Scalar Replacement (Objectives)

- To be able to alleviate the memory bottleneck through scalar replacement
- To be able to reason about the pitfalls of scalar replacement

Removing array references

- What does temporal reuse imply?
  - reuse of a value (true or input dependence)

```
DO  i=2,N
    a(i) = a(i-1)+10
ENDDO
```

```
a0 = a(1);
DO  i=2,N
    a0 = a0+10
    a(i) = a0;
ENDDO
```

- How can we take advantage of this?
  - keep value in register between uses or definition and use
- This is called *scalar replacement* or *register pipelining*
- We count on register assignment to put all scalar values in registers

Scalar Replacement Algorithm

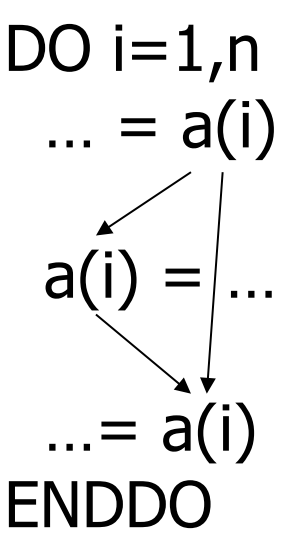
- Replace innermost loop temporal reuse with sequence of scalar temporaries -- *scalar replacement*
- **Method**
  1. Prune dependence graph
  2. Determine number of registers required
  3. Replace references
  4. Insert copies
  5. Code motion
  6. Initialization
  7. Unroll to eliminate copies

Pruning Overview

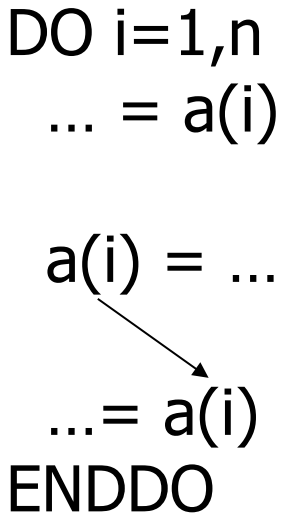
- Must have distance vector describing dependence
  - guarantee when reuse occurs
  - called a *consistent* dependence
- **Problem:** dependence graph does not take into account that a value may be **killed**
- **Solution:** prune the graph so that it represents the **true** flow of values
  - Only represent guaranteed value flow
  - Perform scalar replacement on this graph

Why Graph Pruning?

- Dependence graph does not represent true flow of values



We want:

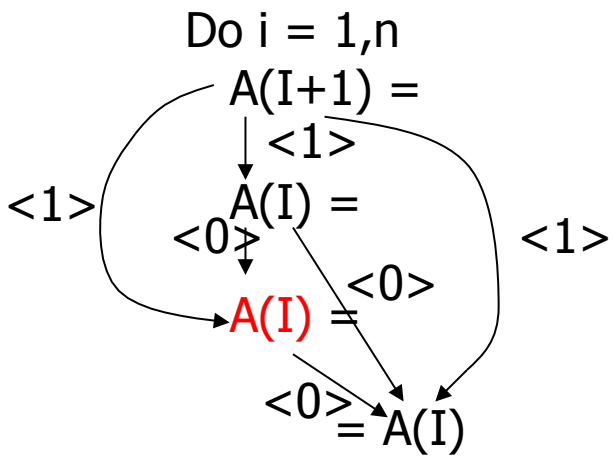


Pruning the Dependence Graph

- What do we need?
  - determine for each array reference which other reference **generates** the value used
  - a reference generates a value if it defines it or it is the first to load it from memory
  - called a *generator* (leader)
- How do we get this?
  - If there is a true dependence, find the **most recent definition**
  - otherwise, find the **least recent use**

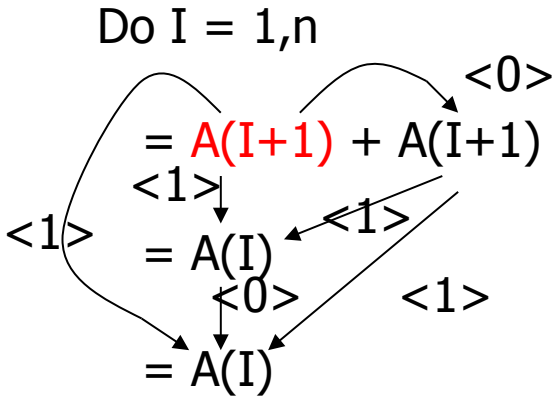
Most Recent Definition

- The most recent definition will have the **smallest** dependence distance
- The most recent definition will **not** have a loop independent outgoing output dependence to a definition that has an outgoing true dependence with the same distance



Least Recent Use

- Consider multiple previous uses
- The least recent use will have the largest dependence distance
- The least recent use will have no incoming loop independent dependence



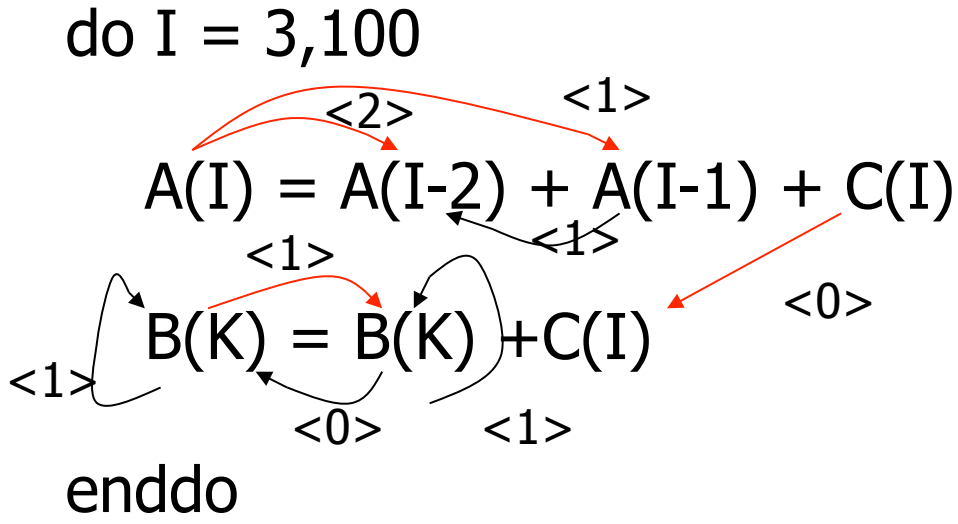
Algorithm

- An edge is **valid** if it is loop independent (source and sink in same loop body) or is innermost-loop carried

for each $v \in V$ *consider v's incoming dependences*
  if $\exists$ an incoming true dependence then
    T = {e | e is a true dependence with the smallest distance & **valid**(e)}
    e_g = e | $e \in$ T and the source of e does not have an outgoing loop-independent dependence whose sink has a true dependence in T
  else
    T = {e | e is an input dependence with the largest distance and valid(e)}
    e_g = e | $e \in T$ and the source of e does not have an incoming loop-independent consistent input dependence

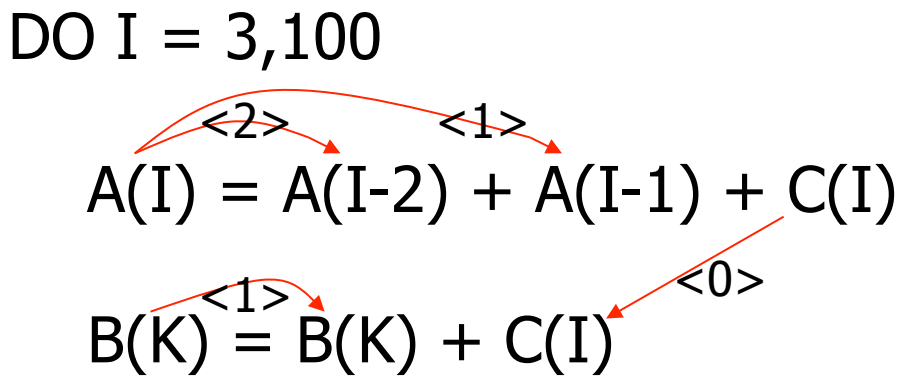
  for each incoming edge e
    if $e \neq e_g$ and e is not an output dependence then
      remove e from the graph
  if e_g is inconsistent remove it
end

Pruning Example



Registers Required

- Source of each true or input dependence is a generator
- Each generator requires $d_n(e) + 1$ registers where e is the edge with the largest distance



How many registers needed for each generator?
Special case, B(K) needs just 1 register

Reference Replacement

- register names - array$generator#$$d_n(e)$
- insert load before use generators

```
DO I = 3, 100
   C$1$0 = C(I)
   A(I) = A(I-2) + A(I-1) + C(I)
   B(K) = B(K) + C(I)
ENDDO
```

Reference Replacement

- insert store after def generators

```
DO I = 3, 100
   C$1$0 = C(I)
   A(I) = A(I-2) + A(I-1) + C(I)
   A(I) = A$0$0
   B(K) = B(K) + C(I)
   B(K) = B$2$0
ENDDO
```

Reference Replacement

- replace references with appropriate register name

```
DO I = 3, 100
   C$1$0 = C(I)
   A$0$0 = A$0$2 + A$0$1 + C$1$0
   A(I) = A$0$0
   B$2$0 = B$2$0 + C$1$0
   B(K) = B$2$0
ENDDO
```

Insert copies

- move value up one register number at end of loop body to keep the value correct across loop iterations

```
DO I = 3, 100
   C$1$0 = C(I)
   A$0$0 = A$0$2 + A$0$1 + C$1$0
   A(I) = A$0$0
   B$2$0 = B$2$0 + C$1$0
   B(K) = B$2$0
   A$0$2 = A$0$1
   A$0$1 = A$0$0
ENDDO
```

Code Motion

- Move invariant loads and stores out of loop nest

```
DO I = 3, 100
   C$1$0 = C(I)
   A$0$0 = A$0$2 + A$0$1 + C$1$0
   A(I) = A$0$0
   B$2$0 = B$2$0 + C$1$0
   A$0$2 = A$0$1
   A$0$1 = A$0$0
ENDDO
B(K) = B$2$0
```

Code Motion

- BE CAREFUL!!!

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

```
DO I = 1, N
    DO J = 1, N
      A$0$0= A$0$0 + A(J)
      A(I) = A$0$0
    ENDDO
ENDDO
```

Only move consistent dependences

Initialization

- peel max of all registers required by generators, minus 1
- replace the sink of pruned dependences with temporary array$generator#$$d_n(e)$, on peeled iteration j *iff* $d_n(e) < j$

```
! Iteration 1; I =3
C$1$0 = C(3)
A$0$0 = A(1) + A(2) + C$1$0
A(3) = A$0$0
B$2$0 = B(K) + C$1$0
A$0$2 = A$0$1
A$0$1 = A$0$0
! Iteration 2; I =4
C$1$0 = C(4)
A$0$0 = A(2) + A$0$1+ C$1$0
A(4) = A$0$0
B$2$0 = B$2$0 + C$1$0
A$0$2 = A$0$1
A$0$1 = A$0$0
```

```
DO I = 5, 100
  C$1$0 = C(I)
  A$0$0 = A$0$2 + A$0$1 + C$1$0
  A(I) = A$0$0
  B$2$0 = B$2$0 + C$1$0
  A$0$2 = A$0$1
  A$0$1 = A$0$0
ENDDO
B(K) = B$2$0
```

Loop Unrolling

- can remove copies in loop
- unroll loop by the maximum of all register requirements,
  $R = max(regs(n))$.
- for each new loop body, L_i, $1 \leq i \leq R\text{-}1$, rename register:
  replace reference array$generator#$$d_n(e)$ with
  array$generator#(($$d_n(e)$-$i$) *mod* R)

Example

```
DO I = 5, 100, 3
   C$1$0 = C(I)
   A$0$0 = A$0$2 + A$0$1 + C$1$0
   A(I) = A$0$0
   B$2$0 = B$2$0 + C$1$0
   C$1$0 = C(I+1)
   A$0$2 = A$0$1 + A$0$0 + C$1$0
   A(I+1) = A$0$2
   B$2$0 = B$2$0 + C$1$0
   C$1$0 = C(I+2)
   A$0$1 = A$0$0 + A$0$2 + C$1$0
   A(I+2) = A$0$1
   B$2$0 = B$2$0 + C$1$0
 ENDDO
 B(K) = B$2$0
```