COMP 515: Advanced Compilation for Vector and Parallel Processors

Prof. Vivek Sarkar
Department of Computer Science
Rice University
vsarkar@rice.edu

https://wiki.rice.edu/confluence/display/PARPROG/COMP515

Control Dependences (Recap)

Chapter 7

IF Conversion: Forward Branches

- **Remove forward branches by inserting appropriate guards**

```
      DO 100 I = 1,N
C1     IF (A(I).GT.10) GO TO 60
20     A(I) = A(I) + 10
C2     IF (B(I).GT.10) GO TO 80
40     B(I) = B(I) + 10
60     A(I) = B(I) + A(I)
80     B(I) = A(I) - 5
      ENDDO
```

➔

```
        DO 100 I = 1,N
          m1 = A(I).GT.10
 20       IF(.NOT.m1) A(I) = A(I) + 10
          IF(.NOT.m1) m2 = B(I).GT.10
 40       IF(.NOT.m1.AND..NOT.m2) B(I) = B(I) + 10
 60       IF(.NOT.m1.AND..NOT.m2.OR.m1)A(I) = B(I) + A(I)
 80       IF(.NOT.m1.AND..NOT.m2.OR.m1.OR..NOT.m1
                .AND.m2) B(I) = A(I) - 5
         ENDDO
```

IF Conversion: Forward Branches

- **We can simplify to:**

```
   DO 100 I = 1,N
        m1 = A(I).GT.10
20      IF(.NOT.m1) A(I) = A(I) + 10
        IF(.NOT.m1) m2 = B(I).GT.10
40      IF(.NOT.m1.AND..NOT.m2)
                B(I) = B(I) + 10
60      IF(m1.OR..NOT.m2)
                A(I) = B(I) + A(I)
80      B(I) = A(I) - 5
   ENDDO
```

- **and then vectorize to:**

```
     m1(1:N) = A(1:N).GT.10
20   WHERE(.NOT.m1(1:N)) A(1:N) = A(1:N) + 10
     WHERE(.NOT.m1(1:N)) m2(1:N) = B(1:N).GT.10
40   WHERE(.NOT.m1(1:N).AND..NOT.m2(1:N))
                B(1:N) = B(1:N) + 10
60   WHERE(m1(1:N).OR..NOT.m2(1:N))
                A(1:N) = B(1:N) + A(1:N)
80   B(1:N) = A(1:N) - 5
```

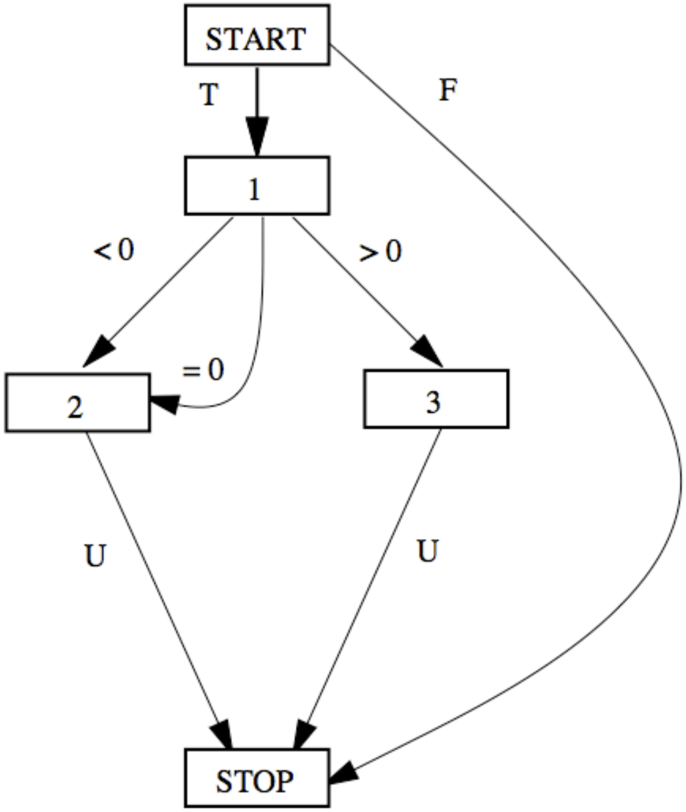
Control Dependence: Definition

Node Y is *control dependent* on node X with label L in *CFG* if and only if

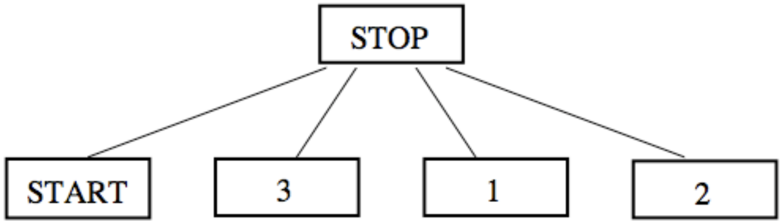
1. there exists a nonnull path $X \longrightarrow Y$, starting with the edge labeled L, such that Y post-dominates every node, W, strictly between X and Y in the path, and
2. Y does not post-dominate X.

Reference: "The Program Dependence Graph and its Use in Optimization", J. Ferrante et al, *ACM TOPLAS, 1987*

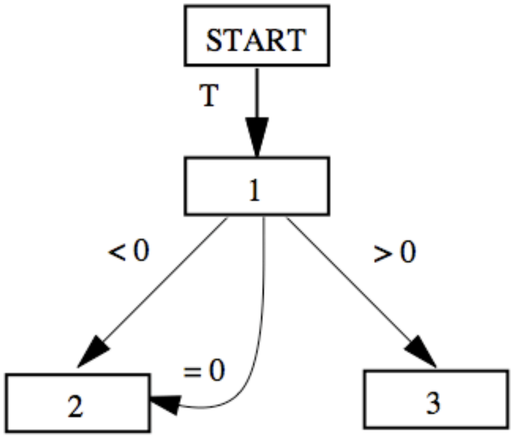
Example: Acyclic CFG and its Control Dependence Graph (CDG)



CONTROL FLOW GRAPH



POSTDOMINATOR TREE



CONTROL DEPENDENCE GRAPH

Control Dependence and Parallelization

- From Chapter 2: Most loop transformations are unaffected by loop-independent dependences
  - A forward-branch need not inhibit coarse-grain parallelization
- Iteration-reordering transformations like loop reversal, loop skewing, strip mining, index-set splitting, loop interchange do not affect loop-independent dependences
- Statement reordering transformations might be problematic: loop fusion, loop distribution
  - Distribution can be performed by including control dependences in recurrence analysis, and performing scalar expansion on branch condition
  - Fusion of loops that do not contain exit branches is also possible

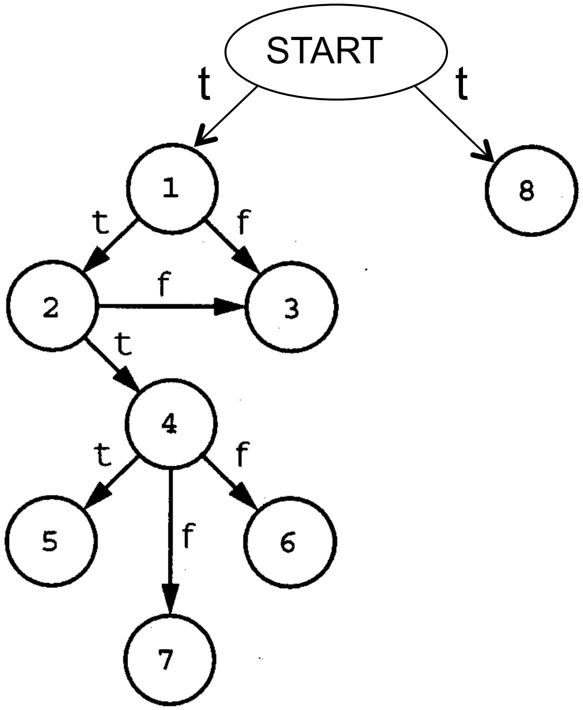
Loop Distribution

- **Example:**

```
    DO I = 1, N
1       IF (A(I).NE.0) THEN
2         IF (B(I)/A(I).GT.1) GOTO 4
        ENDIF
3       A(I) = B(I)
        GOTO 8
4       IF (A(I).GT.T) THEN
5         T = (B(I) - A(I)) + T
        ELSE
6         T = (T + B(I)) - A(I)
7         B(I) = A(I)
        ENDIF
8       C(I) = B(I) + C(I)
    ENDDO
```

Control Dependence Graph

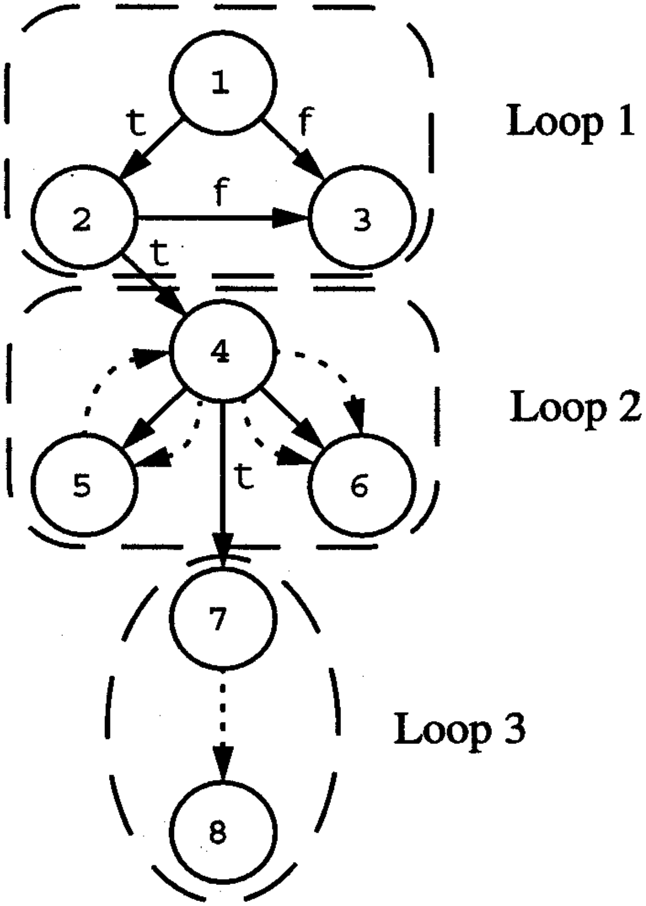
for loop body



Loop Distribution

- Fusion into "like" regions
  - Loop 1 is parallel
  - Loop 2 is sequential
  - Loop 3 is parallel

```
DO I = 1, N
 1      IF (A(I).NE.0) THEN
 2        IF (B(I)/A(I).GT.1) GOTO 4
        ENDIF
 3      A(I) = B(I)
        GOTO 8
 4      IF (A(I).GT.T) THEN
 5        T = (B(I) - A(I)) + T
        ELSE
 6        T = (T + B(I)) - A(I)
 7        B(I) = A(I)
        ENDIF
 8      C(I) = B(I) + C(I)
     ENDDO
```



Selective IF Conversion: Need execution variables E2(I) and E4(I) to hold result of branches at statement 2 and 4

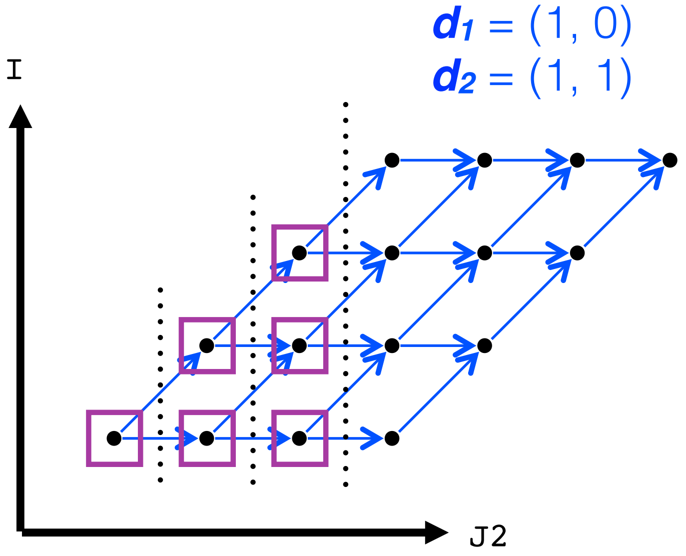
Conclusion

- Idea behind control flow dependences
- If-conversion
  - Types of branches and branch removal
  - Iterative dependences (append range to each statement)
- Control Dependence Procedure as alternative to if-conversion
- Execution model for control dependence graphs
- Loop Distribution (selective if-conversion)
- Code Generation

Performance Issues with Wavefront Transformation

- Large synchronization overhead
  - Need barrier for each outer-iteration (J2 loop)
- Performance issues
  - Non-uniform iteration lengths in DOALL loop
  - Non-contiguous data access after skewing (in sequential version or when DOALL loop is chunked)

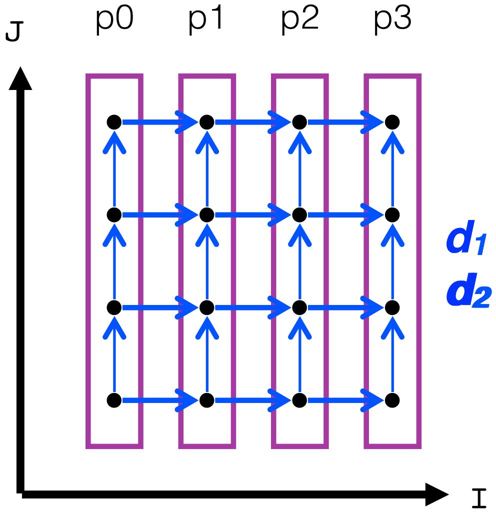
```
! ex.2
DO J2 = 1, N+M-1
  ILW = MAX(1,J2-M+1)
  IUP = MIN(N,J2)
  PARALLEL DO I = ILW, IUP
    J = J2 - I + 1
    A(J,I) = A(J-1,I) + A(J,I-1)
  END DO
END DO
```



Doacross Parallelization

- Loop-carried dependences exist among iterations
- Parallel execution can be enabled via point-to-point synchronization among iterations of DOACROSS loop
  - Synchronizations are expressed using POST and WAIT

```
! ex.2
DOACROSS I = 1, N
  DO J = 1, M
    IF (I.GE.2) WAIT(I-1,J)
    A(J,I) = A(J-1,I) + A(J,I-1)
    POST(I,J)
  END DO
END DO
```



$d_1 = (0, 1)$

$\mathbf{d_2 = (1, 0)}$

Implementing POST and WAIT operations

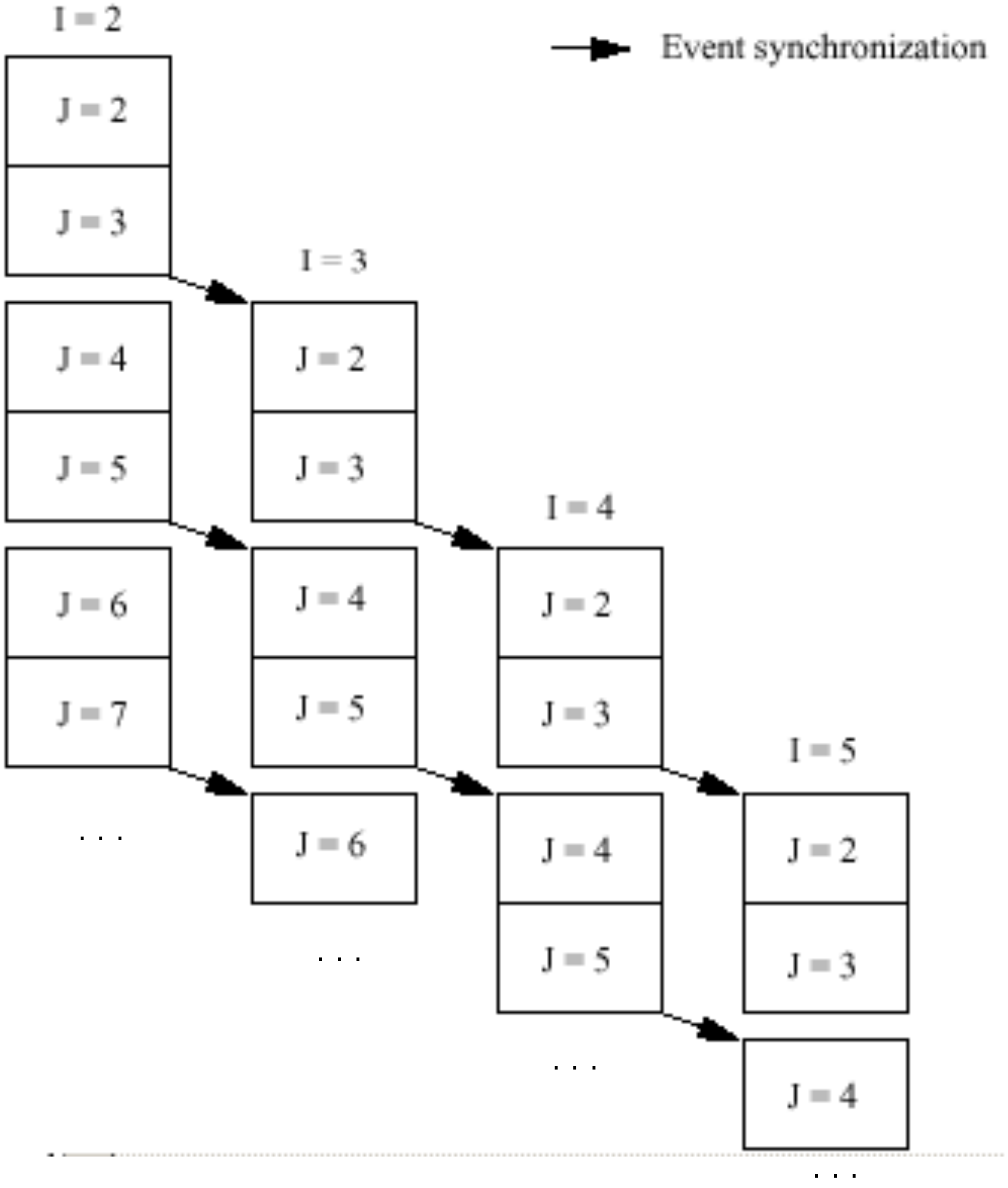
Two approaches:

1. **Use event variables (Section 6.6.2 of textbook)**
    - Allocate an array of event variables, one per iteration
    - Perform POST and WAIT operations on event variables, e.g., POST (EV(I, J)) and WAIT (EV(I-1, J))
    - Pros: straightforward implementation approach
    - Cons: inefficient in space, not adaptable to available hardware parallelism
2. **Special runtime support for post/wait (OpenMP 4.1)**
    - Each processor maintains only n integer synchronization variables, where n is the number of loops in a doacross loop nest
    - Dependent iteration examines source iteration's sync variables to check ready condition
    - Pros: space-efficient (only n*P sync variables for P processors)
    - Cons: need runtime support in addition to compiler transformation

Extension with 2x unroll/tiling

```
DO I = 2, N-1
    DO J = 2, N-1
        A(I, J) = .25 * (A(I-1, J) + A(I, J-1) +
                         A(I+1, J) + A(I, J+1))
    ENDDO
ENDDO
==>
POST (EV(1, 1))
DOACROSS I = 2, N-1
    K = 0
    DO J = 2, N-1, 2   ! TILE SIZE = 2
        K = K+1
        WAIT (EV(I-1,K))
        DO m = J, MIN(J+1, N-1)
            A(I, m) = .25 * (A(I-1, m) + A(I, m-1) +
                             A(I+1, m) + A(I, m+1))
        ENDDO
        POST (EV(I, K+1))
    ENDDO
ENDDO
```

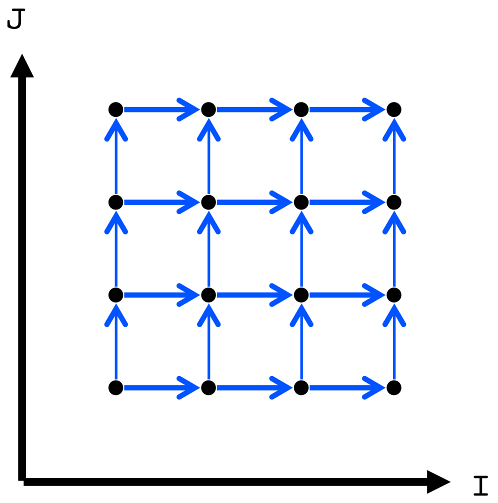
Extension with 2x unroll/tiling (contd)



Doacross Support in OpenMP 4.1

- `ordered(`*n*`)` : *n* specifies nest-level of doacross
- `depend(sink:` *vect*`)` : wait for iteration *vect* to reach `source`
- `depend(source)` : notify that current iteration reached
- C code example

```
! ex.5b
#pragma omp for ordered(2)
for (i = 1; i < n; i++) {
  for (j = 1; j < m; j++) {
    A[i][j] = foo(i, j);          // S1
    #pragma omp ordered depend(sink: i-1,j) \\
                        depend(sink: i,j-1)
    B[i][j] = bar(A[i][j],
                  B[i-1][j],
                  B[i][j-1]);     // S2
    #pragma omp ordered depend(source)
    C[i][j] = baz(B[i][j]);       // S3
}
```



Compiler Improvement of Register Usage

Chapter 8

Overview

- Improving memory hierarchy performance by compiler transformations
    - Scalar Replacement
    - Unroll-and-Jam
- Saving memory loads & stores
- Make good use of the processor registers

Motivating Example

```
DO I = 1, N
   DO J = 1, M
        A(I) = A(I) + B(J,I)
   ENDDO
ENDDO
```

- A(I) can be left in a register throughout the inner loop
- Standard register allocation fails to recognize this

```
DO I = 1, N
   T = A(I)
   DO J = 1, M
        T = T + B(J,I)
   ENDDO
   A(I) = T
ENDDO
```

- All loads and stores to A in the inner loop have been saved
- High chance of T being allocated a register by standard register allocation

Scalar Replacement

- Convert array reference to scalar reference to improve performance of the register allocator
- Our approach is to use dependences to achieve these memory hierarchy transformations

Dependence and Memory Hierarchy

- True or Flow dependence - save loads and cache misses
- Anti dependence - save cache misses
- Output dependence - save stores and cache misses
- Input "dependence" - save loads and cache misses
  - Read-read control flow path with no intervening write

```
A(I) = ... + B(I)
...  = A(I) + k
A(I) = ...
...  = B(I)
```

Dependence and Memory Hierarchy

- Loop Carried dependences - Consistent dependences most useful for memory management purposes
- Consistent dependences - dependences with constant threshold (dependence distance)

Dependence and Memory Hierarchy

- Problem of overcounting optimization opportunities. For example

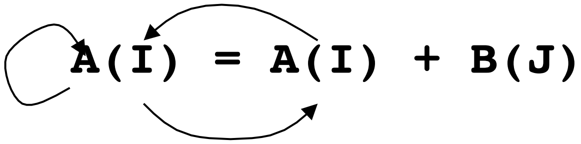
```
S1: A(I) = ...
S2: ...  = A(I)
S3: ...  = A(I)
```

- But we can save only two memory references not three
- Solution - Prune edges from dependence graph which don't correspond to savings in memory accesses

Using Dependences

- In the reduction example

```
DO I = 1, N
   DO J = 1, M
```



```
      A(I) = A(I) + B(J)
   ENDDO
ENDDO
```

```
DO I = 1, N
   T = A(I)
   DO J = 1, M
      T = T + B(J)
   ENDDO
   A(I) = T
ENDDO
```

- True dependence - replace the references to A in the inner loop by scalar T
- Output dependence - store can be moved outside the inner loop
- Anti dependence - load can be moved before the inner loop

Scalar Replacement

- Example: Scalar Replacement in case of loop independent dependence

```
DO I = 1, N
     A(I) = B(I) + C
     X(I) = A(I)*Q
ENDDO
```

```
DO I = 1, N
     t = B(I) + C
     A(I) = t
     X(I) = t*Q
ENDDO
```

- One fewer load for each iteration for reference to *A*

Scalar Replacement

- Example: Scalar Replacement in case of loop carried dependence spanning single iteration

```
DO I = 1, N
    A(I) = B(I-1)
    B(I) = A(I) + C(I)
ENDDO
```

```
tB = B(0)
DO I = 1, N
    tA = tB
    A(I) = tA
    tB = tA + C(I)
    B(I) = tB
ENDDO
```

- One fewer load for each iteration for reference to B which had a loop carried true dependence spanning 1 iteration
- Also one fewer load per iteration for reference to A

Scalar Replacement

- Example: Scalar Replacement in case of loop carried dependence spanning multiple iterations

```
DO I = 1, N
    A(I) = B(I-1) + B(I+1)
ENDDO
```

```
t1 = B(0)
t2 = B(1)
DO I = 1, N
    t3 = B(I+1)
    A(I) = t1 + t3
    t1 = t2
    t2 = t3
ENDDO
```

- One fewer load for each iteration for reference to B which had a loop carried input dependence spanning 2 iterations
- Invariants maintained were

`t1=B(I-1);t2=B(I);t3=B(I+1)`

Eliminate Scalar Copies

```
t1 = B(0)
t2 = B(1)
DO I = 1, N
    t3 = B(I+1)
    A(I) = t1 + t3
    t1 = t2
    t2 = t3
ENDDO
```

- Unnecessary register-register copies
- Unroll loop 3 times

```
t1 = B(0)
t2 = B(1)
mN3 = MOD(N,3)
DO I = 1, mN3
    t3 = B(I+1)
    A(I) = t1 + t3
    t1 = t2
    t2 = t3
ENDDO
DO I = mN3 + 1, N, 3
    t3 = B(I+1)
    A(I) = t1 + t3
    t1 = B(I+2)
    A(I+1) = t2 + t1
    t2 = B(I+3)
    A(I+2) = t3 + t2
ENDDO
```

Preloop

Main Loop