COMP 515: Advanced Compilation for Vector and Parallel Processors

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

https://wiki.rice.edu/confluence/display/PARPROG/COMP515
Compiler Improvement of Register Usage

Chapter 8 (contd)
Scalar Replacement (Recap)

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

\[
\begin{align*}
\text{DO } I & = 1, N \\
A(I) & = B(I-1) + B(I+1)
\end{align*}
\]

\[
\begin{align*}
t1 & = B(0) \\
t2 & = B(1) \\
\text{DO } I & = 1, N \\
t3 & = B(I+1) \\
A(I) & = t1 + t3 \\
t1 & = t2 \\
t2 & = t3 \\
\text{ENDDO}
\end{align*}
\]

- 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 by unrolling

\[
\begin{align*}
t1 &= B(0) \\
t2 &= B(1) \\
DO \ I &= 1, N \\
\quad t3 &= B(I+1) \\
\quad A(I) &= t1 + t3 \\
\quad t1 &= t2 \\
\quad t2 &= t3 \\
ENDDO
\end{align*}
\]

- Unnecessary register-register copies
- Unroll loop 3 times

\[
\begin{align*}
t1 &= B(0) \\
t2 &= B(1) \\
mN3 &= MOD(N, 3) \\
DO \ I &= 1, mN3 \\
\quad t3 &= B(I+1) \\
\quad A(I) &= t1 + t3 \\
\quad t1 &= t2 \\
\quad t2 &= t3 \\
ENDDO
\end{align*}
\]

\[
\begin{align*}
&DO \ I = mN3 + 1, N, 3 \\
&\quad t3 = B(I+1) \\
&\quad A(I) = t1 + t3 \\
&\quad t1 = B(I+2) \\
&\quad A(I+1) = t2 + t1 \\
&\quad t2 = B(I+3) \\
&\quad A(I+2) = t3 + t2 \\
ENDDO
\end{align*}
\]
Pruning the dependence graph

- Prune all anti dependence edges
- Prune flow and input dependence edges that do not represent a potential reuse
- Prune redundant input dependence edges
- Prune output dependence edges after rest of the pruning is done
Pruning the dependence graph

• Phase 1: Eliminate killed dependences
  – When killed dependence is a flow dependence
    S1: A(I+1) = ...
    S2: A(I)  = ...
    S3: ...  = A(I)
    - Store in S2 is a killing store. Flow dependence from S1 to S3 is pruned
  – When killed dependence is an input dependence
    S1: ... = A(I+1)
    S2: A(I)  = ...
    S3: ...  = A(I-1)
    - Store in S2 is a killing store. Input dependence from S1 to S3 is pruned
Pruning the dependence graph

• Phase 2: Identify *generators*

\[
\begin{align*}
\text{DO } & I = 1, N \\
A(I+1) & = A(I-1) + B(I-1) \\
A(I) & = A(I) + B(I) + B(I+1)
\end{align*}
\]

• Any assignment reference with at least one flow dependence emanating from it to another statement in the loop

• Any use reference with at least one input dependence emanating from it and no input or flow dependence into it
Pruning the dependence graph

- Phase 3: Find name partitions and eliminate input dependences
  - Use Typed Fusion
    - References as vertices
    - An edge joins two references
    - Output and anti- dependences are bad edges
    - Name of array as type

- Eliminate input dependences between two elements of same name partition unless source is a generator
Scalar Replacement: Putting it together

1. Prune dependence graph; Apply typed fusion

2. Select a set of name partitions using register pressure moderation

3. For each selected partition
   A) If non-cyclic, replace using set of temporaries
   B) If cyclic replace reference with single temporary
   C) For each inconsistent dependence
      Use index set splitting or insert loads and stores

4. Unroll loop to eliminate scalar copies
**Unroll-and-Jam**

```plaintext
DO I = 1, N*2
    DO J = 1, M
        A(I) = A(I) + B(J)
    ENDDO
ENDDO

• Can we achieve reuse of references to B?
• Use transformation called Unroll-and-Jam

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

• Unroll outer loop twice and then fuse the copies of the inner loop
• Brought two uses of B(J) together
```
DO I = 1, N*2, 2
    DO J = 1, M
        A(I) = A(I) + B(J)
        A(I+1) = A(I+1) + B(J)
    ENDDO
ENDDO

• Apply scalar replacement on this code

DO I = 1, N*2, 2
    s0 = A(I)
    s1 = A(I+1)
    DO J = 1, M
        t = B(J)
        s0 = s0 + t
        s1 = s1 + t
    ENDDO
    A(I) = s0
    A(I+1) = s1
ENDDO

• Half the number of loads as the original program
Legality of Unroll-and-Jam

• Is unroll-and-jam always legal?

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

• This is wrong!!

• Apply unroll-and-jam
Legality of Unroll-and-Jam

Legality of unroll-and-jam

<table>
<thead>
<tr>
<th>I=4</th>
<th>S(4,1)</th>
<th>S(4,2)</th>
<th>S(4,3)</th>
<th>S(4,4)</th>
</tr>
</thead>
<tbody>
<tr>
<td>I=3</td>
<td>S(3,1)</td>
<td>S(3,2)</td>
<td>S(3,3)</td>
<td>S(3,4)</td>
</tr>
<tr>
<td>I=2</td>
<td>S(2,1)</td>
<td>S(2,2)</td>
<td>S(2,3)</td>
<td>S(2,4)</td>
</tr>
<tr>
<td>I=1</td>
<td>S(1,1)</td>
<td>S(1,2)</td>
<td>S(1,3)</td>
<td>S(1,4)</td>
</tr>
</tbody>
</table>

Legality of unroll-and-jam.

<table>
<thead>
<tr>
<th>I=4</th>
<th>S(4,1)</th>
<th>S(4,2)</th>
<th>S(4,3)</th>
<th>S(4,4)</th>
</tr>
</thead>
<tbody>
<tr>
<td>I=3</td>
<td>S(3,1)</td>
<td>S(3,2)</td>
<td>S(3,3)</td>
<td>S(3,4)</td>
</tr>
<tr>
<td>I=2</td>
<td>S(2,1)</td>
<td>S(2,2)</td>
<td>S(2,3)</td>
<td>S(2,4)</td>
</tr>
<tr>
<td>I=1</td>
<td>S(1,1)</td>
<td>S(1,2)</td>
<td>S(1,3)</td>
<td>S(1,4)</td>
</tr>
</tbody>
</table>

J=1 J=2 J=3 J=4
Legality of Unroll-and-Jam

• Direction vector in this example was (<,>)
  — This makes loop interchange illegal
  — Unroll-and-Jam is loop interchange followed by unrolling inner loop
    followed by another loop interchange

• But does loop interchange illegal imply unroll-and-jam illegal? NO
Legality of Unroll-and-Jam

• Consider this example

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

• Direction vector is (<,>); still unroll-and-jam possible because of distances involved
Conditions for legality of unroll-and-jam

• **Definition:** Unroll-and-jam to factor \( n \) consists of unrolling the outer loop \( n-1 \) times and fusing those copies together.

• **Theorem:** An unroll-and-jam to a factor of \( n \) is legal iff there exists no dependence with direction vector \((<,>)\) such that the distance for the outer loop is less than \( n \).
Unroll-and-jam Algorithm

1. Create preloop
2. Unroll main loop \(m\) (the unroll-and-jam factor) times
3. Apply typed fusion to loops within the body of the unrolled loop
4. Apply unroll-and-jam recursively to the inner nested loop
**Unroll-and-jam example**

\[
\begin{align*}
&\text{DO } I = 1, N \\
&\text{DO } K = 1, N \\
&\quad A(I) = A(I) + X(I,K) \\
&\text{ENDDO} \\
&\text{DO } J = 1, M \\
&\text{DO } K = 1, N \\
&\quad B(J,K) = B(J,K) + A(I) \\
&\text{ENDDO} \\
&\text{ENDDO} \\
&\text{DO } J = 1, M \\
&\text{DO } K = 1, N \\
&\quad C(J,I) = B(J,N)/A(I) \\
&\text{ENDDO} \\
&\text{ENDDO}
\end{align*}
\]

\[
\begin{align*}
&\text{DO } I = mN^2+1, N, 2 \\
&\text{DO } K = 1, N \\
&\quad A(I) = A(I) + X(I,K) \\
&\quad A(I+1) = A(I+1) + X(I+1,K) \\
&\text{ENDDO} \\
&\text{DO } J = 1, M \\
&\text{DO } K = 1, N \\
&\quad B(J,K) = B(J,K) + A(I) \\
&\quad B(J,K) = B(J,K) + A(I+1) \\
&\text{ENDDO} \\
&\text{ENDDO} \\
&\quad C(J,I) = B(J,N)/A(I) \\
&\quad C(J,I+1) = B(J,N)/A(I+1) \\
&\text{ENDDO} \\
&\text{ENDDO}
\end{align*}
\]
Conclusion

• We have learned two memory hierarchy transformations:
  — scalar replacement
  — unroll-and-jam

• They reduce the number of memory accesses by maximum use of processor registers
Solve exercise 8.2 in book

• Hand-transform the following loop nest to achieve high register reuse. What transformations did you use? What is the ratio of floating-point operations to loads before and after the transformation? How many registers did you assume i.e., how many registers do you need?

DO I = 1, N
  DO J = 1, N
    A(I+1,J+1) = A(I,J+1) + A(I+1,J) + B(J)
  END DO
END DO

• Due by 5pm on Tuesday, Nov 24th

• Homework should be submitted in class or to Annepha Pemberton, Duncan Hall 3080
Inter-iteration Scalar Replacement Using Array SSA Form

Rishi Surendran\textsuperscript{1} Rajkishore Barik\textsuperscript{2} Jisheng Zhao\textsuperscript{1} Vivek Sarkar\textsuperscript{1}

\textsuperscript{1}Rice University

\textsuperscript{2}Intel Labs

**Inter-iteration Scalar Replacement Example**

<table>
<thead>
<tr>
<th>Original Loop</th>
<th>After Scalar Replacement</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: for $i = 1$ to $n$ do</td>
<td>1: $t_0 = A[0]$</td>
</tr>
<tr>
<td>3: end for</td>
<td>3: for $i = 1$ to $n$ do</td>
</tr>
<tr>
<td></td>
<td>4: $t_2 = A[i + 1]$</td>
</tr>
<tr>
<td></td>
<td>5: $B[i] = 0.3333 \times (t_0 + t_1 + t_2)$</td>
</tr>
<tr>
<td></td>
<td>6: $t_0 = t_1$</td>
</tr>
<tr>
<td></td>
<td>7: $t_1 = t_2$</td>
</tr>
<tr>
<td></td>
<td>8: end for</td>
</tr>
</tbody>
</table>

- Jacobi-1D kernel from Polybench/C benchmark suite
- The value accessed by the expression $A[i + 1]$ in iteration $k$ is again accessed by expression $A[i]$ in iteration $k + 1$
  - $A[i + 1]$ is the generator for $A[i]$
CURRENT APPROACHES FOR SCALAR REPLACEMENT

• Scalar replacement using non-SSA representations
  ■ David Callahan et.al. [PLDI 1990]
    ▶ Does not handle control flow
    ▶ Requires precise dependence information
  ■ Steve Carr, Ken Kennedy [SPE 1994]
    ▶ Complex: includes 14 different steps such as availability analysis, reachability analysis and anticipability analysis
    ▶ Requires precise dependence information

• Scalar replacement using array SSA form
  ■ Stephen Fink et.al. [SAS 2000]
    ▶ Does not require dependence information
    ▶ Does not handle inter-iteration reuse
CURRENT APPROACHES FOR SCALAR REPLACEMENT

• Scalar replacement using non-SSA representations
  ■ David Callahan et.al. [PLDI 1990]
    ▶ Does not handle control flow
    ▶ Requires precise dependence information
  ■ Steve Carr, Ken Kennedy [SPE 1994]
    ▶ Complex: includes 14 different steps such as availability analysis, reachability analysis and anticipability analysis
    ▶ Requires precise dependence information

• Scalar replacement using array SSA form
  ■ Stephen Fink et.al. [SAS 2000]
    ▶ Does not require dependence information
    ▶ Does not handle inter-iteration reuse

Goal: Inter-iteration scalar replacement using array SSA form
Interstitial Scalar Replacement Steps

- Extended array SSA form construction
- Subscript analysis
  - Available subscript analysis for redundant load elimination
  - Dead subscript analysis for dead store elimination
- Transformations
  - Elimination of loads/store
  - Prolog/epilog code generation
ARRAY SSA FORM AND EXTENSIONS

- Program representation capturing precise element-level data flow information for array variables
- Every use and definition has a unique name
- Four different types of $\phi$ functions
  - **Control $\phi$**
    - Same semantics as scalar SSA phi function
  - **Definition $\phi$ ($d\phi$)**
    - Inserted immediately after a definition
    - Kathleen Knobe, Vivek Sarkar [POPL 98]
  - **Use $\phi$ ($u\phi$)**
    - Inserted immediately after a use
    - Stephen Fink et.al. [SAS 2000]
Array SSA Form and Extensions

- Program representation capturing precise element-level data flow information for array variables
- Every use and definition has a unique name
- Four different types of $\phi$ functions
  - Control $\phi$
    - Same semantics as scalar SSA phi function
  - Definition $\phi (d\phi)$
    - Inserted immediately after a definition
    - Kathleen Knobe, Vivek Sarkar [POPL 98]
  - Use $\phi (u\phi)$
    - Inserted immediately after a use
    - Stephen Fink et.al. [SAS 2000]
  - Header $\phi (h\phi)$
    - Control phi function in the loop header
    - Our extension to handle inter-iteration reuse
## Motivating Example

<table>
<thead>
<tr>
<th>Original Loop</th>
<th>After Scalar Replacement</th>
</tr>
</thead>
<tbody>
<tr>
<td>/* 5 loads and 2 stores/iteration */</td>
<td>/* 1 load and 1 store/iteration */</td>
</tr>
<tr>
<td>1:   for $i = 1$ to $n$ do</td>
<td>1:   $tA_{i-1} = A[0]$</td>
</tr>
<tr>
<td>4:   end for</td>
<td>4:   $tB_i = B[1]$</td>
</tr>
<tr>
<td>5:   for $i = 1$ to $n$ do</td>
<td>5:   for $i = 1$ to $n$ do</td>
</tr>
<tr>
<td>6:   $tA_{i+1} = tA_{i-1} + tB_{i-1}$</td>
<td>6:   $tA_{i+1} = tA_{i-1} + tB_{i-1}$</td>
</tr>
<tr>
<td>7:   $tB_{i+1} = B[i + 1]$</td>
<td>7:   $tB_{i+1} = B[i + 1]$</td>
</tr>
<tr>
<td>8:   $tA_i = tA_i + tB_i + tB_{i+1}$</td>
<td>8:   $tA_i = tA_i + tB_i + tB_{i+1}$</td>
</tr>
<tr>
<td>10:  $tA_{i-1} = tA_i$</td>
<td>10:  $tA_{i-1} = tA_i$</td>
</tr>
<tr>
<td>11:  $tA_i = tA_{i+1}$</td>
<td>11:  $tA_i = tA_{i+1}$</td>
</tr>
<tr>
<td>12:  $tB_{i-1} = tB_i$</td>
<td>12:  $tB_{i-1} = tB_i$</td>
</tr>
<tr>
<td>13:  $tB_i = tB_{i+1}$</td>
<td>13:  $tB_i = tB_{i+1}$</td>
</tr>
<tr>
<td>14:  end for</td>
<td>14:  end for</td>
</tr>
<tr>
<td>15:  $A[n + 1] = tA_{i+1}$</td>
<td>15:  $A[n + 1] = tA_{i+1}$</td>
</tr>
</tbody>
</table>
# Step 1: Extended Array SSA Form

<table>
<thead>
<tr>
<th>Three Address Code</th>
<th>Extended Array SSA form</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: for $i = 1$ to $n$ do</td>
<td>1: $A_0 = ...$</td>
</tr>
<tr>
<td>2: $t_1 = A[i - 1]$</td>
<td>2: $B_0 = ...$</td>
</tr>
<tr>
<td>3: $t_2 = B[i - 1]$</td>
<td>3: for $i = 1$ to $n$ do</td>
</tr>
<tr>
<td>4: $t_3 = t_1 + t_2$</td>
<td>4: $A_1 = \hphi(A_0, A_9)$</td>
</tr>
<tr>
<td>5: $A[i + 1] = t_3$</td>
<td>5: $B_1 = \hphi(B_0, B_7)$</td>
</tr>
<tr>
<td>6: $t_4 = A[i]$</td>
<td>6: $t_1 = A_2[i - 1]$</td>
</tr>
<tr>
<td>7: $t_5 = B[i]$</td>
<td>7: $A_3 = \uphi(A_2, A_1)$</td>
</tr>
<tr>
<td>8: $t_6 = B[i + 1]$</td>
<td>8: $t_2 = B_2[i - 1]$</td>
</tr>
<tr>
<td>9: $t_7 = t_4 + t_5$</td>
<td>9: $B_3 = \uphi(B_2, B_1)$</td>
</tr>
<tr>
<td>10: $t_8 = t_7 + t_6$</td>
<td>10: $t_3 = t_1 + t_2$</td>
</tr>
<tr>
<td>11: $A[i] = t_8$</td>
<td>11: $A_4[i + 1] = t_3$</td>
</tr>
<tr>
<td>12: end for</td>
<td>12: $A_5 = \dphi(A_4, A_3)$</td>
</tr>
<tr>
<td></td>
<td>13: $t_4 = A_6[i]$</td>
</tr>
<tr>
<td></td>
<td>14: $A_7 = \uphi(A_6, A_5)$</td>
</tr>
<tr>
<td></td>
<td>15: $t_5 = B_4[i]$</td>
</tr>
<tr>
<td></td>
<td>16: $B_5 = \uphi(B_4, B_3)$</td>
</tr>
<tr>
<td></td>
<td>17: $t_6 = B_6[i + 1]$</td>
</tr>
<tr>
<td></td>
<td>18: $B_7 = \uphi(B_6, B_5)$</td>
</tr>
<tr>
<td></td>
<td>19: $t_7 = t_4 + t_5$</td>
</tr>
<tr>
<td></td>
<td>20: $t_8 = t_7 + t_6$</td>
</tr>
<tr>
<td></td>
<td>21: $A_8[i] = t_8$</td>
</tr>
<tr>
<td></td>
<td>22: $A_9 = \dphi(A_8, A_7)$</td>
</tr>
<tr>
<td></td>
<td>23: end for</td>
</tr>
</tbody>
</table>
Computes the set of array elements available at each of the $\phi$-functions ($\phi$, $u\phi$, $d\phi$, $h\phi$)

- Available elements: Elements that are read/written in the current or previous $\tau$ iterations
  - $\tau$ is a tuning parameter
- Computes a lattice value for each of the $\phi$-functions
  $$\mathcal{L}(A_j) = \{(i_1, d_1), (i_2, d_2), \ldots\}$$
  - Each ordered pair, $(i_k, d_k)$ represents an available array subscript, $i_k$ and the iteration distance from the generator, $d_k$
    - $i_k$ is a spatial dimension
    - $d_k$ is a temporal dimension
## Data Flow Equations For Available Subscript Analysis

<table>
<thead>
<tr>
<th>SSA Node</th>
<th>Data Flow Equation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A_r = \phi(A_p, A_q)$</td>
<td>$\mathcal{L}(A_r) = \text{JOIN}(\mathcal{L}(A_p), \mathcal{L}(A_q))$</td>
</tr>
<tr>
<td>$A_r = h\phi(A_p, A_q)$</td>
<td>$\mathcal{L}(A_r) = \text{JOIN}(\text{SHIFT}(\mathcal{L}(A_q)), \mathcal{L}(A_p))$</td>
</tr>
<tr>
<td>$A_r = d\phi(A_p, A_q)$</td>
<td>$\mathcal{L}(A_r) = \text{INSERT}(\mathcal{L}(A_p), \mathcal{L}(A_q))$</td>
</tr>
<tr>
<td>$A_r = u\phi(A_p, A_q)$</td>
<td>$\mathcal{L}(A_r) = \text{UPDATE}(\mathcal{L}(A_p), \mathcal{L}(A_q))$</td>
</tr>
</tbody>
</table>
Data Flow Equations for Available Subscript Analysis

<table>
<thead>
<tr>
<th>SSA Node</th>
<th>Data Flow Equation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A_r = \phi(A_p, A_q)$</td>
<td>$\mathcal{L}(A_r) = \text{JOIN} (\mathcal{L}(A_p), \mathcal{L}(A_q))$</td>
</tr>
<tr>
<td>$A_r = h\phi(A_p, A_q)$</td>
<td>$\mathcal{L}(A_r) = \text{JOIN} (\text{SHIFT}(\mathcal{L}(A_q)), \mathcal{L}(A_p))$</td>
</tr>
<tr>
<td>$A_r = d\phi(A_p, A_q)$</td>
<td>$\mathcal{L}(A_r) = \text{INSERT} (\mathcal{L}(A_p), \mathcal{L}(A_q))$</td>
</tr>
<tr>
<td>$A_r = u\phi(A_p, A_q)$</td>
<td>$\mathcal{L}(A_r) = \text{UPDATE} (\mathcal{L}(A_p), \mathcal{L}(A_q))$</td>
</tr>
</tbody>
</table>

Definitely Same and Definitely Different Analyses

Given two expressions $a$ and $b$

- $\mathcal{DS}(a, b) = true$, if $a$ and $b$ are guaranteed to have the same value
- $\mathcal{DD}(a, b) = true$, if $a$ and $b$ are guaranteed to have different values
**DATA FLOW EQUATIONS FOR AVAILABLE SUBSCRIPT ANALYSIS**

<table>
<thead>
<tr>
<th>SSA Node</th>
<th>Data Flow Equation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A_r = \phi(A_p, A_q)$</td>
<td>$\mathcal{L}(A_r) = \text{JOIN}(\mathcal{L}(A_p), \mathcal{L}(A_q))$</td>
</tr>
<tr>
<td>$A_r = h\phi(A_p, A_q)$</td>
<td>$\mathcal{L}(A_r) = \text{JOIN}(\text{SHIFT}(\mathcal{L}(A_q)), \mathcal{L}(A_p))$</td>
</tr>
<tr>
<td>$A_r = d\phi(A_p, A_q)$</td>
<td>$\mathcal{L}(A_r) = \text{INSERT}(\mathcal{L}(A_p), \mathcal{L}(A_q))$</td>
</tr>
<tr>
<td>$A_r = u\phi(A_p, A_q)$</td>
<td>$\mathcal{L}(A_r) = \text{UPDATE}(\mathcal{L}(A_p), \mathcal{L}(A_q))$</td>
</tr>
</tbody>
</table>

JOIN: Finds the intersection of 2 sets

- $\text{JOIN}(A, B) = \{(i_1, d) | (i_1, d_1) \in A \text{ and } \exists (i'_1, d'_1) \in B \text{ and } DS(i_1, i'_1) = true \text{ and } d = \max(d_1, d'_1)\}$
DATA FLOW EQUATIONS FOR AVAILABLE SUBSCRIPT ANALYSIS

<table>
<thead>
<tr>
<th>SSA Node</th>
<th>Data Flow Equation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A_r = \phi(A_p, A_q)$</td>
<td>$\mathcal{L}(A_r) = \text{JOIN}(\mathcal{L}(A_p), \mathcal{L}(A_q))$</td>
</tr>
<tr>
<td>$A_r = h\phi(A_p, A_q)$</td>
<td>$\mathcal{L}(A_r) = \text{JOIN}(\text{SHIFT}(\mathcal{L}(A_q)), \mathcal{L}(A_p))$</td>
</tr>
<tr>
<td>$A_r = d\phi(A_p, A_q)$</td>
<td>$\mathcal{L}(A_r) = \text{INSERT}(\mathcal{L}(A_p), \mathcal{L}(A_q))$</td>
</tr>
<tr>
<td>$A_r = u\phi(A_p, A_q)$</td>
<td>$\mathcal{L}(A_r) = \text{UPDATE}(\mathcal{L}(A_p), \mathcal{L}(A_q))$</td>
</tr>
</tbody>
</table>

SHIFT: Shifts all the elements by one iteration.

- $\text{SHIFT}((i_1, d_1), (i_2, d_2), \ldots) = ((i_1 - step_1, d_1 + 1), (i_2 - step_2, d_2 + 1), \ldots)$

  - $step_1, step_2, \ldots$ are the coefficients of the induction variable
**Step 3: Redundant Load Elimination**

- **Identifying redundant loads**
  - A load, $A_i[x]$ is redundant if $x$ is available along the incoming definition
    - $A_k = u\phi(A_i, A_j)$
    - $\exists (y, d) \in \mathcal{L}(A_j)$, $d$ is the reuse distance
    - $DS(x, y) = true$

- **Transformation**
  - Replace load $A_i[x]$ with a scalar temporary read $tA_x$
  - Insert initialization of scalar temporaries in loop preheader
  - Insert statement $tA_x := tA_{x+step}$ at the end of loop body
    - Number of copy statements = reuse distance
# Loop After Load Elimination

<table>
<thead>
<tr>
<th>Original Loop</th>
<th>After Load Elimination</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: for $i = 1$ to $n$ do</td>
<td>1: $tA_{i-1} = A[0]$</td>
</tr>
<tr>
<td>4: end for</td>
<td>4: $tB_i = B[1]$</td>
</tr>
<tr>
<td>5: for $i = 1$ to $n$ do</td>
<td>6: $tA_{i+1} = tA_{i-1} + tB_{i-1}$</td>
</tr>
<tr>
<td>6: $A[i + 1] = tA_{i+1}$</td>
<td>7: $A[i + 1] = tA_{i+1}$</td>
</tr>
<tr>
<td>7: $tA_i = tA_i + tB_i + tB_{i+1}$</td>
<td>8: $tB_{i+1} = B[i + 1]$</td>
</tr>
<tr>
<td>8: $tB_i = tB_i$</td>
<td>9: $tA_i = tA_i$</td>
</tr>
<tr>
<td>10: $tA_{i-1} = tA_i$</td>
<td>11: $tA_{i-1} = tA_i$</td>
</tr>
<tr>
<td>11: $tA_i = tA_{i+1}$</td>
<td>12: $tA_i = tA_{i+1}$</td>
</tr>
<tr>
<td>12: $tB_{i-1} = tB_i$</td>
<td>13: $tB_{i-1} = tB_i$</td>
</tr>
<tr>
<td>13: $tB_i = tB_{i+1}$</td>
<td>14: $tB_i = tB_{i+1}$</td>
</tr>
<tr>
<td>14: end for</td>
<td>15: end for</td>
</tr>
</tbody>
</table>
Experimental Setup

- Implemented in LLVM 3.2
  - Scalar evolution is used to perform subscript analysis
  - Basic alias analysis
  - Subscript analysis is run for 5 iterations ($\tau = 5$)
- 32-core 3.55 GHz IBM Power7
  - 256 GB memory
  - SUSE Linux
- Stencil based benchmarks (sequential)
  - Jacobi variants, Rician Denoising
  - Unroll-and-jam by 4 on Jacobi 2D 5-point
Reduction in Number of Loads

O3 : LLVM -O3
O3SR : LLVM -O3 with scalar replacement
4.6 - 37.8% reduction in number of loads with scalar replacement
Speedup up to $2.29 \times$ compared to LLVM -O3
SUMMARY

• Extensions to array SSA form for inter-iteration reuse analysis
• Subscript analysis for identifying redundant loads and dead stores
• Transformation algorithms for redundant load elimination and dead store elimination
• Performance improvement up to $2.29 \times$ compared to LLVM -O3