Improving Register Usage

Parul Halwe-05305002
Sameera Deshpande-05305601
Vibhooti Verma-05305016

11^{th} November, 2005

Why Improve Register Usage ?

- Speed Mismatch of Memory and Processor
- To Reduce Number of Memory Accesses
- To Exploit the Data Dependencies

Methods for Improving Register Usage

- Scalar Register Allocation
- Scalar Replacement
- Unroll and Jam
- Loop Interchange

Register Coloring Technique

This attempts to allocate all of the uses of each scaler variable in a single live range.

1. It identifies the live rangesof variableand gives each a unique name
2. builds interference graph
3. uses fast coloring algorithm with colors equal to no of registers avalable
4. if coloring fails, one varible out of registers. Attempts again.

Example for vector variables

$$
\begin{array}{l}
\quad DO\ I = 1, N \\
\quad\quad DO\ J = 1, M \\
: \quad\quad A[I] = A[I] + B[J] \\
\quad ENDO \\
ENDO
\end{array}
$$

$$
\begin{array}{l}
\quad DO\ I = 1, N \\
\quad\quad T = A(I) \\
\quad\quad DO\ J = 1, M \\
: \quad\quad T = T + B[J] \\
\quad ENDO \\
\quad A[I] = T \\
ENDO
\end{array}
$$

Data Dependence For Register Reuse

True Dependency : Value of source can be kept in register till the sink is reached.

Antidependence : No scope for improving register usage.

Output Dependency Last statement value need to be stored.

Input Depedence :Present an oppertunity to eliminate a load at the second reference.

Loop Carried and Loop Independent Reuse

loop Independent

$$\begin{array}{ll} S_1 & A(I) = \\ S_2 & ... = A(I) \\ S_1 & t = ... \\ S_2 & ... = t \end{array}$$

Loop Carried and Loop Independent Reuse

loop Independent

S_1	$A(I) =$
S_2	$\ldots = A(I)$
S_1	$t = \ldots$
S_2	$\ldots = t$

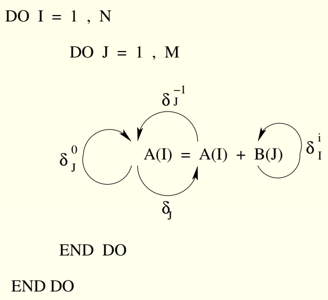
Register Reuse in Loop Carried Dependency

$$\begin{array}{ll} S_1 & \ldots = A(I) \\ S_3 & A(I+1) = \\ S_1 & \ldots = t \\ S_3 & t = \ldots \end{array}$$

Register Reuse in Loop Carried Dependency

$$
\begin{array}{ll}
S_1 & \ldots = A(I) \\
S_3 & A(I+1) = \\
S_1 & \ldots = t \\
S_3 & t = \ldots
\end{array}
$$

A Register Allocation Example



$DO\ I = 1, N$
$DO\ J = 1, M$

$: \qquad A[I] = A[I] + B[J]$

$ENDO$

$DO\ I = 1, N$

$T = A[I]$

$DO\ J = 1, M$

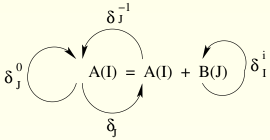
$: \qquad T = T + B[J]$

$ENDO$

A Register Allocation Example

DO I = 1 , N

DO J = 1 , M



END DO

END DO

$DO\ I = 1, N$
$DO\ J = 1, M$

$: \quad A[I] = A[I] + B[J]$

$ENDO$

$DO\ I = 1, N$

$T = A[I]$

$DO\ J = 1, M$

$: \quad T = T + B[J]$

$ENDO$

What is scalar replacement?

- Array references are converted into scalar references.
- It is source to source transformation.

Scalar Replacement Algorithm

The algorithm is roughly divided into 4 phases:

1. Find named partitions using graph pruning
2. Apply register pressure moderation
3. Replace each selected name partition with references to scalars :
    1. If non-cyclic partition, replace it using collection of unique temporaries obtained from loop dependencies.
    2. If cyclic partition, replace it with single temporary as explained in single spanning loop-iteration.
    3. For all other inconsistant dependence, use index set splitting.
4. Unroll the loop to eliminate register to register copies.

Consider the example:

DO I = 1, N
S1: $A(I+1) = A(I-1) + B(I-1)$
S2: $A(I) = A(I) + B(I) + B(I+1)$
ENDO

Pruning the dependence graph

The kind of edges that must be eliminated to prune graph are:

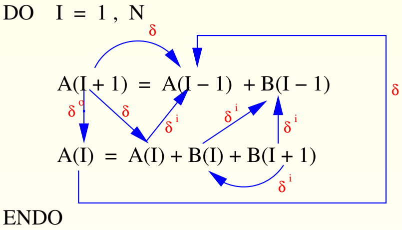
- Write occurs between location involved in between end points of dependence.
  - Flow dependence edges
  - Input dependence edges
- Output dependence edges
- Antidependence edges

Remaining dependence edges are used as measure of memory usage.

Consider the example:

DO I = 1, N
S1: $A(I+1) = A(I-1) \;\; + \;\; B(I-1)$
S2: $A(I) = A(I) \;\; + \;\; B(I) \;\; + \;\; B(I+1)$
ENDO

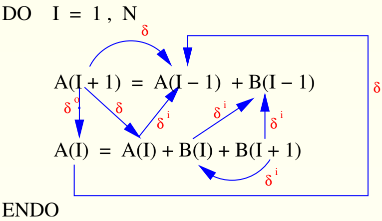
Dependence graph is:



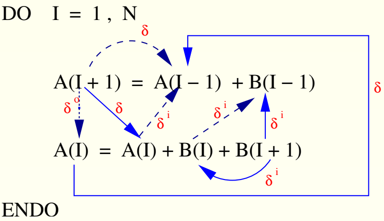
Consider the example:

DO I = 1, N
S1: $A(I+1) = A(I-1) \ + \ B(I-1)$
S2: $A(I) = A(I) \ + \ B(I) \ + \ B(I+1)$
ENDO

Dependence graph is:



Pruned dependence graph is:



Algorithm for pruning edges

Phase 1 : Eliminate Killed Dependences

Phase 2 : Identify *Generators*

Phase 3 : Find *Named Partitions* and Eliminate Input Dependences

Handling Loop-independent Dependences

If dependence is loop-independent dependence, then

- Generate scalar to hold referenced value.
- Replaice second reference with a reference to scalar.

eg:

DO I = 1, N
$S1: \quad A(I) = B(I) + C$
$S2: \quad X(I) = A(I) * Q$
ENDO

DO I = 1, N
. $\quad T = B(I) + C$
. $\quad A(I) = T$
. $\quad X(I) = T * Q$
ENDO

Handling Loop-independent Dependences

If dependence is loop-independent dependence, then

- Generate scalar to hold referenced value.
- Replaice second reference with a reference to scalar.

eg:

DO I = 1, N
$S1: \quad A(I) = B(I) + C$
$S2: \quad X(I) = A(I) * Q$
ENDO

DO I = 1, N
. $\quad T = B(I) + C$
. $\quad A(I) = T$
. $\quad X(I) = T * Q$
ENDO

Handling Loop-independent Dependences

If dependence is loop-independent dependence, then

- Generate scalar to hold referenced value.
- Replaice second reference with a reference to scalar.

eg:

DO I = 1, N
$S1: \quad A(I) = B(I) + C$
$S2: \quad X(I) = A(I) * Q$
ENDO

DO I = 1, N
. $\quad T = B(I) + C$
. $\quad A(I) = T$
. $\quad X(I) = T * Q$
ENDO

Dependences spanning over single loop-iteration

What do I mean by Dependences spanning over single loop-iteration is

- First and Last statements in collection are on same iteration
- OR if they are on succesive iteration then last statement does not occur lexically after first statement.

eg:

DO I = 1, N
$S1: \quad A(I) = B(I-1)$
$S2: \quad B(I) = A(I) + C(I)$
ENDO

$TB = B(0)$
DO I = 1, N
. $TA = TB$
. $A(I) = TA$
. $TB = TA + C(I)$
. $B(I) = TB$
ENDO

Dependences spanning over single loop-iteration

What do I mean by Dependences spanning over single loop-iteration is

- First and Last statements in collection are on same iteration
- OR if they are on succesive iteration then last statement does not occur lexically after first statement.

eg:

DO I = 1, N
$S1: \quad A(I) = B(I-1)$
$S2: \quad B(I) = A(I) + C(I)$
ENDO

$TB = B(0)$
DO I = 1, N
$\quad TA = TB$
$\quad A(I) = TA$
$\quad TB = TA + C(I)$
$\quad B(I) = TB$
ENDO

Dependences spanning over single loop-iteration

What do I mean by Dependences spanning over single loop-iteration is

- First and Last statements in collection are on same iteration
- OR if they are on succesive iteration then last statement does not occur lexically after first statement.

eg:

DO I = 1, N
$S1: \quad A(I) = B(I-1)$
$S2: \quad B(I) = A(I) + C(I)$
ENDO

$TB = B(0)$
DO I = 1, N
. $TA = TB$
. $A(I) = TA$
. $TB = TA + C(I)$
. $B(I) = TB$
ENDO

Dependences spanning over multiple loop-iterations

DO I = 1, N
$S1 : A(I) = B(I-1) + B(I+1)$
ENDO

$T1 = B(0)$
$T2 = B(1)$
DO I = 1, N
. $T3 = B(I+1)$
. $A(I) = T1 + T3$
. $T1 = T2$
. $T2 = T3$
ENDO

Dependences spanning over multiple loop-iterations

DO I = 1, N
$S1 : A(I) = B(I-1) + B(I+1)$
ENDO

$T1 = B(0)$
$T2 = B(1)$
DO I = 1, N
. $T3 = B(I+1)$
. $A(I) = T1 + T3$
. $T1 = T2$
. $T2 = T3$
ENDO

Eliminating Scalar Copies

The code generated by using previous methos does achieve desired reduction in loads, but scalar copy instructions get introduced. To reduce that overhead, we unroll the loop to cycle length of permutations. So, the above code looks like :

$T1 = B(0)$
$T2 = B(1)$
$MN3 = MOD(N, 3)$
DO I = 1, MN3
. $T3 = B(I+1)$
. $A(I) = T1 + T3$
. $T1 = T2$
. $T2 = T3$
ENDO

DO I = MN3 + 1, N, 3
. $T3 = B(I+1)$
. $A(I) = T1 + T3$
. $T1 = B(I+2)$
. $A(I+1) = T2 + T1$
. $T2 = B(I+3)$
. $A(I+2) = T3 + T2$

Register Pressure moderation

- Ideally it is desired that scalar replacement procedure to be applied to all named partitions.
- But restriction on number of floating point registers available.
- Large number of scalars so generated may compete for limited registers, which cannot be handled by scalar register allocators.
- Hence, register pressure moderation is needed.

Scalar Replacement Algorithm

The algorithm is roughly divided into 4 phases:

1. Find named partitions using graph pruning
2. Apply register pressure moderation
3. Replace each selected name partition with references to scalars :
    1. If non-cyclic partition, replace it using collection of unique temporaries obtained from loop dependencies.
    2. If cyclic partition, replace it with single temporary as explained in single spanning loop-iteration.
    3. For all other inconsistant dependence, keep loads and stores as they are.
4. Unroll the loop to eliminate register to register copies.

Definition

Consider the following example .....

$$
\begin{array}{l}
DO \;\; I = 1, N * 2 \\
\quad DO \;\; J = 1, M \\
\quad\quad A(I) = A(I) + B(J) \\
\quad ENDDO \\
ENDDO
\end{array}
$$

Equivalent Code After Unroll-And-Jam Transformation

Following loop is obtained after applying unroll-and-jam transformation on the previous loop:-

$$
\begin{array}{l}
DO \quad I = 1, N * 2, 2 \\
\quad DO \quad J = 1, M \\
\quad\quad A(I) = A(I) + B(J) \\
\quad\quad A(I+1) = A(I+1) + B(J) \\
\quad ENDDO \\
ENDDO
\end{array}
$$

Equivalent Loop After Scalar Replacement

The equivalent loop scalar replacement is:

$$
\begin{array}{l}
DO\ \ I = 1, N*2, 2 \\
\quad s0 = A(I) \\
\quad s1 = A(I+1) \\
\quad DO\ \ J = 1, M \\
\quad\quad t = B(J) \\
\quad\quad s0 = s0 + t \\
\quad\quad s1 = s1 + t \\
\quad ENDDO \\
\quad A(I) = s0 \\
\quad A(I+1) = s1 \\
ENDDO
\end{array}
$$

What has been achieved by this transformation is :-

- This new version of the loop performs only one load for two uses.
- Inner loop runs for less number of times.
- We need only three registers throughout the inner loop.

Definition

So unroll-and-jam can be defined as :
An **Unroll-and-Jam** to factor n consists of unrolling the outer loop $n - 1$ times to to create n copies of the inner loop and fusing those copies together.

Improving Efficiency of Pipelining Functions

Consider another example:

$DO \quad J = 1, M * 2$
$\quad DO \quad I = 1, N$
$\quad\quad A(I, J) = A(I + 1, J) + A(I - 1, J)$
$\quad ENDDO$
$ENDDO$

The equivalent loop after Unroll-and-jam transformation is:

$DO \quad J = 1, M * 2, 2$
$\quad DO \quad I = 1, N$
$\quad\quad A(I, J) = A(I + 1, J) + A(I - 1, J)$
$\quad\quad A(I, J + 1) = A(I + 1, J + 1) + A(I - 1, J + 1)$
$\quad ENDDO$
$ENDDO$

Loop After scalar Replacement

After scalar replacement the loop looks like:

$$
\begin{array}{ll}
 & \quad s0 = A(0,1) \\
 & \quad s1 = A(0,2) \\
 & \quad DO \;\; I = 1, N \\
S1: & \qquad\qquad s0 = A(I+1, J) + s0 \\
S2: & \qquad\qquad s1 = A(I+1, J+1) + s1 \\
 & \qquad A(I,J) = s1 \\
 & \qquad A(I,J+1) = s0 \\
 & \quad ENDDO \\
 & ENDDO
\end{array}
$$

Here..

- The loop next takes half the time to execute.
- The inner loop has two loads and two stores for each pair of floating point operations.

Legality of Unroll-and-Jam

Can this transformation be ever illegal?

Consider the following loop:

$$
\begin{array}{l}
DO\quad I = 1, N*2 \\
\quad DO\quad J = 1, M \\
\quad\quad A(I+1, J-1) = A(I, J) + B(I, J) \\
\quad ENDDO \\
ENDDO
\end{array}
$$

The corresponding unroll-and-jam tranformed loop is:

$$
\begin{array}{l}
DO\quad I = 1, N*2, 2 \\
\quad DO\quad J = 1, M \\
\quad\quad A(I+1, J-1) = A(I, J) + B(I, J) \\
\quad\quad A(I+2, J-1) = A(I+1, J) + B(I+1, J) \\
\quad ENDDO \\
ENDDO
\end{array}
$$

Legality of Unroll-and-Jam

Can this transformation be ever illegal?
Consider the following loop:

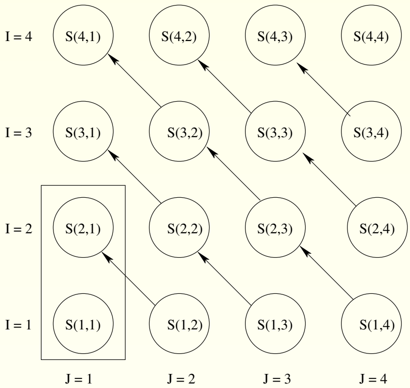
$$
\begin{array}{l}
DO\ \ I = 1, N * 2 \\
\quad DO\ \ J = 1, M \\
\quad\quad A(I+1, J-1) = A(I, J) + B(I, J) \\
\quad ENDDO \\
ENDDO
\end{array}
$$

The corresponding unroll-and-jam tranformed loop is:

$$
\begin{array}{l}
DO\ \ I = 1, N * 2, 2 \\
\quad DO\ \ J = 1, M \\
\quad\quad A(I+1, J-1) = A(I, J) + B(I, J) \\
\quad\quad A(I+2, J-1) = A(I+1, J) + B(I+1, J) \\
\quad ENDDO \\
ENDDO
\end{array}
$$

The Dependence Pattern for Given Example

Following figure shows the dependence pattern for iterations in iteration space.
Assume $N = 2$ and $M = 4$
The unrolling pattern is denoted by the rectangular box:



Why is it Illegal

- The dependence direction vector for this loop is (¡,¿).
- Unroll-and-Jam can be thought of as loop interchange,followed by unrolling inner loop, again followed by loop interchange

So is it always illegal whenever loop interchange is illegal?

Why is it Illegal

- The dependence direction vector for this loop is (¡,¿).
- Unroll-and-Jam can be thought of as loop interchange,followed by unrolling inner loop, again followed by loop interchange

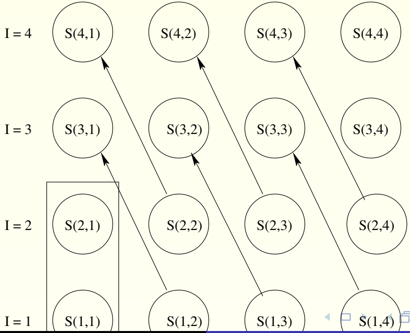
So is it always illegal whenever loop interchange is illegal?

Another Example for Unroll-and-Jam

Consider another example:

$DO \quad I = 1, N * 2$
$\quad DO \quad J = 1, M$
$\quad\quad A(I + 2, J - 1) = A(I, J) + B(I, J)$
$\quad ENDDO$
$ENDDO$

The dependence pattern in this case is:



Theorm 1

Theorm 1: An unroll-and-jam to factor *n* is legal if and only if there exists no dependence with direction vector (¡,¿) such that the distance for the outer loop is ¡ n.

So the unroll-and-jam tranformation to factor *n* is clearly legal if:-

- No dependence with direction vector (¡,¿) exists.
- If such a dependence exists then it must have distance *n* or greater.

Example to Understand the Algorithm

It is a recursive algorithm and takes as input a loop that is to be transformed and unroll factor n. Lets see how the algorithm tranforms the following loop that is not perfectly nested:-

$DO I = 1, N$
  $DO K = 1, N$
    $A(I) = A(I) + X(I, K)$
  $ENDDO$
  $DO J = 1, M$
    $DO K = 1, N$
      $B(J, K) = B(J, K) + A(I)$
    $ENDDO$
  $ENDDO$
  $DO J = 1, M$
    $C(J, I) = B(J, N)/A(I)$
  $ENDDO$
$ENDDO$

Example to Understand the Algorithm

It is a recursive algorithm and takes as input a loop that is to be transformed and unroll factor n. Lets see how the algorithm tranforms the following loop that is not perfectly nested:-

```
DO I = 1, N
  DO K = 1, N
    A(I) = A(I) + X(I, K)
  ENDDO
  DO J = 1, M
    DO K = 1, N
      B(J, K) = B(J, K) + A(I)
    ENDDO
  ENDDO
  DO J = 1, M
    C(J, I) = B(J, N)/A(I)
  ENDDO
ENDDO
```

Example to Understand the Algorithm

It is a recursive algorithm and takes as input a loop that is to be transformed and unroll factor n. Lets see how the algorithm tranforms the following loop that is not perfectly nested:-

$$
\begin{array}{l}
DO I = 1, N \\
\quad DO K = 1, N \\
\quad\quad A(I) = A(I) + X(I, K) \\
\quad ENDDO \\
\quad DO J = 1, M \\
\quad\quad DO K = 1, N \\
\quad\quad\quad B(J, K) = B(J, K) + A(I) \\
\quad\quad ENDDO \\
\quad ENDDO \\
\quad DO J = 1, M \\
\quad\quad C(J, I) = B(J, N)/A(I) \\
\quad ENDDO \\
ENDDO
\end{array}
$$

Loop after First Call to Algorithm UnrollAndJam

$DOI = mN2 + 1, N, 2$
  $DOK = 1, N$
    $A(I) = A(I) + X(I, K)$
    $A(I + 1) = A(I + 1) + X(I + 1, K)$
  $ENDDO$
  $DOJ = 1, M$
    $DOK = 1, N$
      $B(J, K) = B(J, K) + A(I)$
      $B(J, K) = B(J, K) + A(I + 1)$
    $ENDDO$
    $C(J, K) = B(J, N)/A(I)$
    $C(J, I + 1) = B(J, N)/A(I + 1)$
  $ENDDO$
$ENDDO$

Final Transformed Loop

Finally the nested inner J loop will look like:-

$$
\begin{array}{l}
mM2 = MOD(M, 2) \\
DOJ = 1, mM2 \\
\quad DOJ = 1, N \\
\quad\quad B(J, K) = B(J, K) + A(I) \\
\quad\quad B(J, K) = B(J, K) + A(I + 1) \\
\quad ENDDO \\
\quad C(J, I) = B(J, N)/A(I) \\
\quad C(J, I + 1) = B(J, N)/A(I + 1) \\
ENDDO \\
DOJ = nM2 + 1, M, 2 \\
\quad DOK = 1, N \\
\quad\quad B(J, K) = B(J, K) + A(I) \\
\quad\quad B(J, K) = B(J, K) + A(I + 1) \\
\quad\quad B(J + 1, K) = B(J + 1, K) + A(I) \\
\quad\quad B(J + 1, K) = B(J + 1, K) + A(I + 1) \\
\quad ENDDO \\
\quad C(J, I) = B(J, N)/A(I) \\
\quad C(J, I + 1) = B(J, N)/A(I + 1) \\
\quad C(J + 1, I) = B(J + 1, N)/A(I) \\
\quad C(J + 1, I + 1) = B(J + 1, N)/A(I + 1) \\
ENDDO
\end{array}
$$

Loop Interchange for Register Reuse

```
  DO I = 2, N
    DO J = 1, M
:        A(I) = A(J, I - 1)
      ENDO
   ENDO
```


```
  DO I = 2, N
    DO J = 1, M
:        A(I) = A(J, I - 1)
      ENDO
   ENDO
```

Loop Interchange for Register Reuse

$$
: \quad \begin{array}{l} DO\ I = 2, N \\ \quad DO\ J = 1, M \\ \qquad A(I) = A(J, I-1) \\ \quad\ ENDO \\ \ ENDO \end{array}
$$

$$
: \quad \begin{array}{l} DO\ I = 2, N \\ \quad DO\ J = 1, M \\ \qquad A(I) = A(J, I-1) \\ \quad\ ENDO \\ \ ENDO \end{array}
$$

Transformed Loops

$$
\begin{array}{l}
DO\ J = 1, M \\
\quad DO\ I = 2, N \\
: \qquad A(J, I) = A(J, I-1) \\
\qquad ENDO \\
\quad ENDO
\end{array}
$$

$$
\begin{array}{l}
DO\ J = 1, M \\
\quad R_1 = A(J, 1) \\
\quad DO\ I = 2, N \\
: \qquad A(J, I) = R_1 \\
\qquad ENDO \\
\quad ENDO
\end{array}
$$

Transformed Loops

$$
:\quad \begin{array}{l} DO\ J = 1, M \\ \quad DO\ I = 2, N \\ \qquad A(J, I) = A(J, I-1) \\ \quad ENDO \\ ENDO \end{array}
$$

$$
:\quad \begin{array}{l} DO\ J = 1, M \\ \quad R_1 = A(J, 1) \\ \quad DO\ I = 2, N \\ \qquad A(J, I) = R_1 \\ \quad ENDO \\ ENDO \end{array}
$$

Considerations for Loop Interchange

$$
\begin{array}{l}
DO\ J = 1, N \\
\quad DO\ K = 1, N \\
\qquad\qquad DO\ I = 1, 256 \\
\qquad\qquad\quad A(I, J, K) = A(I, J-1, K) + A(I, J-1, K-1) + A(I, J, K-1) \\
\qquad\qquad ENDDO
\end{array}
$$

:

$$
DO\ J = 1, N
$$

Interchanged Loop

$$
\begin{array}{l}
DO\ K = 1, N \\
\quad DO\ I = 1, 256 \\
: \qquad DO\ J = 1\ N \\
: \qquad\quad A(I, J, K) = A(I, J-1, K) + A(I, J-1, K-1) + A(I, J, K-1) \\
: \qquad\quad A(I, J, K+1) = A(I, J-1, K+1) + A(I, J-1, K) + A(I, J, K) \\
\qquad ENDDO \\
\quad ENDO \\
ENDO
\end{array}
$$

Loop After Applying Unroll-and-Jam Tranformation

$DO\ K = 1, N$
$r1\ = A(I, 0, K)$
$r2\ = A(I, 0, K + 1)$
$DO\ I = 1, 256$
: $DO\ J = 1\ N$
: $r0 = r1 + A(I, J - 1, K - 1) + A(I, J, K - 1)$
: $A(I, J, K) = r0$
: $r2 = r2 + r1 + r0$
: $A(I, J, K + 1) = r2$
: $r1 = r0$
$ENDDO$
$ENDO$
$ENDO$

Code After Scalar Replacement

$$
\begin{array}{l}
DO\ I = 1, 100 \\
\quad DO\ J = 1, 50 \\
\qquad DO\ K = 1, N \\
\qquad\quad A(K, J, I) = A(K, J, I-1) + B(K, I) \\
\qquad ENDO \\
\quad ENDO \\
ENDO
\end{array}
$$

Loop Interchange Alogrithm

1. Form the direction matrix for the loop nest and use it to identify the loops other than the scalarization loop that can legally be moved to the innermost position.
2. For each such loop l , let count(l) be thenumber of rows of the direction matrix that have "¡" in the position corresponding to l and "=" every other poditiob
3. pick the loop l that maximizes the product of count(l) and iteration count of loop l.

Conclusion

- More dependencies result in better register usage.
- Better register usage results in better overall performance.
- Since we try to optimize number of reads and stores performed, the running time is reduced significantly.

References I