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

COMP 515 Lecture 16 3 November, 2015

Compiler Improvement of Register Usage

Chapter 8 (contd)

Scalar Replacement (Recap)

- 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 by unrolling

```
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

Preloop

Main Loop

```
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
```

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

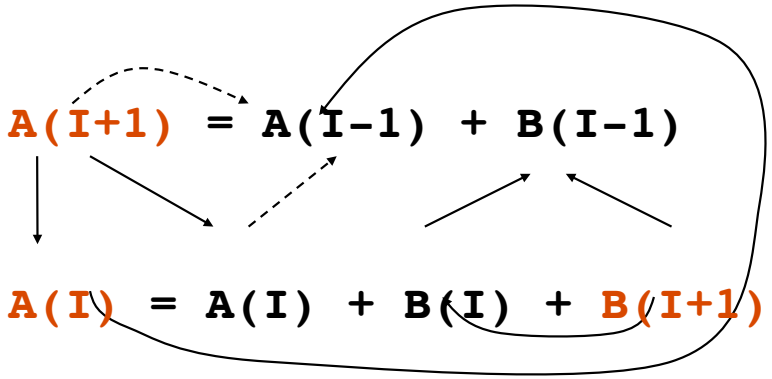
```
S1: ... = A(I+1)

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*

```
DO I = 1, N
```




```
ENDDO
```

- 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

```
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

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
```

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

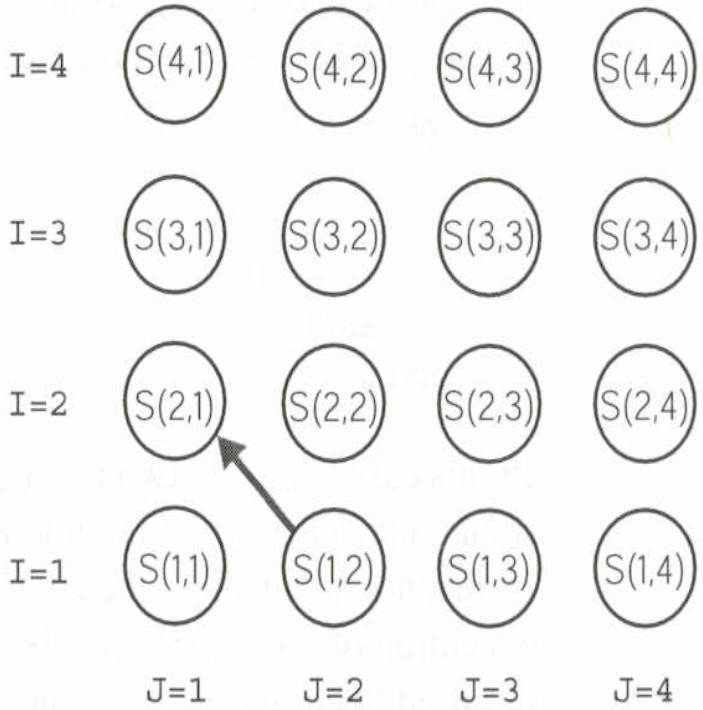
- Apply unroll-and-jam

```
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
```

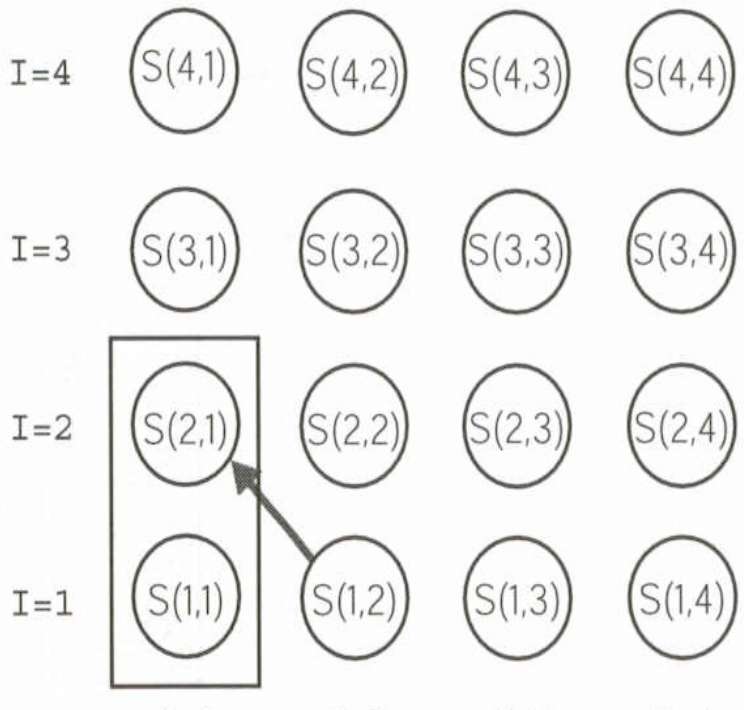
- This is wrong!!!

Legality of Unroll-and-Jam

Legality of unroll-and-jam



Legality of unroll-and-jam.



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

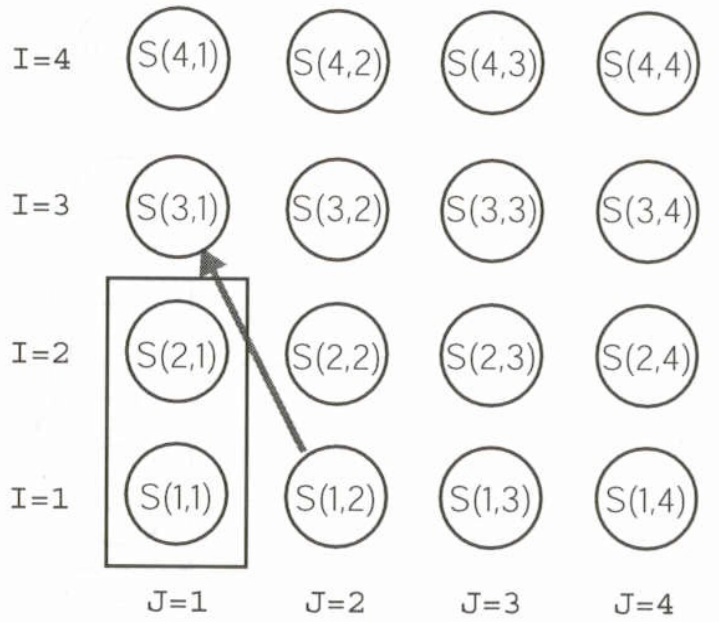
Legality of Unroll-and-Jam

- 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

Legality of unroll-and-jam.



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

```
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
```

```
DO I = mN2+1, N, 2
 DO K = 1, N
  A(I) = A(I) + X(I,K)
  A(I+1) = A(I+1) + X(I+1,K)
 ENDDO
 DO J = 1, M
   DO K = 1, N
     B(J,K) = B(J,K) + A(I)
     B(J,K) = B(J,K) + A(I+1)
   ENDDO
   C(J,I) = B(J,N)/A(I)
   C(J,I+1) = B(J,N)/A(I+1)
 ENDDO
ENDDO
```

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

Homework #4 (Written Assignment)

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[1] Rajkishore Barik[2] Jisheng Zhao[1] Vivek Sarkar[1]

[1]Rice University

[2]Intel Labs

Inter-iteration Scalar Replacement Using Array SSA Form. Rishi Surendran, Rajkishore Barik, Jisheng Zhao, Vivek Sarkar. The 23rd International Conference on Compiler Construction (CC 2014), April 2014.

Inter-iteration Scalar Replacement Example

Original Loop	After Scalar Replacement
1: **for** $i = 1$ to n **do** 2: $\quad B[i] = 0.3333 * (A[i-1] + A[i] + A[i+1])$ 3: **end for**	1: $t_0 = A[0]$ 2: $t_1 = A[1]$ 3: **for** $i = 1$ to n **do** 4: $\quad t_2 = A[i+1]$ 5: $\quad B[i] = 0.3333 * (t_0 + t_1 + t_2)$ 6: $\quad t_0 = t_1$ 7: $\quad t_1 = t_2$ 8: **end for**

- 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

Inter-iteration 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 ϕ functions
  - Control ϕ
    - Same semantics as scalar SSA phi function
  - Definition ϕ ($d\phi$)
    - Inserted immediately after a definition
    - Kathleen Knobe, Vivek Sarkar [POPL 98]
  - Use ϕ ($u\phi$)
    - 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 ϕ functions
  - Control ϕ
    - Same semantics as scalar SSA phi function
  - Definition ϕ ($d\phi$)
    - Inserted immediately after a definition
    - Kathleen Knobe, Vivek Sarkar [POPL 98]
  - Use ϕ ($u\phi$)
    - Inserted immediately after a use
    - Stephen Fink et.al. [SAS 2000]
  - Header ϕ ($h\phi$)
    - Control phi function in the loop header
    - Our extension to handle inter-iteration reuse

Motivating Example

Original Loop	After Scalar Replacement
/* 5 loads and 2 stores/iteration */	/* 1 load and 1 store/iteration */
1: **for** $i = 1$ to n **do**	1: $tA_{i-1} = A[0]$
2: $A[i+1] = A[i-1] + B[i-1]$	2: $tA_i = A[1]$
3: $A[i] = A[i] + B[i] + B[i+1]$	3: $tB_{i-1} = B[0]$
4: **end for**	4: $tB_i = B[1]$
	5: **for** $i = 1$ to n **do**
	6: $tA_{i+1} = tA_{i-1} + tB_{i-1}$
	7: $tB_{i+1} = B[i+1]$
	8: $tA_i = tA_i + tB_i + tB_{i+1}$
	9: $A[i] = tA_i$
	10: $tA_{i-1} = tA_i$
	11: $tA_i = tA_{i+1}$
	12: $tB_{i-1} = tB_i$
	13: $tB_i = tB_{i+1}$
	14: **end for**
	15: $A[n+1] = tA_{i+1}$

Step 1: Extended Array SSA Form

Three Address Code	Extended Array SSA form
1: **for** $i = 1$ to n **do**	1: $A_0 = \ldots$
2: $t_1 = A[i-1]$	2: $B_0 = \ldots$
3: $t_2 = B[i-1]$	3: **for** $i = 1$ to n **do**
4: $t_3 = t_1 + t_2$	4: $A_1 = h\phi(A_0, A_9)$
5: $A[i+1] = t_3$	5: $B_1 = h\phi(B_0, B_7)$
6: $t_4 = A[i]$	6: $t_1 = A_2[i-1]$
7: $t_5 = B[i]$	7: $A_3 = u\phi(A_2, A_1)$
8: $t_6 = B[i+1]$	8: $t_2 = B_2[i-1]$
9: $t_7 = t_4 + t_5$	9: $B_3 = u\phi(B_2, B_1)$
10: $t_8 = t_7 + t_6$	10: $t_3 = t_1 + t_2$
11: $A[i] = t_8$	11: $A_4[i+1] = t_3$
12: **end for**	12: $A_5 = d\phi(A_4, A_3)$
	13: $t_4 = A_6[i]$
	14: $A_7 = u\phi(A_6, A_5)$
	15: $t_5 = B_4[i]$
	16: $B_5 = u\phi(B_4, B_3)$
	17: $t_6 = B_6[i+1]$
	18: $B_7 = u\phi(B_6, B_5)$
	19: $t_7 = t_4 + t_5$
	20: $t_8 = t_7 + t_6$
	21: $A_8[i] = t_8$
	22: $A_9 = d\phi(A_8, A_7)$
	23: **end for**

Step 2: Available Subscript Analysis

Computes the set of array elements available at each of the ϕ-functions (ϕ, $u\phi$, $d\phi$, $h\phi$)

- Available elements: Elements that are read/written in the current or previous τ iterations
  - τ is a tuning parameter
- Computes a lattice value for each of the ϕ-functions $\mathcal{L}(A_j) = \{(i_1, d_1), (i_2, d_2), ...\}$
  - 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

SSA Node	Data Flow Equation
$A_r = \phi(A_p, A_q)$	$\mathcal{L}(A_r) = \text{JOIN}(\mathcal{L}(A_p), \mathcal{L}(A_q))$
$A_r = h\phi(A_p, A_q)$	$\mathcal{L}(A_r) = \text{JOIN}(\text{SHIFT}(\mathcal{L}(A_q)), \mathcal{L}(A_p))$
$A_r = d\phi(A_p, A_q)$	$\mathcal{L}(A_r) = \text{INSERT}(\mathcal{L}(A_p), \mathcal{L}(A_q))$
$A_r = u\phi(A_p, A_q)$	$\mathcal{L}(A_r) = \text{UPDATE}(\mathcal{L}(A_p), \mathcal{L}(A_q))$

Data Flow Equations For Available Subscript Analysis

SSA Node	Data Flow Equation
$A_r = \phi(A_p, A_q)$	$\mathcal{L}(A_r) = \text{JOIN}(\mathcal{L}(A_p), \mathcal{L}(A_q))$
$A_r = h\phi(A_p, A_q)$	$\mathcal{L}(A_r) = \text{JOIN}(\text{SHIFT}(\mathcal{L}(A_q)), \mathcal{L}(A_p))$
$A_r = d\phi(A_p, A_q)$	$\mathcal{L}(A_r) = \text{INSERT}(\mathcal{L}(A_p), \mathcal{L}(A_q))$
$A_r = u\phi(A_p, A_q)$	$\mathcal{L}(A_r) = \text{UPDATE}(\mathcal{L}(A_p), \mathcal{L}(A_q))$

Definitely Same and Definitely Different Analyses

Given two expressions a and b

- $\mathcal{DS}(a, b) = \mathit{true}$, if a and b are guaranteed to have the same value
- $\mathcal{DD}(a, b) = \mathit{true}$, if a and b are guaranteed to have different values

Data Flow Equations For Available Subscript Analysis

SSA Node	Data Flow Equation
$A_r = \phi(A_p, A_q)$	$\mathcal{L}(A_r) = \text{JOIN}(\mathcal{L}(A_p), \mathcal{L}(A_q))$
$A_r = h\phi(A_p, A_q)$	$\mathcal{L}(A_r) = \text{JOIN}(\text{SHIFT}(\mathcal{L}(A_q)), \mathcal{L}(A_p))$
$A_r = d\phi(A_p, A_q)$	$\mathcal{L}(A_r) = \text{INSERT}(\mathcal{L}(A_p), \mathcal{L}(A_q))$
$A_r = u\phi(A_p, A_q)$	$\mathcal{L}(A_r) = \text{UPDATE}(\mathcal{L}(A_p), \mathcal{L}(A_q))$

JOIN: Finds the intersection of 2 sets

- $\text{JOIN}(A, B)) = \{(i_1, d) | (i_1, d_1) \in A$ and $\exists\ (i'_1, d'_1) \in B$ and $\mathcal{DS}(i_1, i'_1) =$ *true* and $d = max(d_1, d'_1)\}$

Data Flow Equations For Available Subscript Analysis

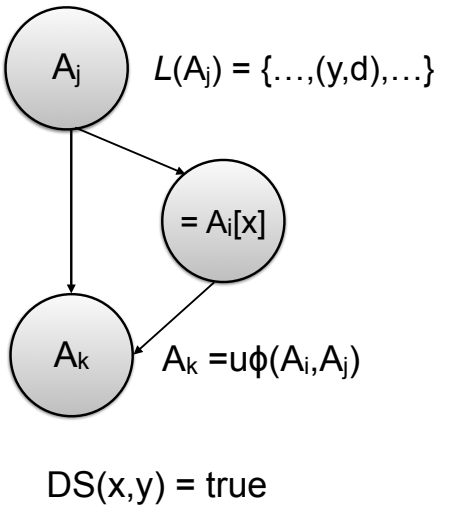
SSA Node	Data Flow Equation
$A_r = \phi(A_p, A_q)$	$\mathcal{L}(A_r) = \text{JOIN}(\mathcal{L}(A_p), \mathcal{L}(A_q))$
$A_r = h\phi(A_p, A_q)$	$\mathcal{L}(A_r) = \text{JOIN}(\text{SHIFT}(\mathcal{L}(A_q)), \mathcal{L}(A_p))$
$A_r = d\phi(A_p, A_q)$	$\mathcal{L}(A_r) = \text{INSERT}(\mathcal{L}(A_p), \mathcal{L}(A_q))$
$A_r = u\phi(A_p, A_q)$	$\mathcal{L}(A_r) = \text{UPDATE}(\mathcal{L}(A_p), \mathcal{L}(A_q))$

SHIFT: Shifts all the elements by one iteration.

- $\text{SHIFT}(\{(i_1, d_1), (i_2, d_2), \ldots\}) = \{(i_1 - step_1, d_1 + 1), (i_2 - step_2, d_2 + 1), \ldots\}$
  - $step_1$, $step_2$, .. 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
    - $\mathcal{DS}(x, y) = true$
- 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



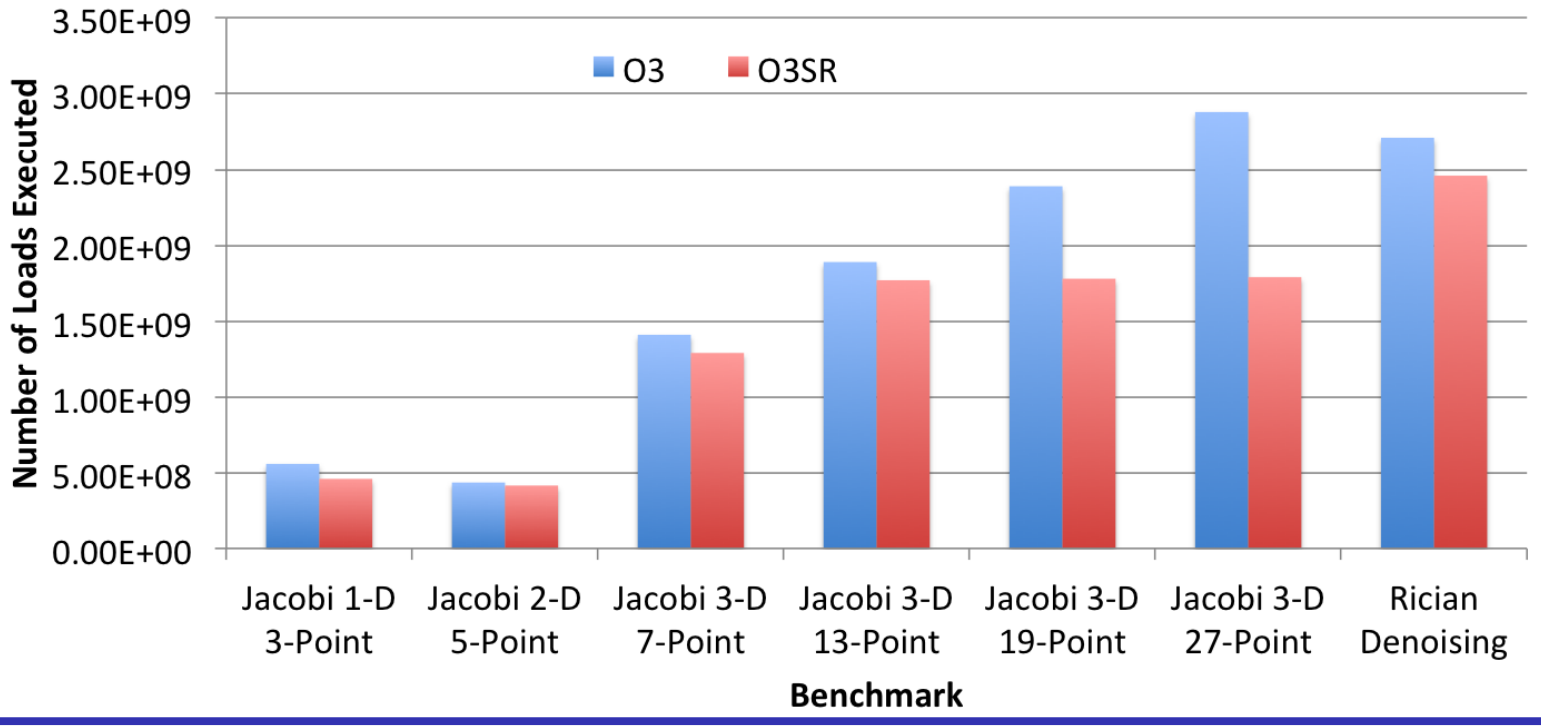
Loop After Load Elimination

Original Loop	After Load Elimination
1: **for** $i = 1$ to n **do** 2: $\quad A[i+1] = A[i-1] + B[i-1]$ 3: $\quad A[i] = A[i] + B[i] + B[i+1]$ 4: **end for**	1: $tA_{i-1} = A[0]$ 2: $tA_i = A[1]$ 3: $tB_{i-1} = B[0]$ 4: $tB_i = B[1]$ 5: **for** $i = 1$ to n **do** 6: $\quad tA_{i+1} = tA_{i-1} + tB_{i-1}$ 7: $\quad A[i+1] = tA_{i+1}$ 8: $\quad tB_{i+1} = B[i+1]$ 9: $\quad tA_i = tA_i + tB_i + tB_{i+1}$ 10: $\quad A[i] = tA_i$ 11: $\quad tA_{i-1} = tA_i$ 12: $\quad tA_i = tA_{i+1}$ 13: $\quad tB_{i-1} = tB_i$ 14: $\quad tB_i = tB_{i+1}$ 15: **end for**

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

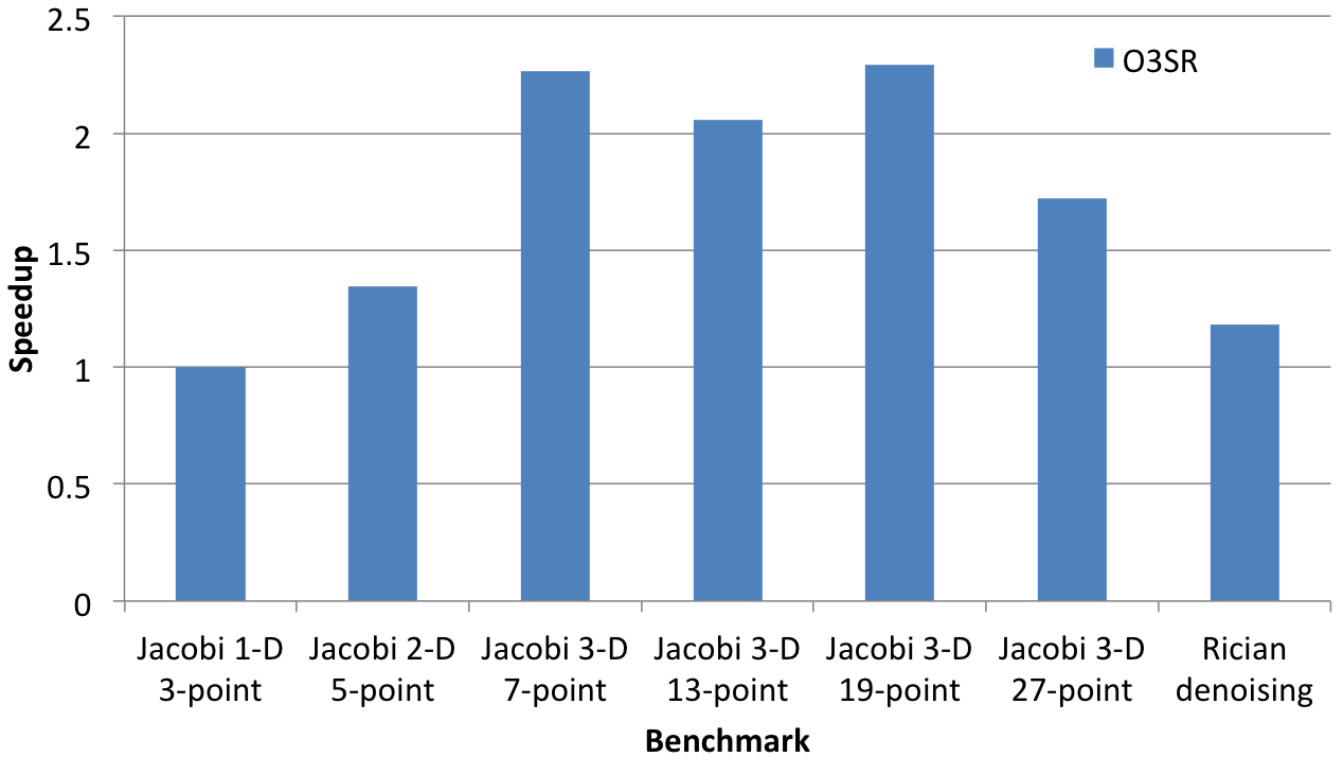


O3 : LLVM -O3

O3SR : LLVM -O3 with scalar replacement

4.6 - 37.8% reduction in number of loads with scalar replacement

SPEEDUP



Speedup up to 2.29× 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× compared to LLVM -O3