The Best Way to Achieve Vector-Like Performance?

J. E. Smith

April 20, 1994
Remember?

the

SEMANTIC
GAP
The Semantic Gap

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

Consider:

\[
\begin{align*}
\text{Do 10 } & \text{ i=1,looplength} \\
& \quad a(i) = b(i) \times x + c(i) \\
10 & \text{ continue}
\end{align*}
\]

CISC compilation:

\[
\begin{align*}
R1 & \leftarrow \text{looplength} \\
R2 & \leftarrow 1 \\
R3 & \leftarrow x \\
\text{loop: } & \quad a(R2) \leftarrow b(R2) \times R3 + c(R2) \\
& \quad \text{if } R2++ < R1 \text{ goto loop}
\end{align*}
\]

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 <- loope length .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
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.

(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)

\begin{verbatim}
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
\end{verbatim}

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

Also, data locality info is passed to hardware;

address stream:

\begin{verbatim}
a(1),a(2),a(3),a(4),...b(1),b(2),b(3),...,c(1),c(2),c(3),...
\end{verbatim}
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
Passing ILP to Hardware

Consider an example of control parallelism:

\[
\text{Do 20 i=1,looplength} \\
\text{if (a(i).eq.b(i)) then} \\
\quad c(i) = a(i) + e(i) \\
\text{endif} \\
20 \text{ continue}
\]

\[V1 \leftarrow \text{a .load a}
\]
\[V2 \leftarrow \text{b .load b}
\]
\[VM \leftarrow \text{V1==V2 .compare}
\]
\[V3 \leftarrow \text{e;VM .load e under mask}
\]
\[V4 \leftarrow \text{V1+V3; VM .add under mask}
\]
\[c \leftarrow \text{V4; VM .store under mask}
\]

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

\[
\begin{align*}
\text{VM:} & \quad 1 \\
\text{skip} & \quad \text{skip} \\
\text{skip} & \quad \text{skip} \\
\text{skip} & \quad \text{skip}
\end{align*}
\]

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
"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

The diagram shows the relationship between the execution time in clock periods and the percentage of ones in VM, with VL=100. The graph illustrates two scenarios:
- No zero skipping
- Zero skipping

The graph indicates a linear relationship between the percentage of ones in VM and the execution time, with a distinction between the two skipping scenarios.
Passing Address Patterns to Hardware

A simple example has already been given.

Also consider:

\[
\begin{align*}
\text{Do 20 } & i = 1, \text{length1} \\
\text{Do 20 } & j=1, \text{length2} \\
& c(i,j) = a(j,i) + e(i,j) \\
\text{20 } & \text{continue}
\end{align*}
\]

\[
\begin{align*}
\text{A1 } & \leftarrow \text{length1} \quad \text{.setup stride} \\
\text{V1 } & \leftarrow e, \text{A1} \quad \text{.load with stride}
\end{align*}
\]

\textit{AND}

\[
\begin{align*}
\text{Do 20 } & i = 1, \text{length1} \\
& c(i) = a(\text{index}(i)) + e(i) \\
\text{20 } & \text{continue}
\end{align*}
\]

\[
\begin{align*}
\text{V1 } & \leftarrow \text{index,1} \quad \text{.load index vector} \\
\text{V2 } & \leftarrow a, \text{V1} \quad \text{.gather with index vector}
\end{align*}
\]
*Modular, highly parallel control structure*

Yet simple instruction issue logic.

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

- with 8 64-element vector registers:

- A factor of 5 fewer logical crosspoints.
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.
How extensive is vector parallelism?
View from Vector Advocate’s Perspective

- Modest
- SuperScalar
- irregular
- serial
- regular
- parallel
View from Superscalar Advocate’s Perspective
View from VLIW Advocate’s Perspective
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:  \[ X(I) = A(I) \times X(I-1) + C(I) \]
Flexibility: Speculative Execution

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

\[
\begin{align*}
x &= a(1) \\
do 410 & \text{ i = 1, looplen} \\
b(i) &= x + c(i) \\
\text{if } (b(i).eq.0) & \text{ x = a(i+1)} \\
410 & \text{ continue}
\end{align*}
\]

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 overshootTM

  I.e. if branch is 90 percent taken, use \(VL = 10\)
  
  If 99 percent taken, use \(VL = 100\)
Speculative Vector Execution

Assume mostly true:

\[ A4 \leftarrow \text{loopen} \quad \text{.loop iterations} \]
\[ A5 \leftarrow \text{maxvl} \quad \text{.a profile-dependent VL} \]
\[ VL \leftarrow \text{min}(A4,A5) \quad \text{.initialize VL} \]
\[ A1 \leftarrow 0 \quad \text{.initialize offset into arrays} \]

\text{loop:}
\[ V1 \leftarrow a, A1 \quad \text{.load } a(i), \text{ offset by } A1 \]
\[ V3 \leftarrow c, A1 \quad \text{.load } c(i), \text{ offset by } A1 \]
\[ V4 \leftarrow V1 + V3 \quad \text{.add to get } b(i) \]
\[ VM \leftarrow V4 == 0 \quad \text{.test } b(i) \]
\[ A2 \leftarrow \text{LZ}(VM) \quad \text{.find leading zero count} \]
\[ A3 \leftarrow A2 - 1 \quad \text{.backup to first false} \]
\[ \text{Br next; } A4 == VL \quad \text{.branch if all were true} \]
\[ S1 \leftarrow V1(A3) \quad \text{.get previous } x \text{ from } A \text{ vector} \]
\[ S4 \leftarrow V3(A2) \quad \text{.get } c \text{ element.} \]
\[ S5 \leftarrow S1 + S4 \quad \text{.add} \]
\[ V4,A2 \leftarrow S5 \quad \text{.put into } b \text{ array} \]
\[ A3 \leftarrow A2 \quad \text{.will be new VL} \]

\text{next:}
\[ VL \leftarrow A3 \quad \text{.VL is up to first false} \]
\[ b,A1 \leftarrow V4 \quad \text{.store into } b \]
\[ A1 \leftarrow A1 + A2 \quad \text{.adjust base address of } A \text{ array} \]
\[ A4 \leftarrow A4 - A2 \quad \text{.decrement remaining iterations} \]
\[ \text{Br done; } A4 <= 0 \quad \text{.quit if all iterations done} \]
\[ VL \leftarrow \text{min}(A4,A5) \quad \text{.readjust VL} \]
\[ \text{Br loop} \]

done:
Speculative Vector Execution

Assume mostly false:

A4 <- looplan .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