Automatic Translation of Fortran Programs to Vector Form

Randy Allen and Ken Kennedy

The problem

- New (as of 1987) vector machines such as the Cray-1 have proven successful
- Most Fortran code is written sequentially, using loops
- Can we exploit parallelism without rewriting everything?

Compiler Vectorization

- Idea: Compiler detects parallelism and automatically converts loops
- No need to rewrite code or learn new language
- Opportunities for parallelism are subtle and difficult to detect
- Programmers need to tweak code into forms the optimizer can recognize

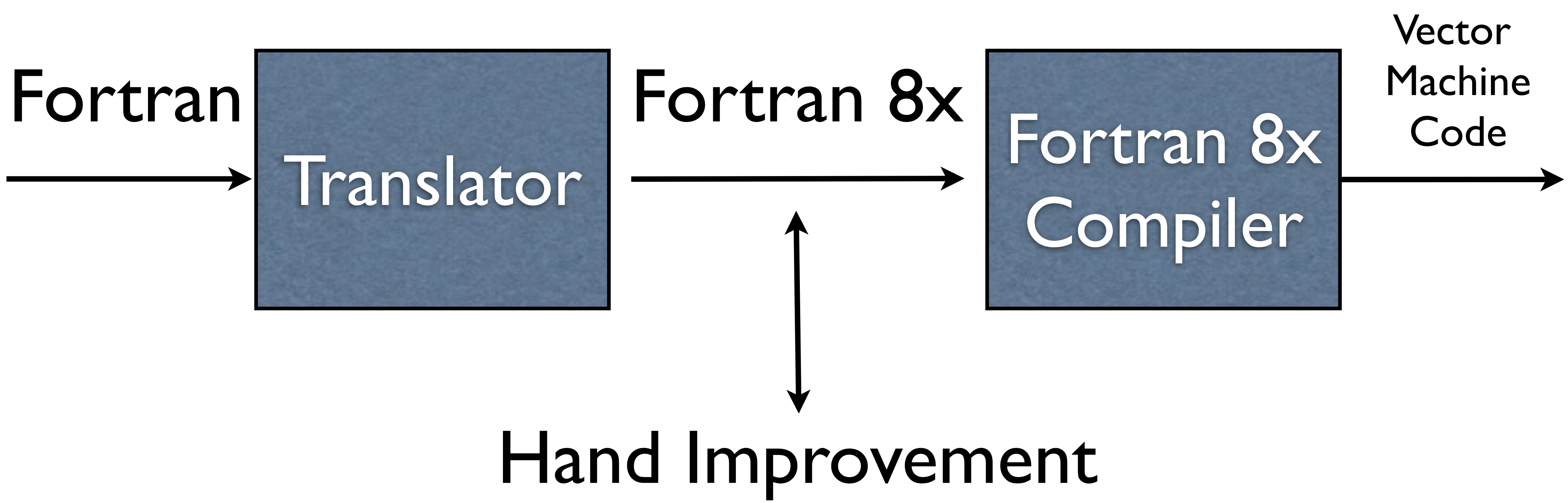
Explicit Vector Instructions

- Idea: Add forms to Fortran 8x to specify parallel operations
- Avoid writing sequential code in the first place
- Programmers better understand what parallelism opportunities exist
- Code must be rewritten

Source-to-Source Translation

- Idea: Automatically convert existing sequential Fortran into parallel Fortran 8x
- Translation only occurs once, so more expensive transformations are practical
- Programmers can add any needed parallelism the translator misses

Parallel Fortran Converter



Vector Operations in Fortran 8x

- Note: As of this paper, Fortran 8x is still theoretical
- Vectors and arrays may be treated as aggregates: X = Y + Z
- Arithmetic operators are applied point-wise
- Scalars are treated as same-valued vectors
- All arrays must have the same dimensions

Simultaneity

- Array assignment (e.g. X = Y) is *simultaneous*. All of Y is fetched before X is stored
- X = X / X(2) uses the value for X(2) prior to the assignment, even though X(2) will be assigned to
- Equivalent to using a temporary variable

Array Sections

- *Triplet* notation allows reference to parts of arrays
- `A(1:100, I) = B(J, 1:100)` assigns 100 elements from row J of B to column I of A
- Third element of triple specifies *stride*: A(2:100:2) references first 50 even subscript positions

Array Identification

- Specifies a named mapping to an array
- `IDENTIFY /1:M/ D(I) = C(I, I + 1)` defines D as the superdiagonal of C
- D is just an alias; it has no storage

Conditional Assignment

- `WHERE(A .GT. 0.0) A = A + B` indicates that only positive elements of A will be modified
- Errors in evaluating the right-hand side must be ignored when the predicate fails
- E.g., `WHERE(A .NE. 0.0) B = B/A`

Library Functions

- Mathematical functions (SIN, SQRT, etc.) are extended to operate on arrays
- New intrinsic array operations: DOTPRODUCT, TRANSPOSE
- `SEQ(1,N)` returns an index array
- Reductions operations, e.g. SUM

Translation Process

```
          DO 20 I = 1, 100
S1          KI = I
            DO 10 J = 1, 300, 3
S2            KI = KI + 2
S3            U(J) = U(J) * W(KI)
S4            V(J + 3) = V(J) + W(KI)
      10    CONTINUE
      20  CONTINUE
```

- Goal: transform S_3 and S_4 into vector instructions and remove them from the inner loop
- Only possible if there is no semantic difference

Simple Case

```
      DO 10 I = 1, 100
         X(I) = X(I) + Y(I)
10    CONTINUE
```

- Easily becomes `X(1:100) = X(1:100) + Y(1:100)`

```
      DO 10 I = 1, 100
         X(I + 1) = X(I) + Y(I)
10    CONTINUE
```

- Cannot be converted, because each iteration depends on the previous
  - Known as a *recurrence*

Dependency Detection

- To distinguish parallel and non-parallel loops, translator must detect self-dependent statements
- First, code is *normalized* to make this test feasible

DO-Loop Normalization

```
          DO 20 I = 1, 100
             KI = I
             DO 10 j = 1, 100
                KI = KI + 2
                U(3 * j − 2) = U(3 * j − 2) * W(KI)
                V(3 * j + 1) = V(3 * j − 2) + W(KI)
     10      CONTINUE
S6           J = 301
     20   CONTINUE
```

- Convert induction variables to iterate from 1 by steps of 1
- Here, J has been replaced by j
- S_6 added to preserve post-condition

Induction Variable Substitution

```
      DO 20 I = 1, 100
         KI = I
         DO 10 j = 1, 100
            U(3 * j - 2) = U(3 * j - 2) * W(KI + 2 * j)
            V(3 * j + 1) = V(3 * j - 2) + W(KI + 2 * j)
10       CONTINUE
         KI = KI + 200
         J = 301
20    CONTINUE
```

- Convert all subscripts to linear functions of induction variables
- KI has been removed from loop and replaced by its initial value plus its increments
- KI updated post-loop with final value
- Note: repeated addition replaced by multiplication

Dead Statement Elimination

```
        DO 20 I = 1, 100
          DO 10 j = 1, 100
S3          U(3 * j - 2) = U(3 * j - 2) * W(I + 2 * j)
S4          V(3 * j + 1) = V(3 * j - 2) + W(I + 2 * j)
   10     CONTINUE
   20   CONTINUE
```

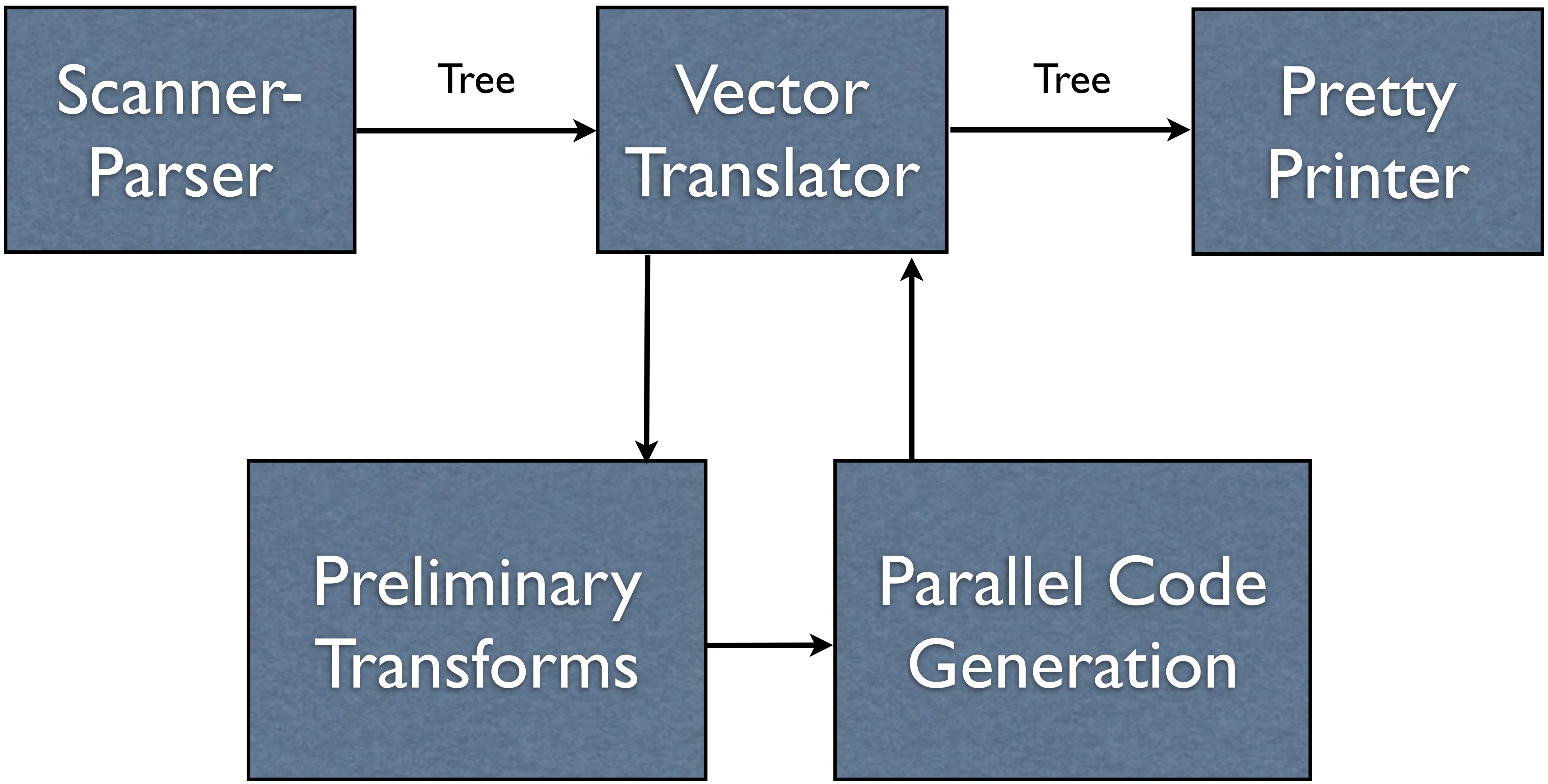
- Assuming J and KI aren't used outside the loop, their final values can be discarded
- Since they also aren't used within the loop, they can be removed entirely

Vector Code Generation

```
        DO 20 I = 1, 100
S3         U(1:298:3) = U(1:298:3) * W(I - 2:I + 200:2)
           DO 10 j = 1, 100
S4            V(3 * j + 1) = V(3 * j - 2) + W(I + 2 * j)
     10    CONTINUE
     20 CONTINUE
```

- Dependency analysis shows S_4 depends on itself, but S_3 does not
- Therefore, S_3 can be vectorized and moved out of the loop

Translation Process



Dependence Analysis

- S_2 depends on S_1 if some execution of S_2 uses a value from a previous S_1
- Self-dependence can only arise in loops

Dependency in Loops

```
      DO 10 J = 1, N
        X(J) = X(J) + C
10    CONTINUE
```

No dependency

```
      DO 10 J = 1, N − 1
        X(J + 1) = X(J) + C
10    CONTINUE
```

Recurrence

Dependency in Loops

```
          DO 10 i = 1, N
(*)          X(f(i)), = F(X(g(i)))
     10   CONTINUE
```

General Form

- (*) depends on itself iff there exist i_1, i_2 such that $1 \leq i_1 < i_2 \leq N$ and $f(i_1) = g(i_2)$
- Most often, *f* and *g* are linear in *i*
- $ax + by = n$ has a linear solution iff $gcd(a,b) \mid n$
- $f(i) = a_0 + a_1 i$; $g(i) = b_0 + b_1 i$
- (*) depends on itself only if $gcd(a_1,b_1) \mid b_0 - a_0$

Dependency in Loops

- Unfortunately, *gcd*(a_1,b_1) is commonly 1
- More sophisticated techniques are needed

COROLLARY 3 (BANERJEE INEQUALITY). *If* $f(x) = a_0 + a_1x$ *and* $g(y) = b_0 + b_1y$ *then statement* (*) *depends on itself only if*

$$-b_1 - (a_1^- + b_1)^+(N-2) \le b_0 + b_1 - a_0 - a_1 \le -b_1 + (a_1^+ - b_1)^+(N-2).$$

- Even these only provide *necessary* conditions for dependence
- Multiple loops are harder still

Indirect Dependence

```
          DO 10 I = 1, 100
S1           T(I) = A(I) * B(I)
S2           S(I) = S(I) + T(I)
S3           A(I + 1) = S(I) + C(I)
     10   CONTINUE
```

- S_1, S_2, and S_3 all depend indirectly on themselves

Types of Dependency

- We say S_2 depends on S_1 if one of these conditions hold
- *True dependence*: S_2 uses the output of S_1
- *Antidependence*: S_1 would use the output of S_2 if they were reversed
- *Output dependence*: S_2 recalculates the output of S_1

Loop-Related Dependency

- *Loop carried dependence*: one statement stores to a location; another statement reads from that location in a *later* iteration
- *Loop independent dependence*: one statement stores to a location; another statement reads from that location in the *same* iteration
- Self-dependence is a special case of loop carried dependence

Testing Procedure

- Test each pair of statements for dependence, building a dependence relation D
- Compute the transitive closure D^+
- Execute statements which do not depend on themselves in D^+ in parallel
- Execution order must be consistent with D^+
- Reduce cycles to π-blocks; the resulting graph is acyclic

Example

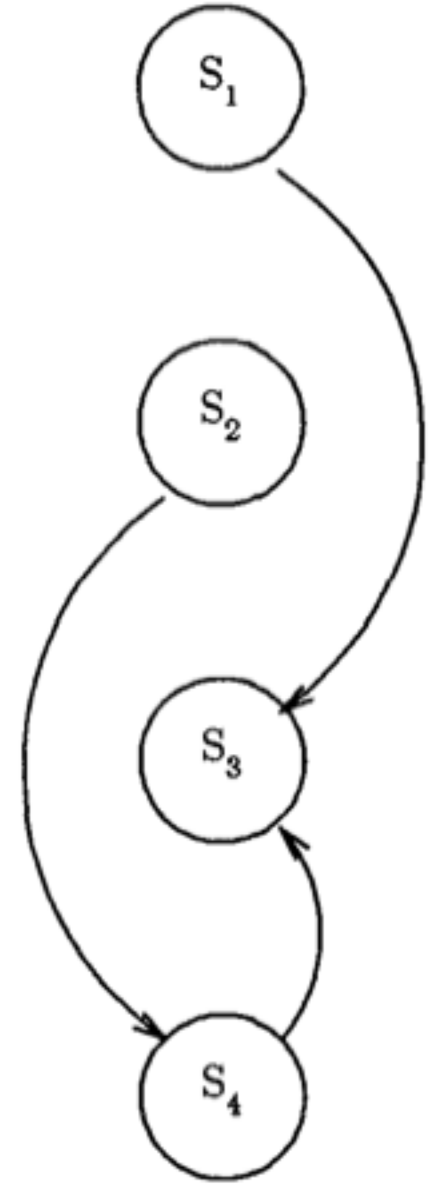
This…

```
        DO 10 I = 1, 99
S1        X(I) = I
S2        B(I) = 100-I
    10  CONTINUE
        DO 20 I = 1, 99
S3        A(I) = F(X(I))
S4        X(I + 1) = G(B(I))
    20  CONTINUE
```

Becomes…

```
X(1:99) = SEQ(1, 99, 1)
B(1:99) = SEQ(99, 1, −1)
X(2:100) = G(B(1:99))
A(1:99) = F(X(1:99))
```

Note: S_4 precedes S_3

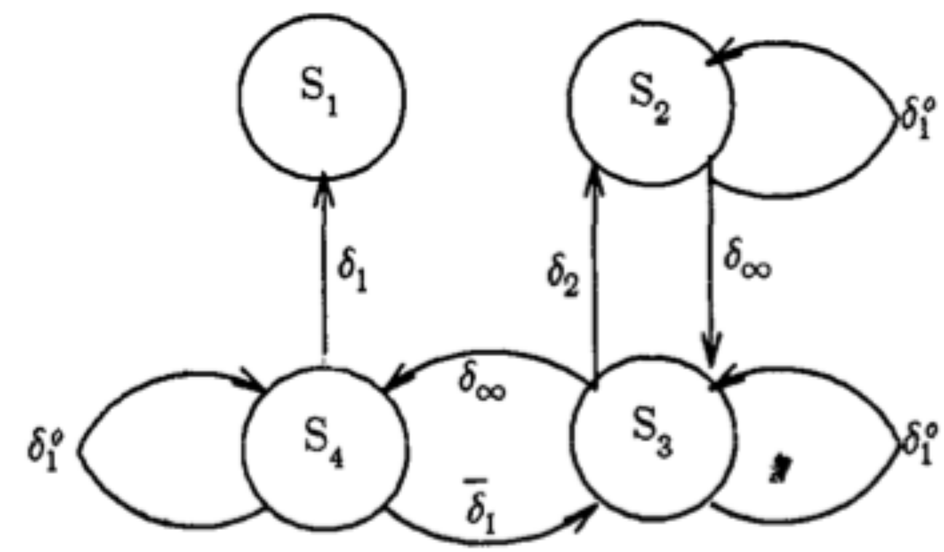


Multiple Loops

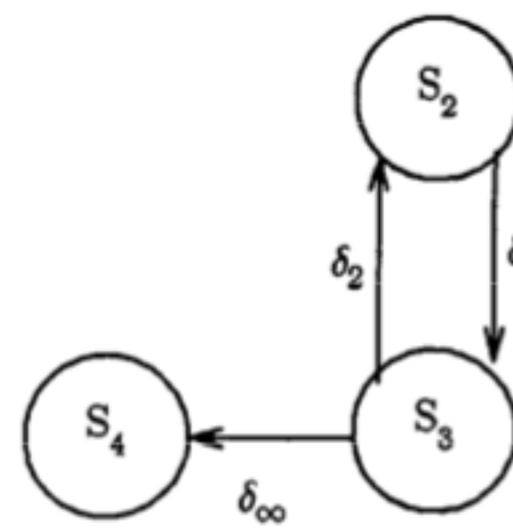
- Important to note which loop carries the dependence
- We can define a maximum depth where a given dependence occurs
- Loop independent dependencies have infinite depth
- Dependency arcs are labeled with depth and type

Example

```
          DO 30 I = 1, 100
S1           X(I) = Y(I) + 10
             DO 20 J = 1, 100
S2              B(J) = A(J, N)
                DO 10 K = 1, 50
S3                 A(J + 1, K) = B(J) + C(J, K)
     10         CONTINUE
S4              Y(I + J) = A(J + 1, N)
     20      CONTINUE
     30   CONTINUE
```




```
          DO 30 I = 1, 100
            code for S2, S3, S4
            generated at lower levels
     30   CONTINUE
S1        X(1:100) = Y(1:100) + 10
```



Example

```
          DO 30 I = 1, 100
             DO 20 J = 1, 100
                code for S2, S3
                generated at lower levels
     20      CONTINUE
S4           Y(I + 1:I + 100) = A(2:101, N)
     30   CONTINUE
S1        X(1:100) = Y(1:100) + 10
```

```
          DO 30 I = 1, 100
             DO 20 J = 1, 100
S2              B(J) = A(J, N)
S3              A(J + 1, 1:100) = B(J) + C(J, 1:100)
     20      CONTINUE
S4           Y(I + 1:I + 100) = A(2:101, N)
     30   CONTINUE
S1        X(1:100) = Y(1:100) + 10
```

Further Techniques

- *Loop interchange*: move recurrences to outer loops
- *Recurrence breaking*: antidependent and output dependent single-statement recurrences can be ignored
- *Thresholds*: recurrences may permit partial vectorization

Conditional Statements

Initial code

```
      DO 100 I = 1, N
        IF (A(I) .LE. 0) GOTO 100
        A(I) = B(I) + 3
100   CONTINUE
```

Convert to data dependency

```
      DO 100 I = 1, N
        BR1(I) = A(I) .LE. 0
        IF (.NOT. BR1(I)) A(I) = B(I) + 3
100   CONTINUE
```

Vectorize

```
BR1(1:N) = A(1:N) .LE. 0
WHERE (.NOT. BR1(1:N)) A(1:N) = B(1:N) + 3
```

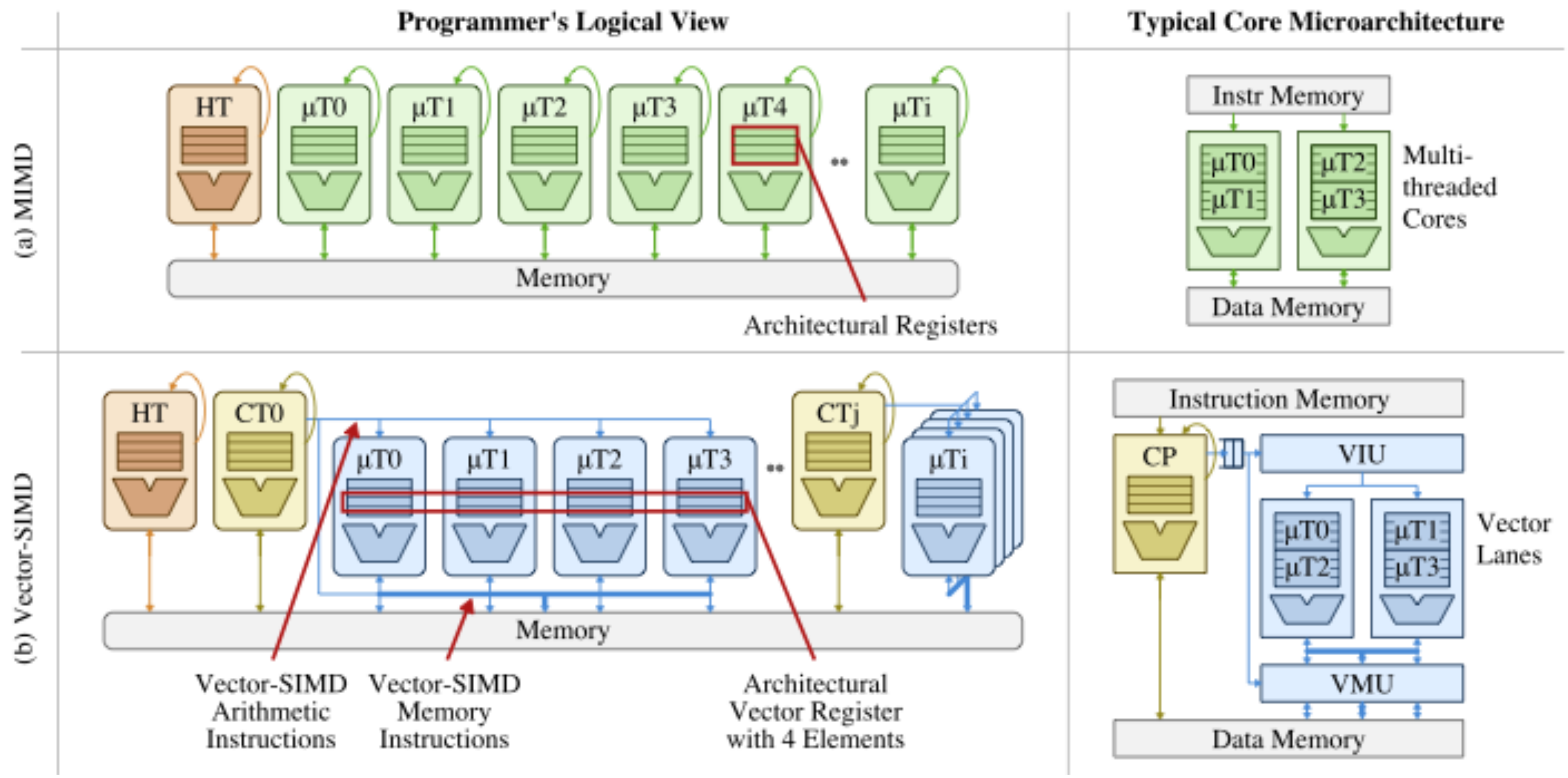
Implementation

- Initial work based on PARAFRASE
- PFC is ~25,000 lines of PL/I
- Implements most of the translations discussed in the paper
- Runs their test case in 1 min on a 3 MB machine

Exploring the tradeoffs between programmability and efficiency in data-parallel accelerators

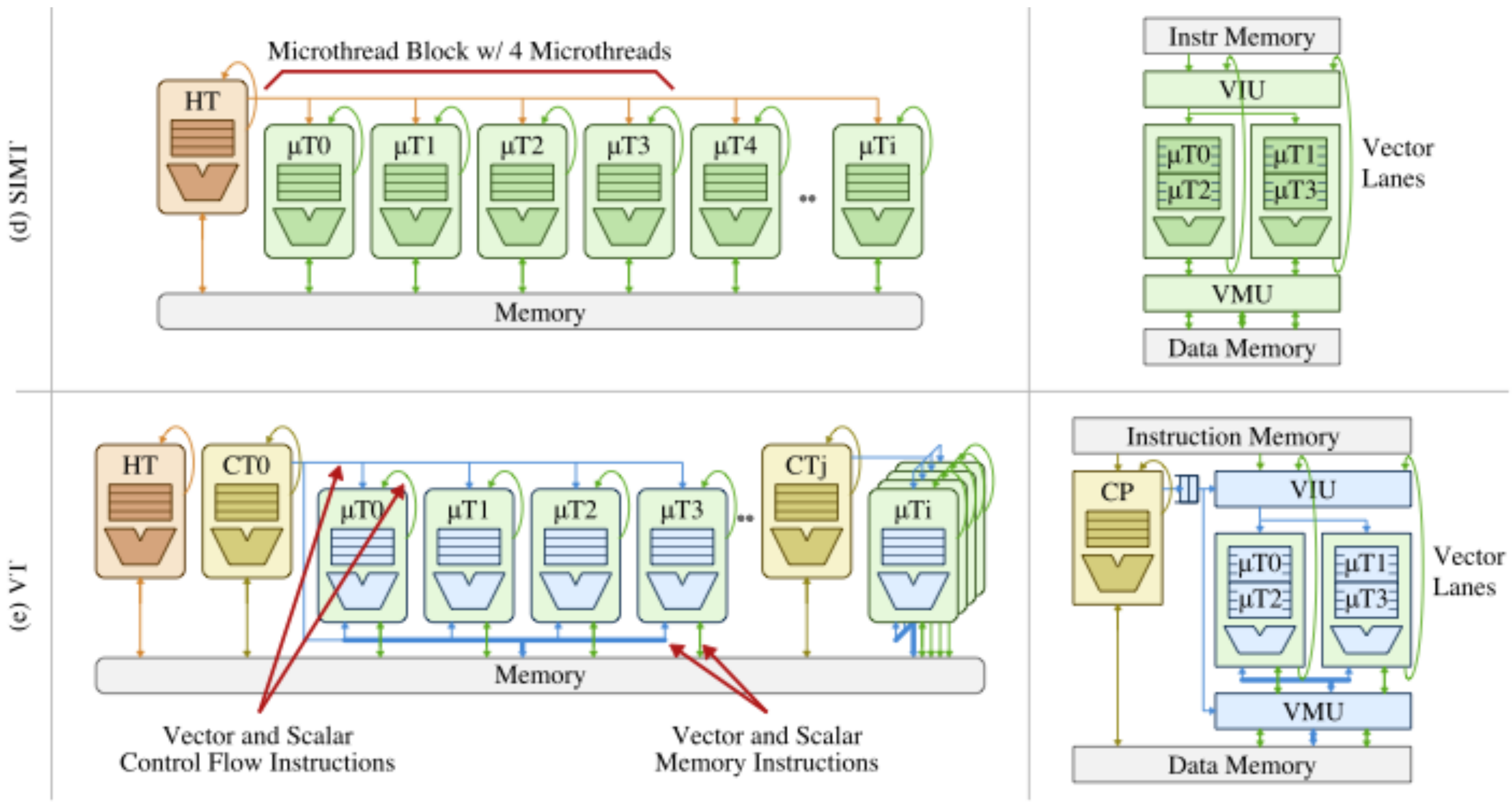
Yunsup Lee, Rimas Avizienis, Alex Bishara, Richard Xia,
Derek Lockhart, Christopher Batten and Krste Asanović

MIMD vs SIMD



Programmer's Logical View
Typical Core Microarchitecture
(a) MIMD
HT
µT0
µT1
µT2
µT3
µT4
µTi
Memory
Architectural Registers
Instr Memory
µT0
µT1
µT2
µT3
Multi-threaded Cores
Data Memory
(b) Vector-SIMD
HT
CT0
µT0
µT1
µT2
µT3
CTj
µTi
Memory
Vector-SIMD Arithmetic Instructions
Vector-SIMD Memory Instructions
Architectural Vector Register with 4 Elements
Instruction Memory
CP
VIU
µT0
µT2
µT1
µT3
Vector Lanes
VMU
Data Memory

Two Hybrid Approaches



SIMT

- Combines MIMD's logical view with vector-SIMD's microarchitecture
- VIU executes multiple µTs using SIMD as long as they proceed on the same control path
- VIU uses masks to selectively disable inactive µTs on different paths

VT

- HT manages CTs; CTs manage µTs
- Vector-fetch instruction indicates scalar instructions to be executed by µTs
- VIU operates µTs in SIMD manner, but scalar branch can cause divergence

Irregular Control Flow Example

```
for ( i = 0; i < n; i++ )
  if ( A[i] > 0 )
    C[i] = x * A[i] + B[i];
```

```
  load      x, x_ptr
loop:
  setvl     vlen, n
  load.v    VA, a_ptr
  load.v    VB, b_ptr
  cmp.gt.v  VF, VA, 0
  mul.sv    VT, x, VA,  VF
  add.vv    VC, VT, VB, VF
  store.v   VC, c_ptr,  VF
  add       a_ptr, vlen
  add       b_ptr, vlen
  add       c_ptr, vlen
  sub       n, vlen
  br.neq    n, 0, loop
```

(b) Vector-SIMD

```
  br.gte tidx, n, done
  add    a_ptr, tidx
  load   a, a_ptr
  br.eq  a, 0, done
  add    b_ptr, tidx
  add    c_ptr, tidx
  load   x, x_ptr
  load   b, b_ptr
  mul    t, x, a
  add    c, t, b
  store  c, c_ptr
done:
```

(c) SIMT

```
  load    x, x_ptr
  mov.sv  VZ, x
loop:
  setvl   vlen, n
  load.v  VA, a_ptr
  load.v  VB, b_ptr
  mov.sv  VD, c_ptr
  fetch.v ut_code
  add     a_ptr, vlen
  add     b_ptr, vlen
  add     c_ptr, vlen
  sub     n, vlen
  br.neq  n, 0, loop
  ...
ut_code:
  br.eq   a, 0, done
  mul     t, z, a
  add     c, t, b
  add     d, tidx
  store   c, d
done:
  stop
```

(d) VT

Summary

- Vector-based microarchitectures more area and energy efficient than scalar-based
- Maven (VT) more efficient and easier to program than vector-SIMD
- Suggestion that VT more efficient but harder to program than SIMT