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 17 1 November, 2011

Acknowledgments

- Slides from previous offerings of COMP 515 by Prof. Ken Kennedy
  - http://www.cs.rice.edu/~ken/comp515/

Compiler Improvement of Register Usage

Chapter 8 (contd)

Simplified View of Scalar Replacement Algorithm (Section 8.3.7)

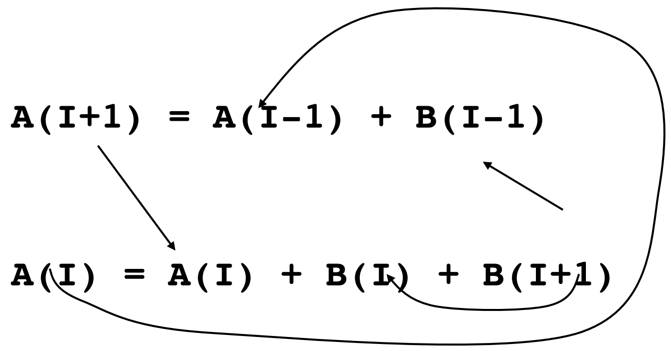
Simplifying assumptions:

- No control flow in loop body
- Other loop transformations (interchange, alignment, fusion, unroll-and-jam, index set splitting) have been performed as a pre-pass and can be ignored here
- Ignore register pressure issues at this stage

High-level Algorithm:

1. Prune dependence graph for scalar replacement
2. Apply “typed fusion” to partition dependence graph into “name partitions” (each partition is a candidate for sharing a scalar variable)
3. For each selected partition
    - A) If non-cyclic, replace using set of temporaries
    - B) If cyclic replace with single temporary
    - C) Insert loads and stores for each inconsistent dependence

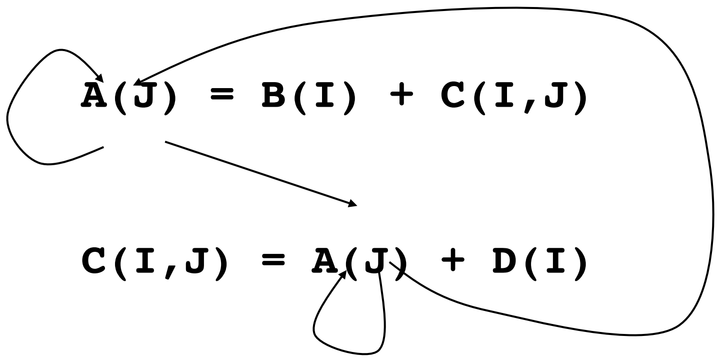
Scalar Replacement: Case A



```
DO I = 1, N

      A(I+1) = A(I-1) + B(I-1)

      A(I) = A(I) + B(I) + B(I+1)

ENDDO
```

```
t0A = A(0); t1A0 = A(1);
tB1 = B(0); tB2 = B(1)
DO I = 1, N
        t1A1 = t0A + tB1
        tB3 = B(I+1)
        t0A = t1A0 + tB3 + tB2
        A(I) = t0A
        t1A0 = t1A1
        tB1 = tB2
        tB2 = tB3
ENDDO
A(N+1) = t1A1
```

Scalar Replacement: Case B



```
DO I = 1, N

        A(J) = B(I) + C(I,J)

        C(I,J) = A(J) + D(I)

ENDDO
```

replace with single temporary...

```
DO I = 1, N
        tA = B(I) + C(I,J)
        C(I,J) = tA + D(I)
ENDDO
A(J) = tA
```

Pruning the Dependence Graph for Scalar Replacement (Section 8.3.1)

<u>Goal:</u>

- Only retain dependence edges that represent a possible elimination of a load and/or a store operation via scalar replacement

NOTE: pruned dependence graph is reference-level (not statement-level)

<u>Pruning Algorithm:</u>

Phase 0: Start with dependence graph containing only flow and input dependences (remove all anti and output dependences)

Phase 1: Eliminate all killed dependences (dependences with a killing store between source and destination)

Phase 2: Identify generators. A generator is a reference with no incoming input/flow dependence and at least one outgoing input/flow dependence.

Phase 3: Find name partitions and eliminate input dependences within partitions. Start with each generator, and mark each reference reachable from the generator in pruned dependence graph as part of that partition. (Typed fusion algorithm can be used for this phase).

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

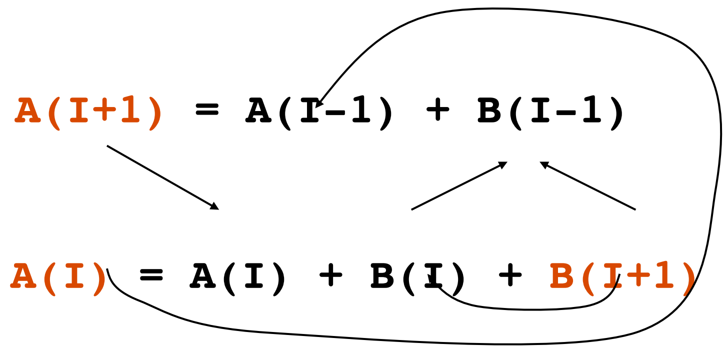
- When killed dependence is an input dependence

```
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

Phase 2: Identify Generators

- Generators are identified below in red



```
DO I = 1, N

        A(I+1) = A(I-1) + B(I-1)

        A(I) = A(I) + B(I) + B(I+1)

ENDDO
```

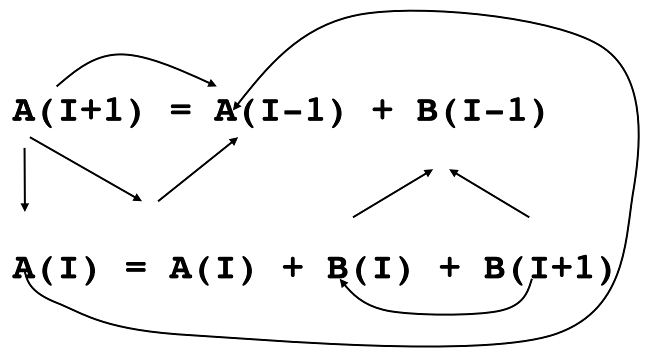
- Any assignment reference with at least one flow dependence emanating from it to another statement in the loop
- Any use reference with at least one input dependence emanating from it and no input or flow dependence into it

Phase 3: Find Name Partitions

- Find name partitions and eliminate input dependences
  - Use Typed Fusion
    - References as vertices
    - Pruned (flow/input) dependence edges are fusible edges
    - Output and anti- dependences are bad edges
    - Name of array as type
- Clean-up: Eliminate input dependences between two elements of same name partition unless source is a generator

Example of Phases 0, 1, 2

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

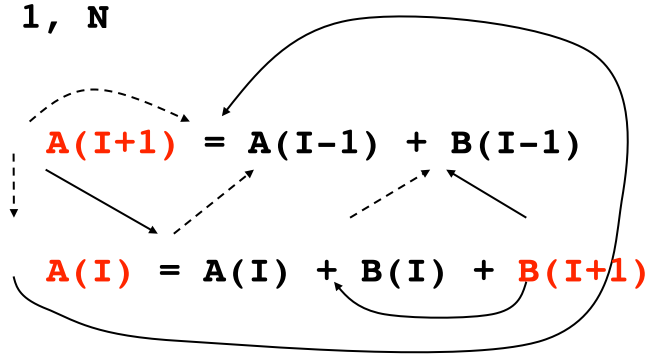


```
A(I+1) = A(I-1) + B(I-1)

A(I) = A(I) + B(I) + B(I+1)
```

```
ENDDO
```

- Dependence pattern before pruning (including input dependences)
- Not all edges suggest memory access savings

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

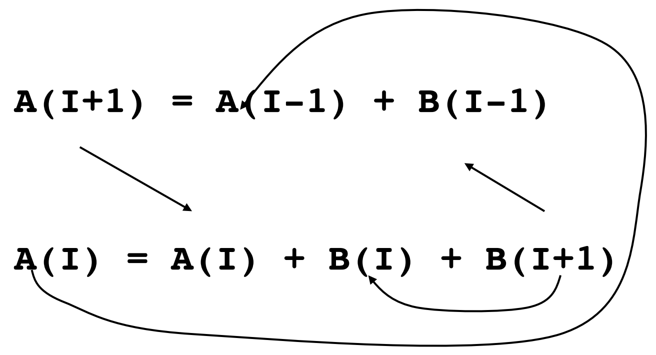


```
A(I+1) = A(I-1) + B(I-1)

A(I) = A(I) + B(I) + B(I+1)
```

```
ENDDO
```

- Dashed edges are pruned
- Each reference has at most one predecessor in the pruned graph
- Generator = source of edge in pruned graph

Scalar Replacement for Previous Example



```
DO I = 1, N

      A(I+1) = A(I-1) + B(I-1)

      A(I) = A(I) + B(I) + B(I+1)

ENDDO
```

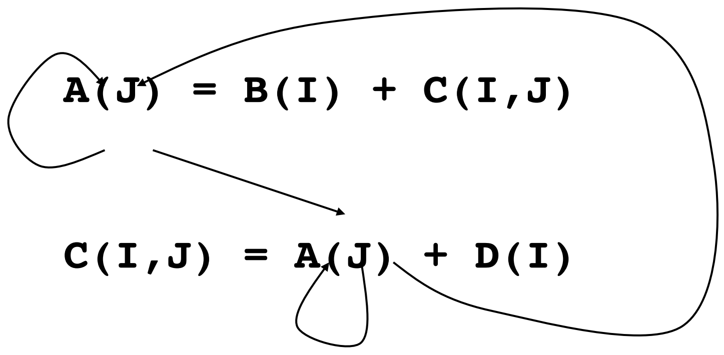
- Apply scalar replacement after pruning the dependence graph
- Needs special-case handling of loop-carried dependences (to be discussed later)

```
t0A = A(0); t1A0 = A(1);
tB1 = B(0); tB2 = B(1);
DO I = 1, N
        t1A1 = t0A + tB1
        tB3 = B(I+1)
        t0A = t1A0 + tB2 + tB3
        A(I) = t0A
        t1A0 = t1A1
        tB1 = tB2
        tB2 = tB3
ENDDO
A(N+1) = t1A1
```

- Only one load and one store per iteration

Complication 1 (pg 390): Handling Dependence Cycle in Loop

– Reference is in a dependence cycle in the loop, and can be replaced by a single scalar variable



```
DO I = 1, N

      A(J) = B(I) + C(I,J)

      C(I,J) = A(J) + D(I)

ENDDO
```

- Assign single scalar to the reference in the cycle
- Replace A(J) by a scalar tA and insert A(J)=tA before or after the loop depending on upward/downward exposed occurrence

Complication 2: Inconsistent Dependences w/ non-constant threshold (pp. 390-391)

- Special cases: Inconsistent dependences

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

- Store to A(J) kills A(I)

- Only one scalar replacement possible

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

- This code can be improved substantially by index set splitting

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

Managing Cache

Allen and Kennedy, Chapter 9

Introduction

- Register
  - One word per register (typically, but there may be exceptions e.g., SIMD registers)
  - Temporal reuse
  - Direct store
  - Eviction (spills) managed by software

- Cache
  - Multiple words in a cache line, multiple lines in an associative set, multiple sets in a cache
  - Temporal and Spatial reuse
  - Load before store
  - Eviction managed by hardware (software can also help)

Spatial Reuse

- Permits high reuse when accessing closely located data

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

  No reuse/locality for Fortran's column-major layout

Spatial Reuse

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

  Iterates over columns instead

Temporal Reuse

- Reuse limited by cache size, LRU replacement strategy

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

Temporal Reuse

- Strip mining + Interchange (or Tiling) can improve temporal reuse when tile size S is chosen so that inner loops can fit in cache

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