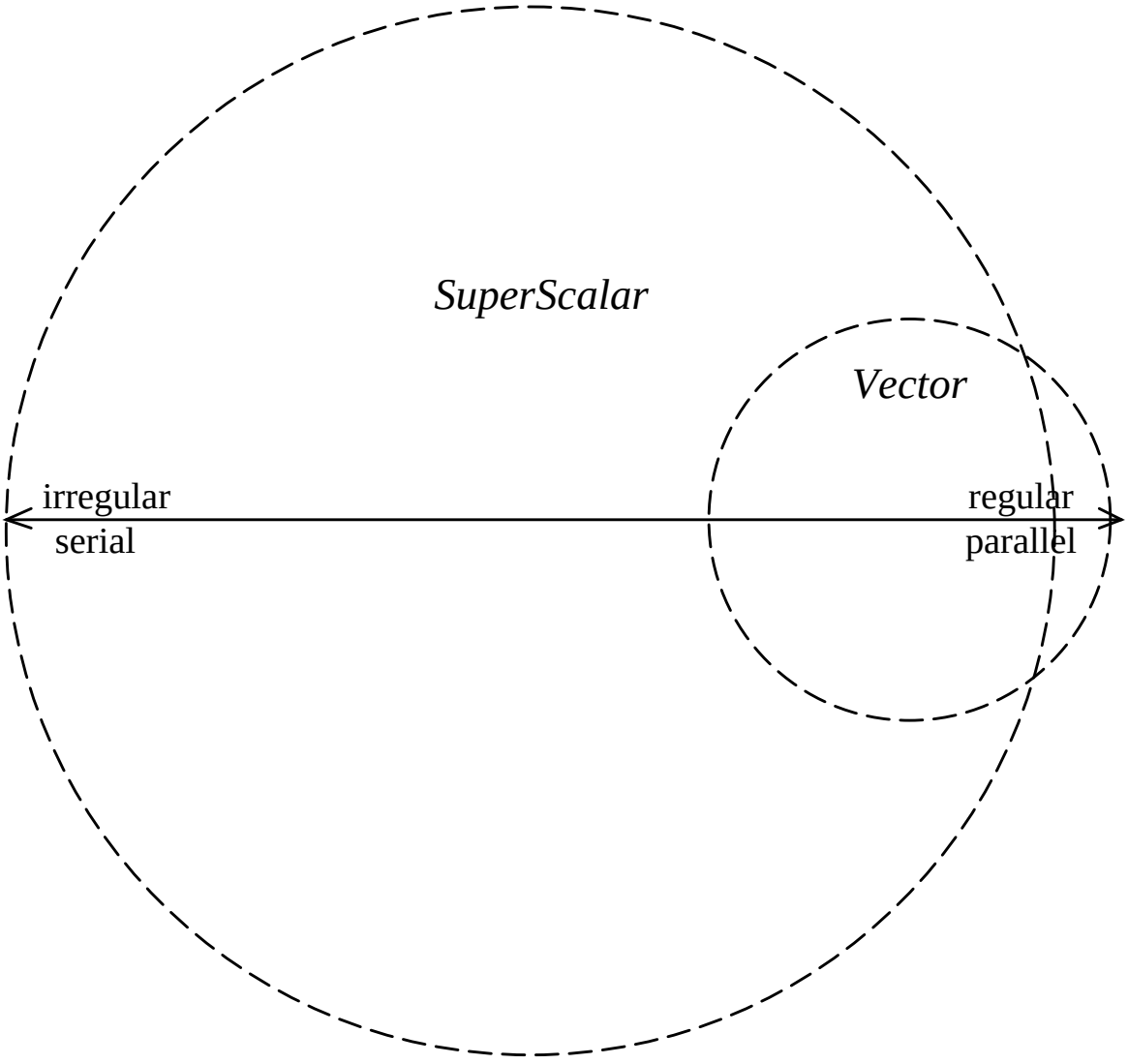
How extensive is vector parallelism?

View from Vector Advocate's Perspective



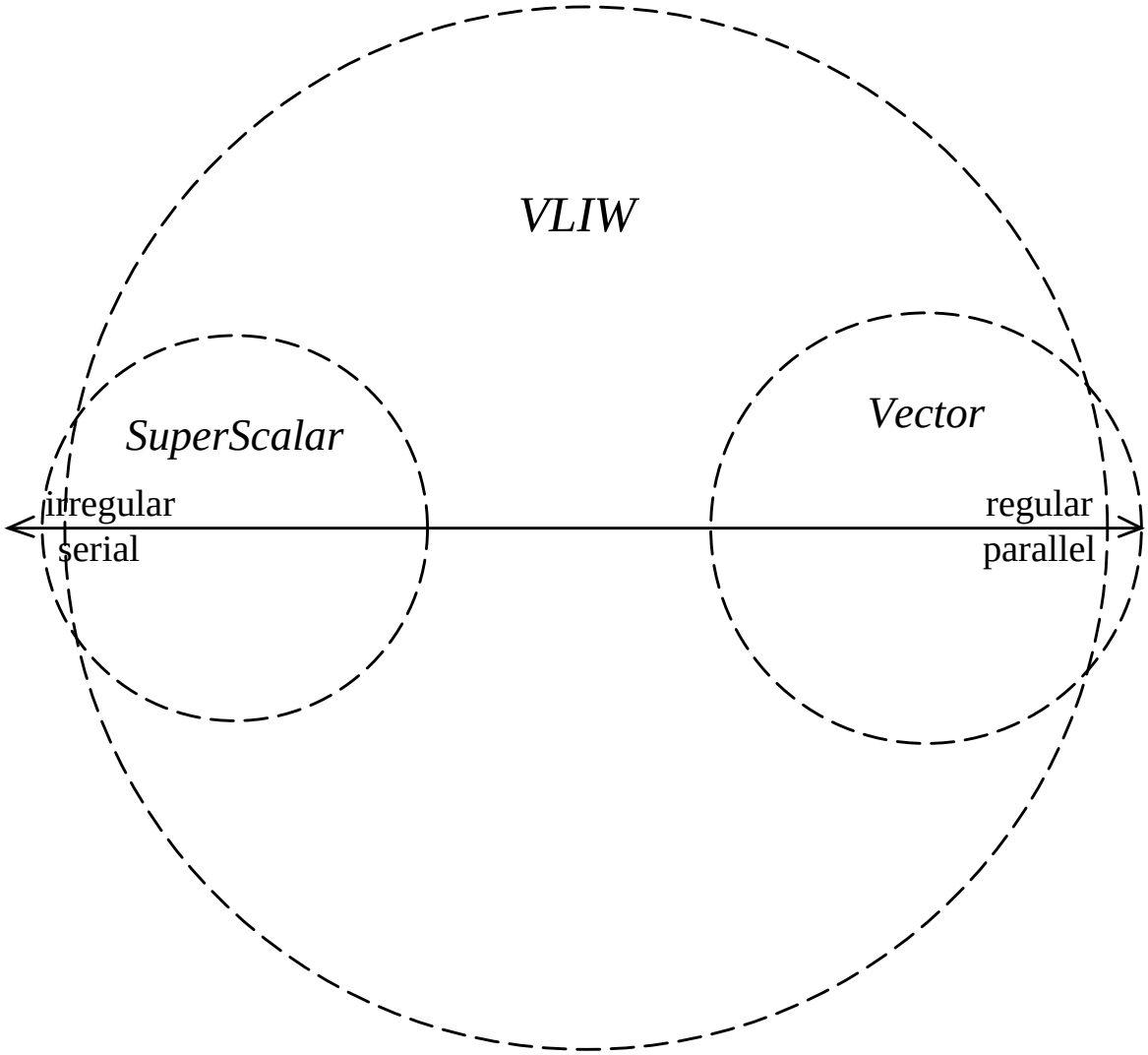
CRAY RESEARCH, INC.

View from Superscalar Advocate's Perspective



CRAY RESEARCH, INC.

View from VLIW Advocate's Perspective



CRAY RESEARCH, INC.

Flexibility: Nested parallelism

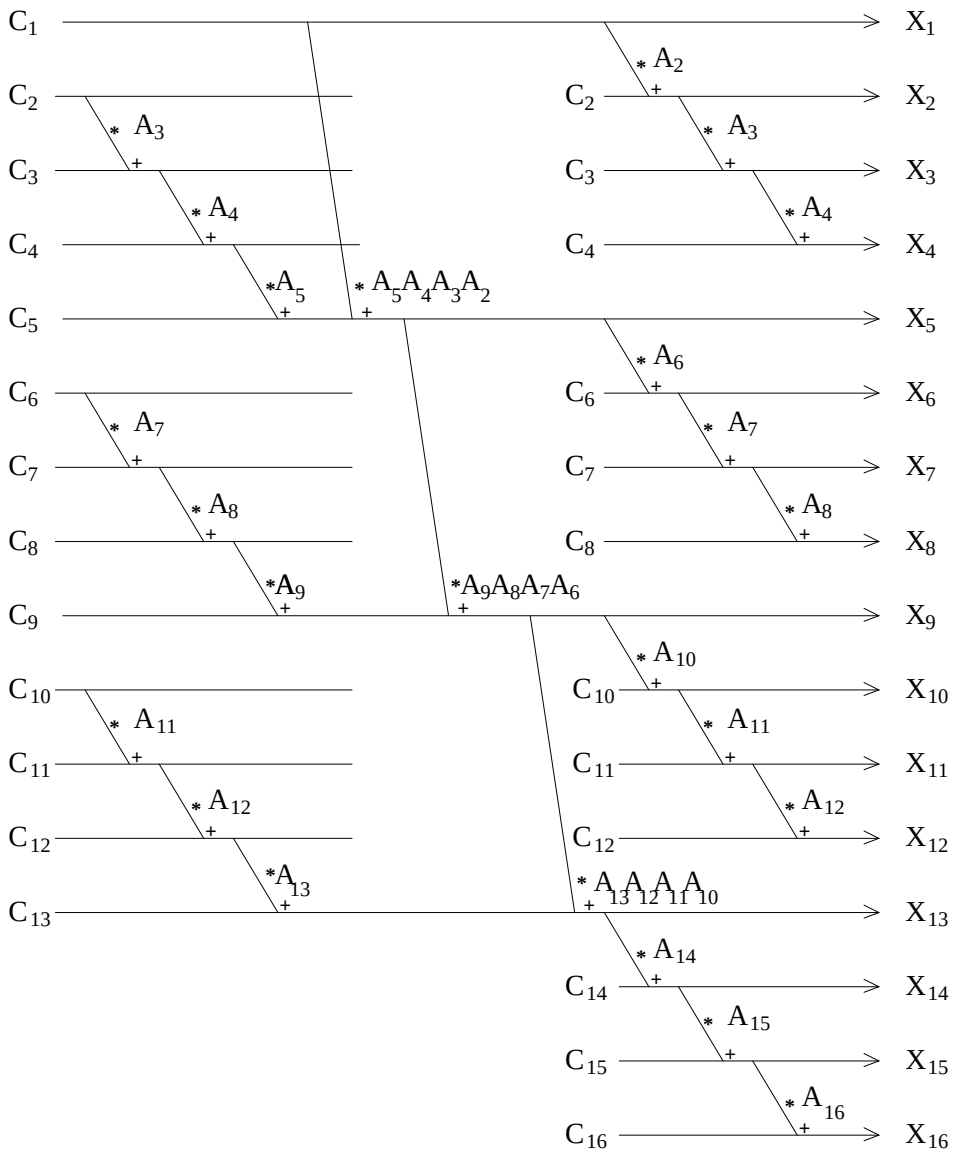
```
      do 30 i = 1,looplen
        if(b(i).ne.0) then
          if(c(i).eq.0) then
            d(i) = a(i) + b(i)
          endif
          c(i) = d(i) + b(i)
        endif
30    continue
```

(stripmine code not shown)

```
V1      <- B(i)            .load b(i)
VM1     <- V1 != 0         .test b != 0
V2      <- C(i); VM1       .load c(i) under mask
VM2     <- V2 == 0; VM1    .test c == 0 under mask
V3      <- A(i); VM2       .load a under mask
V4      <- V1 + V3; VM2    .add under mask
D(i)    <- V4; VM2         .store d under mask
VM3     <- VM1 & !VM2      .for d elements in memory
V4      <- D(i); VM3       .load new d(i) under mask
V5      <- V4 + V1; VM1    .add under mask
C(i)    <- V5; VM1         .store under mask
```

Flexibility: 1st order linear recurrences

Example: X(I) = A(I) * X(I-1) + C(I)



CRAY RESEARCH, INC.

Flexibility: Speculative Execution

- Techniques usually ascribed to superscalar of VLIW may also be applied to vector architectures.

```
      x = a(1)
      do 410 i = 1,looplen
       b(i) = x + c(i)
       if (b(i).eq.0) x = a(i+1)
410   continue
```

If b(i) is always 0, then inner loop:

```
      b(i) = a(i) + c(i)
```

If b(i) is never 0, then inner loop:

```
      b(i) = x + c(i)
```

- VL is profile-dependent

- i.e. one might make the VL artificially short to avoid **speculative overshoot ™**

  I.e. if branch is 90 percent taken, use VL = 10

  If 99 percent taken, use VL = 100

Speculative Vector Execution

Assume mostly true:

```
    A4  <- looplen      .loop iterations
    A5  <- maxvl        .a profile-dependent VL
    VL  <- min(A4,A5)   .initialize VL
    A1  <- 0            .initialize offset into arrays
loop:V1<- a,A1          .load a(i), offset by A1
    V3  <- c,A1         .load c(i), offset by A1
    V4  <- V1 + V3      .add to get b(i)
    VM  <- V4 == 0      .test b(i)
    A2  <- LZ(VM)       .find leading zero count
    A3  <- A2 - 1       .backup to first false
    Br  next;A4 == VL   .branch if all were true
    S1  <- V1(A3)       .get previous x from A vector
    S4  <- V3(A2)       .get c element.
    S5  <- S1 + S4      .add
    V4,A2  <- S5        .put into b array
    A3  <- A2           .will be new VL
next:VL<- A3            .VL is up to first false
    b,A1<- V4           .store into b
    A1  <- A1 + A2      .adjust base address of A array
    A4  <- A4 - A2      .decrement remaining iterations
    Br  done; A4 <= 0   .quit if all iterations done
    VL  <- min(A4,A5)   .readjust VL
    Br  loop
done:
```

Speculative Vector Execution

Assume mostly false:

```
    A4  <- looplen      .number of times to execute loop
    A5  <- maxvl        .a profile-dependent value for VL
    VL  <- min(A4,A5)   .initialize VL
    S1  <- a(1)         .initialize x
    A1  <- 0            .initialize offset into arrays
loop:V3 <- c,A1         .load c(i) offset by A1
    V4  <- S1 + V3      .add to get b(i)
    VM  <- V4 == 0      .test b(i)
    A2  <- LO(VM)       .leading one
    A3  <- A2 - 1       .backup one from first true
    Br  next;A3 == VL   .branch if all were false
    A6  <- A3 + A1      .offset into a
    S1  <- a,A6         .get new a value
next:VL <- A3           .VL is up to first true
    b,A1<- V4           .store into b
    A1  <- A1 + A3      .adjust base address of A array
    A4  <- A4 - A3      .decrement remaining iterations
    Br  done; A4 <= 0   .quit if all iterations done
    VL  <- min(A4,A5)   .readjust VL
    Br  loop
done:
```

Easily Understandable HLL Programming Paradigm

- Concept of vectors is usually understood by programmers
- This helps in expressing programs that are efficient on vector processors
- FORTRAN 90 further enforces the vector model

Conclusions and Observations

- Superscalar, VLIW, Vector advocates can all learn from one another.

- Challenge: close parallelism gap *and* follow RISC principles

- The region between modest superscalar and vector needs to be explored.
  Related question: with vectors, how much superscalar is needed?

- The role of single pipe vectors needs to be studied;
  i.e. vectors offer more than bandwidth.

- Speculative vector execution needs more development.

- Can parallelism-gap-closing principles be adapted to non-numeric processing?

- What vector architecture really needs is a healthy dose of good research...

Acknowledgements:

Greg Faanes

Wei Hsu

Corinna Lee