CSC 255/455 Software Analysis and Improvement

Enhancing Parallelism

Instructor: Chen Ding

Chapter 5, Optimizing Compilers for Modern Architectures, Allen and Kennedy
www.cs.rice.edu/~ken/comp515/lectures/

Where Does Vectorization Fail?

```
procedure vectorize (L, k, D)
// L is the maximal loop nest containing the statement.
// k is the current loop level
// D is the dependence graph for statements in L.
find the set {S1, S2, ... , Sm} of SCCs in D
construct Lp from L by reducing each Si to a single node
use topological sort to order nodes in Lp to {p1, p2, ... , pm}
for i = 1 to m do begin
  if pi is a dependence cycle then
      generate a level-k DO
      construct Di be pi dependence edges in D at level k+1 or greater
      codegen (pi, k+1, Di)
      generate the level-k ENDDO
  else
     vectorize pi with respect to every loop containing it
  end
end vectorize
```


Fine-Grained Parallelism

Techniques to enhance fine-grained parallelism:

- Loop Interchange
- Scalar Expansion
- Scalar Renaming
- Array Renaming
- Node Splitting

Motivational Example

```
DO J = 1, M
  DO I = 1, N
       T = 0.0
       DO K = 1,L
           T = T + A(I,K) * B(K,J)
       ENDDO
       C(I,J) = T
  ENDDO
ENDDO
```


Motivational Example

```
DO J = 1, M
  DO I = 1, N
     T$(I) = 0.0
     DO K = 1,L
           T$(I) = T$(I) + A(I,K) * B(K,J)
      ENDDO
      C(I,J) = T$(I)
   ENDDO
ENDDO
```


Motivational Example II

- Loop Distribution gives us:

```
DO J = 1, M
   DO I = 1, N
      T$(I) = 0.0
   ENDDO
   DO I = 1, N
      DO K = 1,L
         T$(I) = T$(I) + A(I,K) * B(K,J)
      ENDDO
   ENDDO
   DO I = 1, N
      C(I,J) = T$(I)
   ENDDO
ENDDO
```

Motivational Example III

Finally, interchanging `I` and `K` loops, we get:

```
DO J = 1, M
  T$(1:N) = 0.0
    DO K = 1,L
       T$(1:N) = T$(1:N) + A(1:N,K) * B(K,J)
    ENDDO
  C(1:N,J) = T$(1:N)
ENDDO
```

- A couple of new transformations used:
  - Loop interchange
  - Scalar Expansion

Loop Interchange

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

- DV:

Loop Interchange

- Loop interchange is a reordering transformation
- Why?
  - Think of statements being parameterized with the corresponding iteration vector
  - Loop interchange merely changes the execution order of these statements.
  - It does not create new instances, or delete existing instances

```
 DO J = 1, M
    DO I = 1, N
S     <some statement>
    ENDDO
 ENDDO
```

- If interchanged, S(2, 1) will execute before S(1, 2)

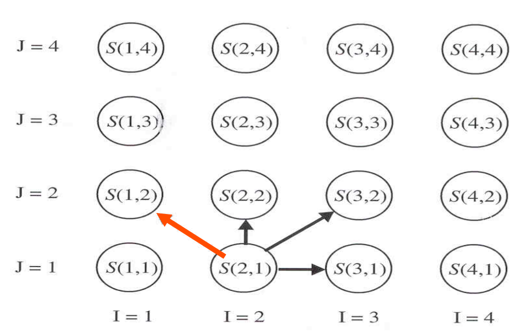
Loop Interchange: Safety

- Safety: not all loop interchanges are safe

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

- If we interchange loops, will we violate a dependence

Loop Interchange: Safety



- A dependence is interchange-preventing with respect to a given pair of loops if interchanging those loops would reorder the endpoints of the dependence.

Scalar Expansion

Scalar Expansion

- Can we vectorize the following code?

```
      DO I = 1, N
S1      T = A(I)
S2      A(I) = B(I)
S3      B(I) = T
     ENDDO
```

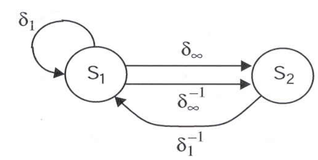

Scalar Expansion

- However, not always profitable. Consider:

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

- Scalar expansion gives us:

```
     T$(0) = T
     DO I = 1, N
S1        T$(I) = T$(I-1) + A
S2        A(I) = T$(I)
     ENDDO
     T = T$(N)
```



Scalar Expansion: Safety

- Scalar expansion is always safe
- When is it profitable?
  - Naïve approach: Expand all scalars, vectorize, shrink all unnecessary expansions.
  - However, we want to predict when expansion is profitable
- Dependences due to reuse of memory location vs. flow of values
  - Dependences due to flows of values must be preserved
  - Dependences due to reuse of memory location can be deleted by expansion

Scalar Expansion: Covering Definitions

- A definition X of a scalar S is a covering definition for loop L if a definition of S placed at the beginning of L reaches no uses of S that occur past X.

```
   DO I = 1, 100
S1     T = X(I)
S2     Y(I) = T
   ENDDO
```

covering

```
   DO I = 1, 100
      IF (A(I) .GT. 0) THEN
S1         T = X(I)
S2         Y(I) = T
      ENDIF
   ENDDO
```

covering

Scalar Expansion: Covering Definitions

- A covering definition does not always exist:

```
     DO I = 1, 100
        IF (A(I) .GT. 0) THEN
S1           T = X(I)
        ENDIF
S2      Y(I) = T
     ENDDO
```

- In SSA terms: There does not exist a covering definition for a variable T if the edge out of the first assignment to T goes to a ϕ-function later in the loop which merges its values with those for another control flow path through the loop

Scalar Expansion: Covering Definitions

- We will consider a collection of covering definitions
- There is a collection C of covering definitions for T in a loop if either:
  - There exists no ϕ-function at the beginning of the loop that merges versions of T from outside the loop with versions defined in the loop, or,
  - The ϕ-function within the loop has no SSA edge to any ϕ-function including itself

Scalar Expansion: Covering Definitions

- Remember the loop which had no covering definition:

```
      DO I = 1, 100
           IF (A(I) .GT. 0) THEN
S1               T = X(I)
            ENDIF
S2         Y(I) = T
       ENDDO
```

- To form a collection of covering definitions, we can insert dummy assignments:

```
      DO I = 1, 100
          IF (A(I) .GT. 0) THEN
S1              T = X(I)
          ELSE
S2              T = T
          ENDIF
S3        Y(I) = T
       ENDDO
```


Scalar Expansion: Covering Definitions

- Algorithm to insert dummy assignments and compute the collection, C, of covering definitions:
  - Central idea: Look for parallel paths to a ϕ-function following the first assignment, until no more exist
  - Algorithm: see textbook

Scalar Expansion: Covering Definitions

```
      DO I = 1, 100
          IF (A(I) .GT. 0) THEN
S1              T = X(I)
          ENDIF
S2        Y(I) = T
      ENDDO
```

After inserting covering definitions:

```
DO I = 1, 100
      IF (A(I) .GT. 0) THEN
S1            T = X(I)
      ELSE
S2            T = T
      ENDIF
S3     Y(I) = T
   ENDDO
```

After scalar expansion:

```
T$(0) = T
DO I = 1, 100
      IF (A(I) .GT. 0) THEN
S1            T$(I) = X(I)
      ELSE
              T$(I) = T$(I-1)
      ENDIF
S2     Y(I) = T$(I)
ENDDO
```


Deletable Dependences

- Uses of T before covering definitions are expanded as T$(I - 1)
- All other uses are expanded as T$(I)
- The deletable dependences are:
  - Backward carried antidependences
  - Backward carried output dependences
  - Forward carried output dependences
  - Loop-independent antidependences into the covering definition
  - Loop-carried true dependences from a covering definition

Scalar Expansion: Drawbacks

- Expansion increases memory requirements
- Solutions:
  - Expand in a single loop
  - Strip mine loop before expansion
  - Forward substitution:

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

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


Scalar and Array Renaming

Scalar Renaming

```
   DO I = 1, 100
S1    T = A(I) + B(I)
S2    C(I) = T + T
S3    T = D(I) - B(I)
S4    A(I+1) = T * T
  ENDDO
```

- Renaming scalar T:

```
 DO I = 1, 100
S1    T1 = A(I) + B(I)
S2    C(I) = T1 + T1
S3    T2 = D(I) - B(I)
S4    A(I+1) = T2 * T2
  ENDDO
```


Scalar Renaming

- will lead to:

```
S3    T2$(1:100) = D(1:100) - B(1:100)
S4    A(2:101) = T2$(1:100) * T2$(1:100)
S1    T1$(1:100) = A(1:100) + B(1:100)
S2    C(1:100) = T1$(1:100) + T1$(1:100)
       T = T2$(100)
```


Scalar Renaming

- Renaming algorithm partitions all definitions and uses into equivalent classes, each of which can occupy different memory locations:
  - Use the definition-use graph to:
  - Pick definition
  - Add all uses that the definition reaches to the equivalence class
  - Add all definitions that reach any of the uses…
  - ..until fixed point is reached

Scalar Renaming: Profitability

- Scalar renaming will break recurrences in which a loop-independent output dependence or antidependence is a critical element of a cycle
- Relatively cheap to use scalar renaming
- Usually done by compilers when calculating live ranges for register allocation

Array Renaming

```
  DO I = 1, N
S1      A(I) = A(I-1) + X
S2      Y(I) = A(I) + Z
S3      A(I) = B(I) + C
  ENDDO
```

- $S_1 \delta_\infty S_2$ $S_2 \delta_\infty^{-1} S_3$ $S_3 \delta_1 S_1$ $S_1 \delta_\infty^{0} S_3$

- Rename A(I) to A$(I):

```
   DO I = 1, N
S1      A$(I) = A(I-1) + X
S2      Y(I) = A$(I) + Z
S3      A(I) = B(I) + C
   ENDDO
```

- Dependences remaining: $S_1 \delta_\infty S_2$ and $S_3 \delta_1 S_1$

Array Renaming: Profitability

- Examining dependence graph and determining minimum set of critical edges to break a recurrence is NP-complete!
- Solution: determine edges that are removed by array renaming and analyze effects on dependence graph
- procedure array_partition:
  - Assumes no control flow in loop body
  - identifies collections of references to arrays which refer to the same value
  - identifies deletable output dependences and antidependences
- Use this procedure to generate code
  - Minimize amount of copying back to the “original” array at the beginning and the end

Node Splitting

Node Splitting

- Can we vectorize the following loop?

```
DO I = 1, N
S1:  A(I) = X(I+1) + X(I)
S2:  X(I+1) = B(I) + 32
ENDDO
```


Node Splitting

```
DO I = 1, N
S1:  A(I) = X(I+1) + X(I)

S2:  X(I+1) = B(I) + 32
ENDDO
```

- Break critical antidependence
- Make copy of node from which antidependence emanates

```
DO I = 1, N
S1':X$(I) = X(I+1)

S1:  A(I) = X$(I) + X(I)
S2:  X(I+1) = B(I) + 32
ENDDO
```

- Recurrence broken
- Vectorized to

```
X$(1:N) = X(2:N+1)
X(2:N+1) = B(1:N) + 32
A(1:N) = X$(1:N) + X(1:N)
```


Node Splitting Algorithm

- Takes a constant loop independent antidependence D
- Add new assignment x: T$=source(D)
- Insert x before source(D)
- Replace source(D) with T$
- Make changes in the dependence graph

Node Splitting

- Determining minimal set of critical antidependences is in NP-C
- Perfect job of Node Splitting difficult
- Heuristic:
  - —Select antidependences
  - —Delete it to see if acyclic
  - —If acyclic, apply Node Splitting

Summary

- Enhancing fine-grained parallelism
  - break dependence cycles
  - pick false dependences
    - caused by memory reuse not value flow
    - can be removed by renaming
  - not dependence cycles can be made acyclic
- Transformations discussed
  -