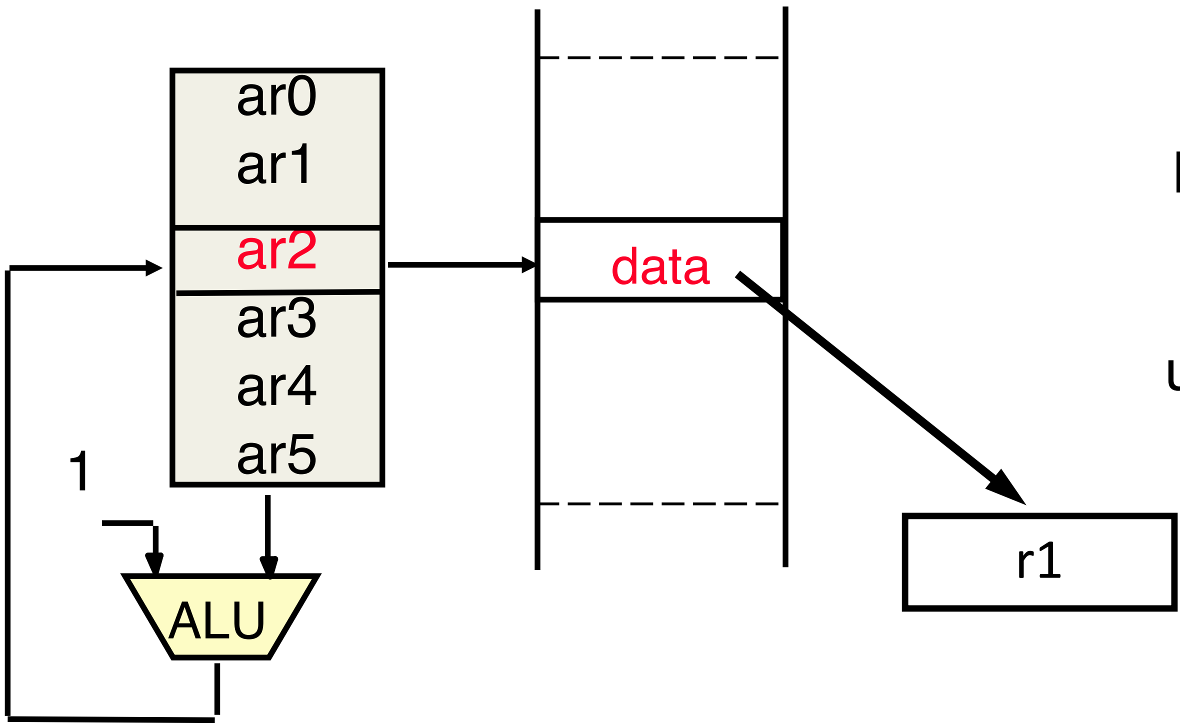
Array Reference Allocation

Guido Araujo

Auto-increment/decrement Modes

- Indirect addressing using auto-increment /decrement.
  - Data locality.
  - Available in the ISA of most embedded processors.



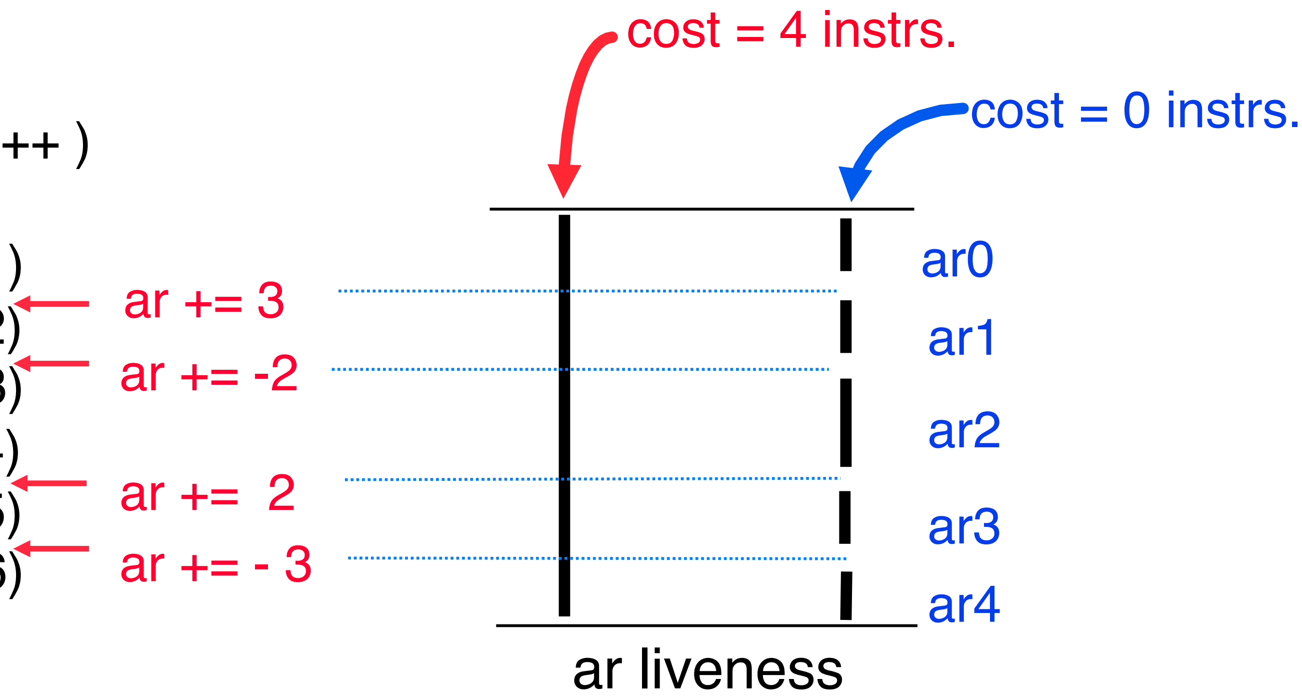
ld r1 , [ar2++]

update ar : ar2 += 3

Local Array Reference Allocation

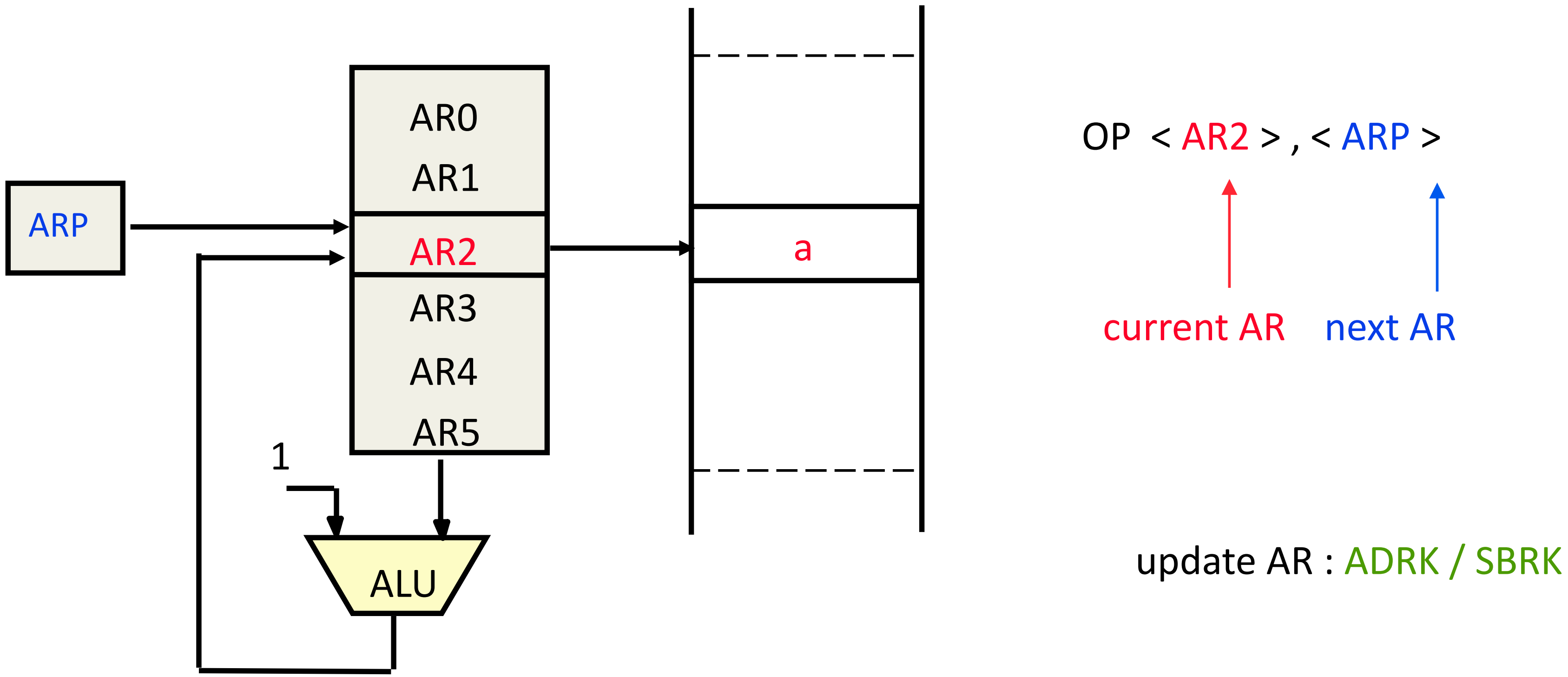
- Problem:
  - Given a sequence of array references in a loop.
  - Determine an allocation for Address Registers (ar's) such as to minimize the number of ar's and update instructions required.

```
for ( i = 2;  i < N - 2;  i++ )
{
    a [ i  - 2]  = 0      (1)
    a [ i + 1]  = 0       (2)
    a [ i  - 1]  = 0      (3)
    a [ i ]       = 0     (4)
    a [ i + 2]  = 0       (5)
    a [ i  - 1]  = 0      (6)
}
```



Addressing Using Address Registers

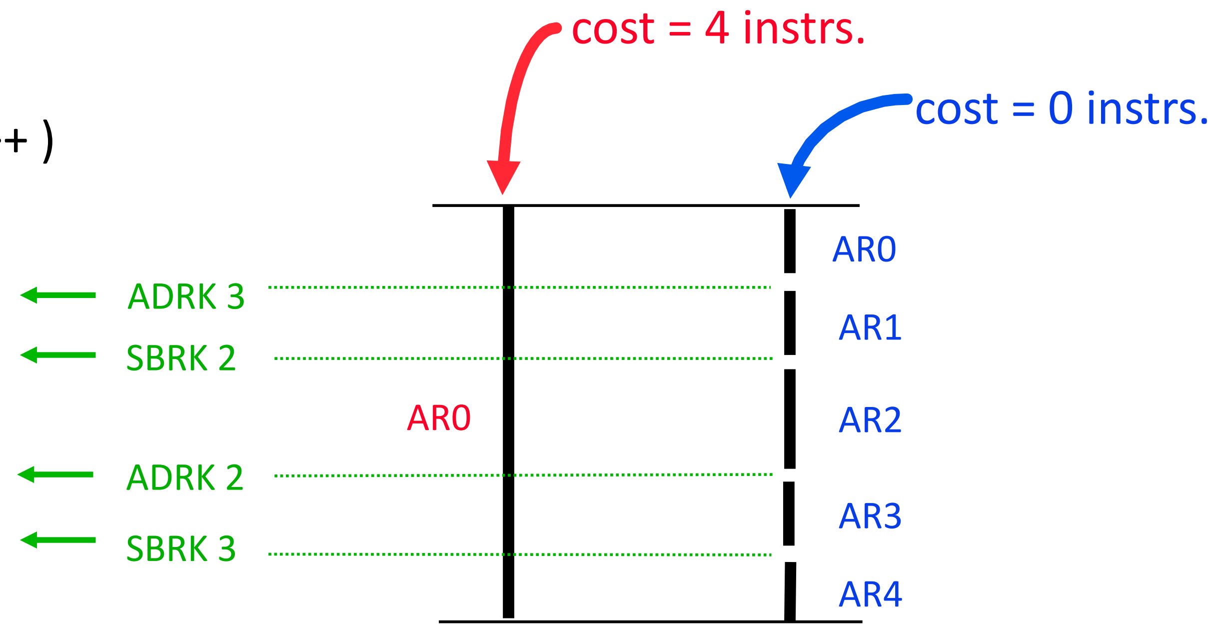
- Indirect addressing using auto-increment /decrement.
  - Computing address is expensive.
  - Sequential access of data-streams is common



Array Index Allocation

- Problem:
  - ❑ Given a sequence of array accesses within a loop.
  - ❑ Determine an allocation for Address Registers (ARs) such as to minimize the number of ARs used and instructions ADRK/SBRK required.

```
for ( i = 2;  i < N - 2;  i++ )
{
   a [ i  - 2]  = 0      (1)
   a [ i + 1]  = 0      (2)
   a [ i  - 1]  = 0      (3)
   a [ i ]      = 0      (4)
   a [ i + 2]  = 0      (5)
   a [ i  - 1]  = 0      (6)
}
```



ARs liveness

Indexing Distance

- Motivation:
  - ❑ Maximize advantage of auto-increment/decrement feature.
  - ❑ Ability to use it is limited by the <u>indexing distance</u>.

```
for ( i = 2;  i < N - 2;  i++ )
{
   a [ i  - 2]        (1)
   a [ i + 1]         (2)
   a [ i  - 1]        (3)
   a [ i ]            (4)
   a [ i + 2]         (5)
   a [ i  - 1]        (6)
}
```

– distance 2 → 4
  - index (2) = i + 1 and index (4) = i
  - d (2,4) = | i - (i + 1) | = 1

– distance 3 → 5
  - index (3) = i - 1 and index (5) = i + 2
  - d (3,5) = | (i + 2) - (i - 1) | = 3

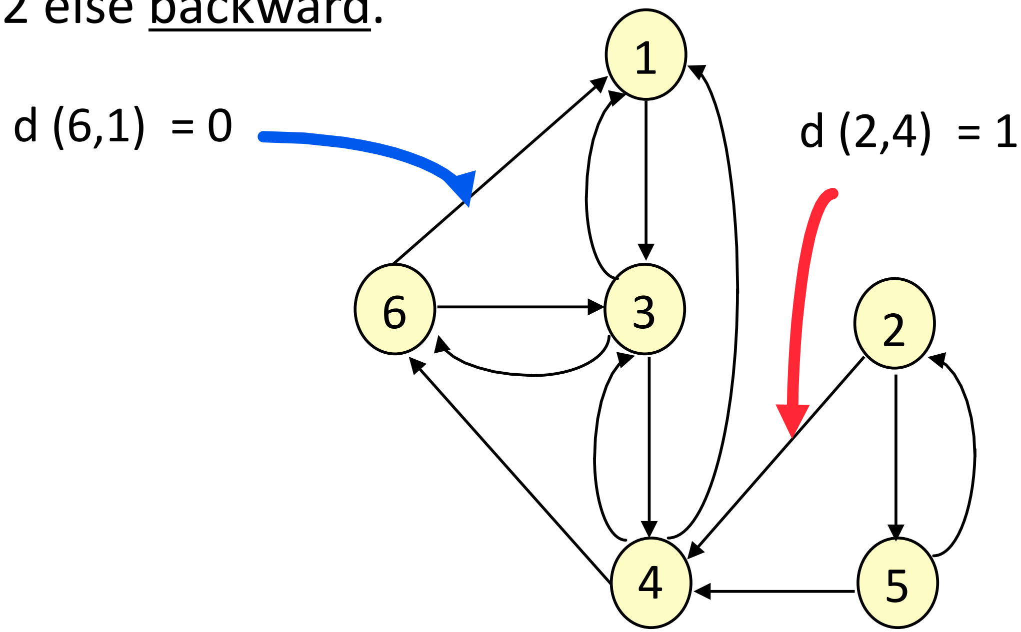
– distance 6 → 1
  - index (6) = i - 1 and index (1) = i - 2
  - d (6,1) = | (i - 2) + 1 - (i - 1) | = 0

consecutive iterations

Indexing Graph (IG)

- Goal
  - Identify long access sequences that can utilize auto-increment/ decrement.
- Indexing Graph:
  - There exists a node n for each array access.
  - There exists an edge (n1,n2) iff d (n1,n2) < = 1.
  - Edge is <u>forward</u> if n1 < n2 else <u>backward</u>.

```
for ( i = 2;  i < N - 2;  i++ )
{
   a [ i  - 2]        (1)
   a [ i + 1]         (2)
   a [ i  - 1]        (3)
   a [ i ]            (4)
   a [ i + 2]         (5)
   a [ i -  1]        (6)
}
```



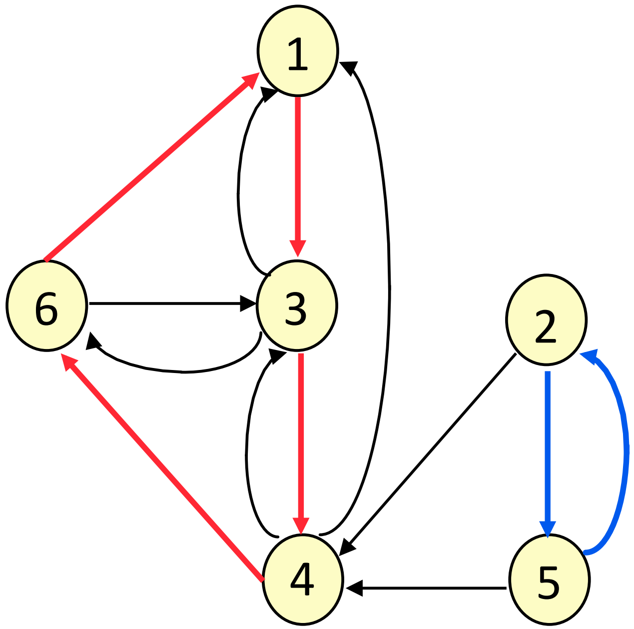
Minimum Disjoint Cycle/Path Covering Problem

- Constraint: cycles with exactly one backward edge.
- All accesses on a cycle/path can use the same AR with auto-inc./dec.
- Problem: Determine a minimum cardinality cycle/path covering of the IG.

```
for ( i = 2;  i < N - 2;  i++ )
{
    a [ i  - 2]        (1)
    a [ i + 1]         (2)
    a [ i  - 1]        (3)
    a [ i ]            (4)
    a [ i + 2]         (5)
    a [ i  - 1]        (6)
}
```

AR0

AR1



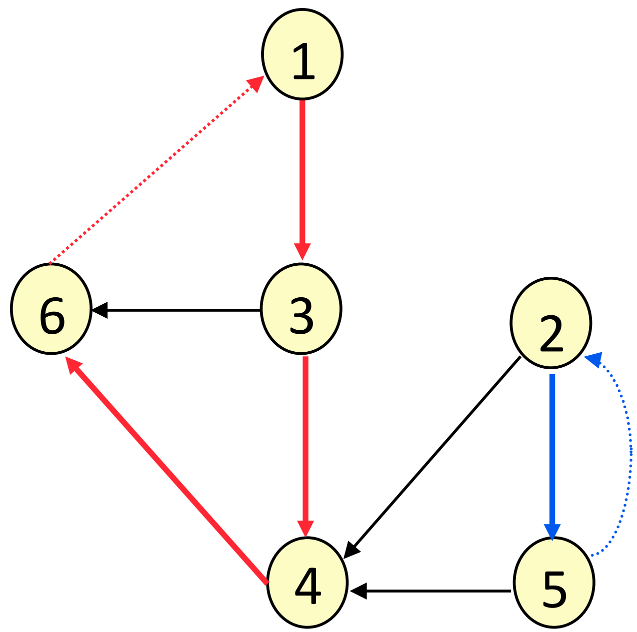
Minimum Disjoint Path Covering Heuristic

- Heuristic:
  - Determine the minimum disjoint-path covering of the IG.
  - Used in Conexant Systems compiler

```
for ( i = 2;  i < N - 2;  i++ )
{
   a [ i  - 2]        (1)
   a [ i + 1]         (2)
   a [ i  - 1]        (3)
   a [ i ]            (4)
   a [ i + 2]         (5)
   a [ i  - 1]        (6)
}
```

AR0

AR1



Number of ARs used : 2

Number of ADRK/SBRK instructions required: 0

Global Array Reference Allocation

```
for (i = 1; i < N-1; i++) {
  avg = a[i] >> 2;
  if (i % 2) {
    avg += a[i-1] << 2;
    a[i] = avg * 3;
  }
  if (avg < error)
    avg -= a[i+1] - error/2;
}
```

```
p = &a[1];
for (i = 1; i < N-1; i++) {
  avg = *p++ >> 2;
  if (i % 2) {
    p += -2;
    avg += *p++ << 2;
    *p++ = avg * 3;
  }
  if (avg < error)
    avg -= *p - error/2;
}
```

Prior Art

- Offset Assignment Problem.
  - Bartley [1992]
  - Liao et al [1995]
  - Leupers et al [1998]
  - Rao et al [1999]
  - Eckstein [1999]
- Array Reference Allocation.
  - Araujo et al [1996]
  - Gebotys [1997]
  - Leupers et al [1998]

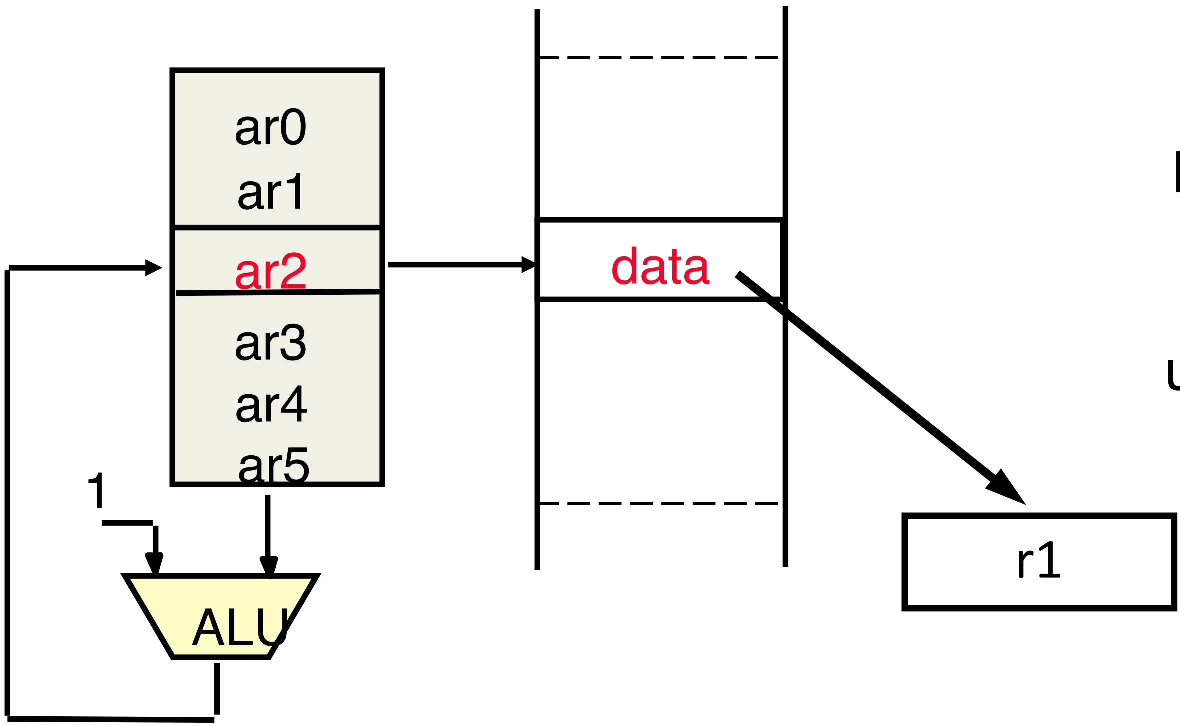
Indirect Addressing

- Address computation is expensive.
  - One every six instructions.
  - 50% of the program bits.

- Indirect addressing is suitable to embedded processors.
  - Implements fast address computation.
  - Enables the design of short instructions.
  - Saves slots during compaction in a VLIW processor.

Auto-increment/decrement Modes

- Indirect addressing using auto-increment /decrement.
  - Data locality.
  - Available in the ISA of most embedded processors.



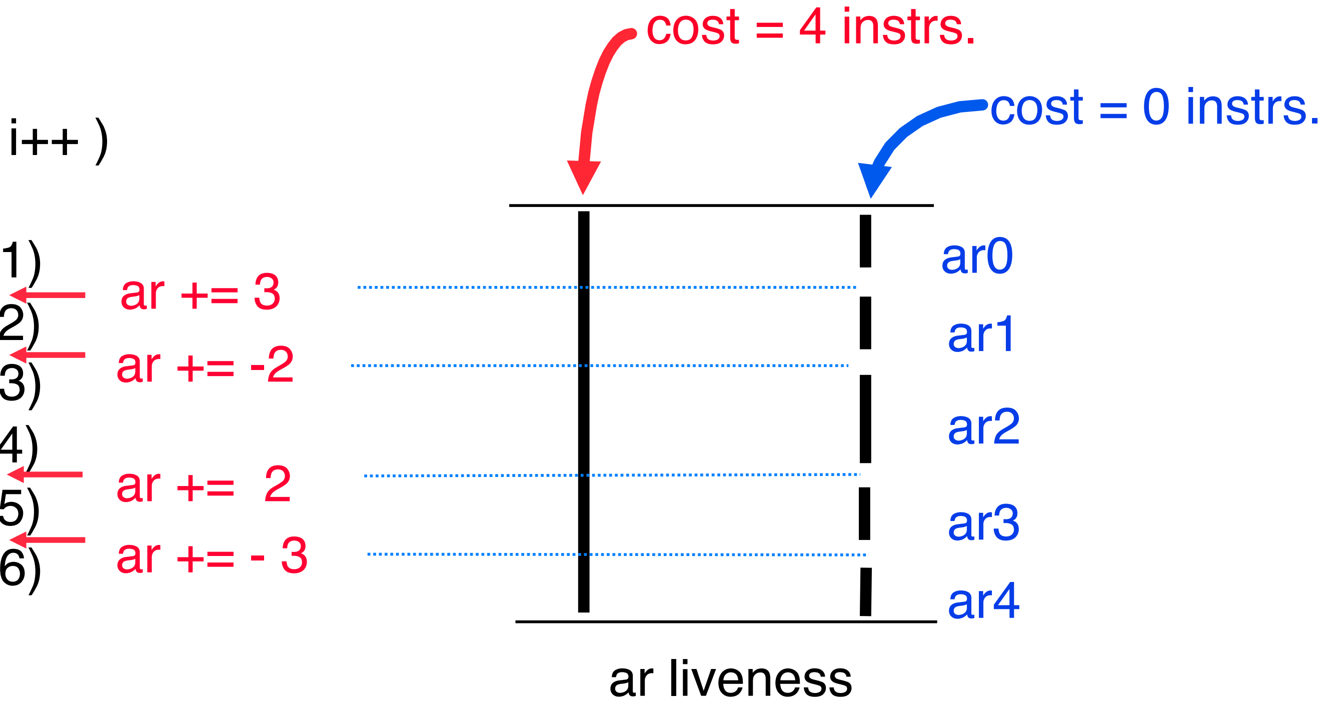
ld r1 , [ar2++]

update ar : ar2 += 3

Local Array Reference Allocation

- Problem:
  - ❑ Given a sequence of array references in a loop.
  - ❑ Determine an allocation for Address Registers (ar's) such as to minimize the number of ar's and update instructions required.

```
for ( i = 2;  i < N - 2;  i++ )
{
    a [ i  - 2]  = 0        (1)
    a [ i + 1]  = 0         (2)
    a [ i  - 1]  = 0        (3)
    a [ i ]        = 0      (4)
    a [ i + 2]  = 0         (5)
    a [ i  - 1]  = 0        (6)
}
```



The Indexing Distance

- Loop with induction variable i, linearly updated by step s.
- Array references r1 = v [a*i+b] and r2 = v [a*i+c].
  - Associate tuples to references: r1 = (a,i,b) and r2 = (a,i,c).
  - Assume that r1 is before r2 in the program order.
  - r1 < r2, if r1 and r2 are in the same iteration.
  - r1 > r2, if r1 is in the next iteration after r2 iteration.

- The indexing distance between r1 = (a, i, b) and r2 = (a, i, c):

$$d(r1, r2) = \begin{cases} |c - b| & \text{if } r1 < r2 \\ |c - b + a * s| & \text{if } r1 > r2 \end{cases}$$

The Indexing Distance (cont.)

- Motivation:
  - ❑ Maximize advantage of auto-increment/decrement feature.
  - ❑ Ability to use it is limited by the indexing distance.

```
for ( i = 2;  i < N - 2;  i++ )
{
    a [ i  - 2]         (1)
    a [ i + 1]          (2)
    a [ i  - 1]         (3)
    a [ i ]             (4)
    a [ i + 2]          (5)
    a [ i  - 1]         (6)

}
```

– distance $2 \longrightarrow 4$
  - (2) = i + 1 and (4) = i
  - d (2,4) = | i - (i + 1) | = 1

– distance $3 \longrightarrow 5$
  - (3) = i - 1 and (5) = i + 2
  - d (3,5) = | (i + 2) - (i - 1) | = 3

– distance $6 \longrightarrow 1$
  - (6) = i - 1 and (1) = i - 2
  - d (6,1) = | (i - 2) + 1 - (i - 1) | = 0

The Multidimensional Case

- Tuples for indices at dimension k: $r1 = (a_k, i, b_k)$ and
- $r2 = (a_k, i, c_k)$
- Dimensional shift:
- Indexing distance:

$$D_k = \begin{cases} 1 & \text{if } k = n \\ \prod_{j=k+1}^{n} size_j & \text{otherwise} \end{cases}$$

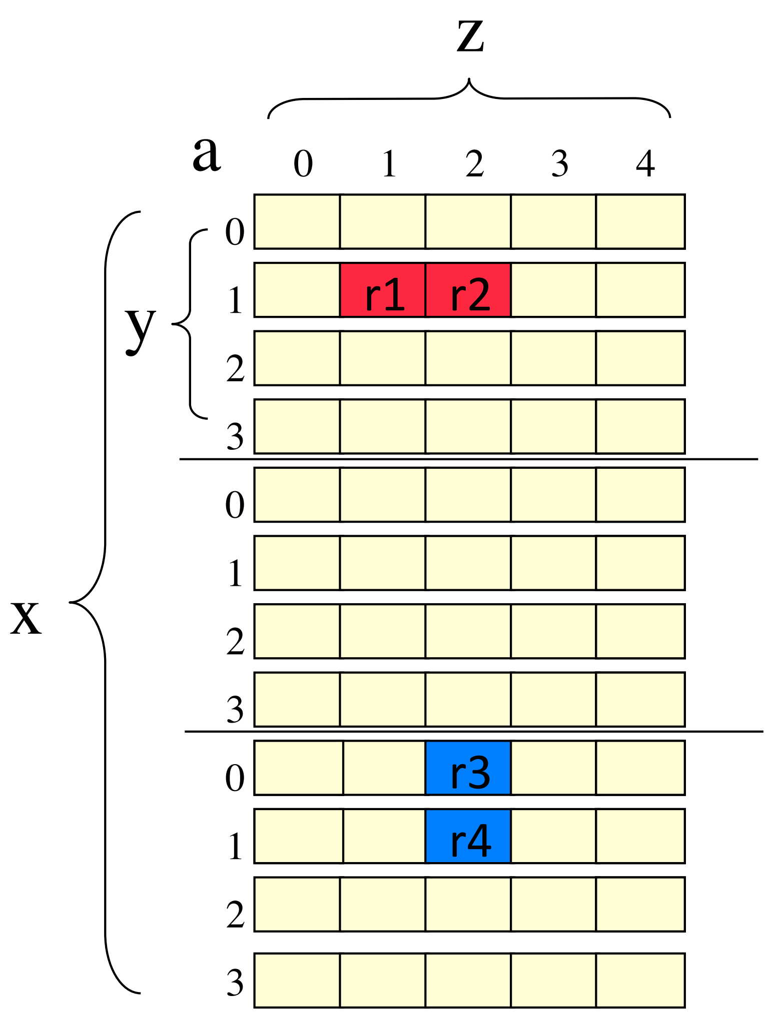
$$d(r1, r2) = \begin{cases} \sum_{k=1}^{n} |(c_k - b_k)| * D_k & \text{if } r1 < r2 \\ \sum_{k=1}^{n} |(c_k - b_k + a_k * s)| * D_k & \text{if } r1 > r2 \end{cases}$$

The Multidimensional Case (cont.)

- Let v[3][4][5] be a tridimensional vector.
- The dimensional shifts for v are:
  - $D_1 = 4 * 5 = 20$
  - $D_2 = 5$
  - $D_3 = 1$

- Consider r1 = v[0][1][1] and r2 = v[0][1][2]:
  - d(r1,r2) = $|2 - 1| * D_3 = 1$

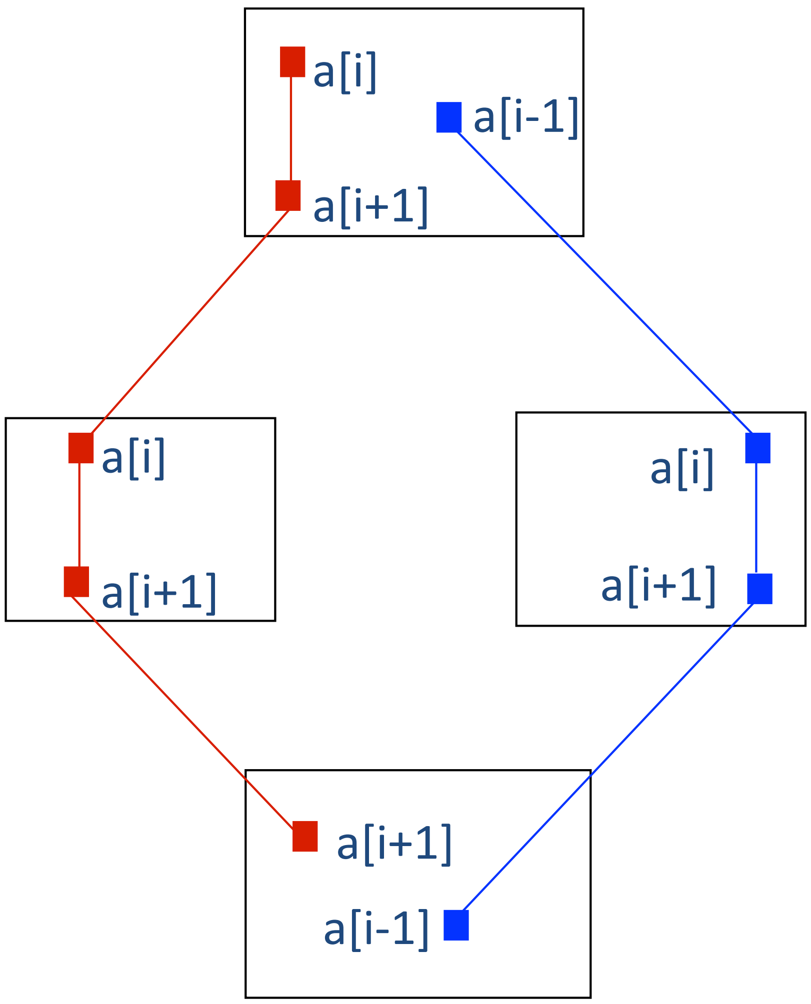
- Consider r3 = v[3][0][2] and r4 = v[3][1][2]:
  - d(r1,r2) = $| 1 - 0 | * D_2 = 5$

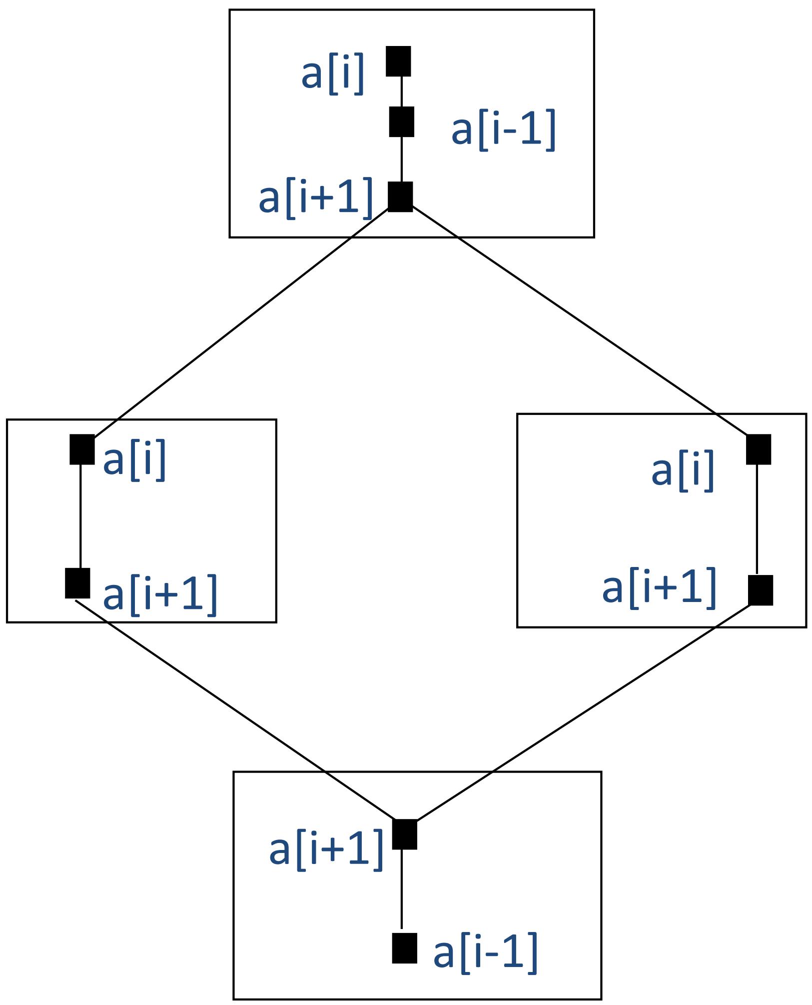


Live Range Growth

- Pointer arithmetic is usually cheaper than memory spilling.
- To decide between auto-increment/decrement or an update instruction, we have to know (at compile time) which single reference reaches any other reference.
  - Have to decide at each join block which single reference leaves the block.
  - Number of join blocks is related to number of update instructions.
  - Use SSA-form to represent references (*Single Reference Form*).
- Basic solution is to grow live ranges of references:
  - Each range is allocated to an address register (ar).
  - Join ranges pairwise until the number of ar's is smaller than the number of ar's in the processor.
  - At each step, join the pair with the smallest join cost.

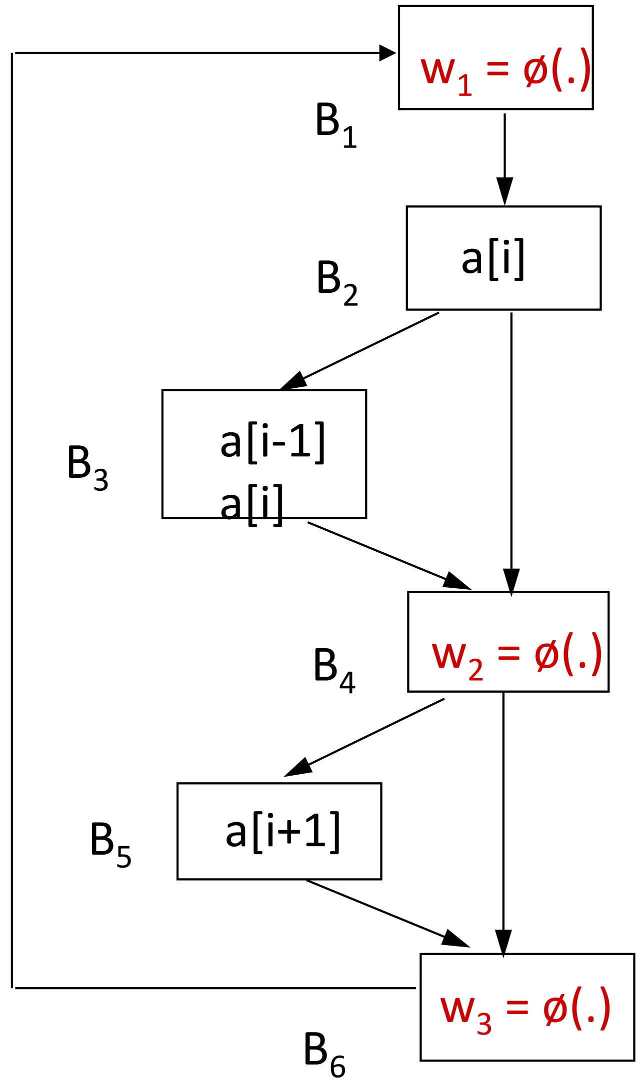
Live Range Growth (cont.)





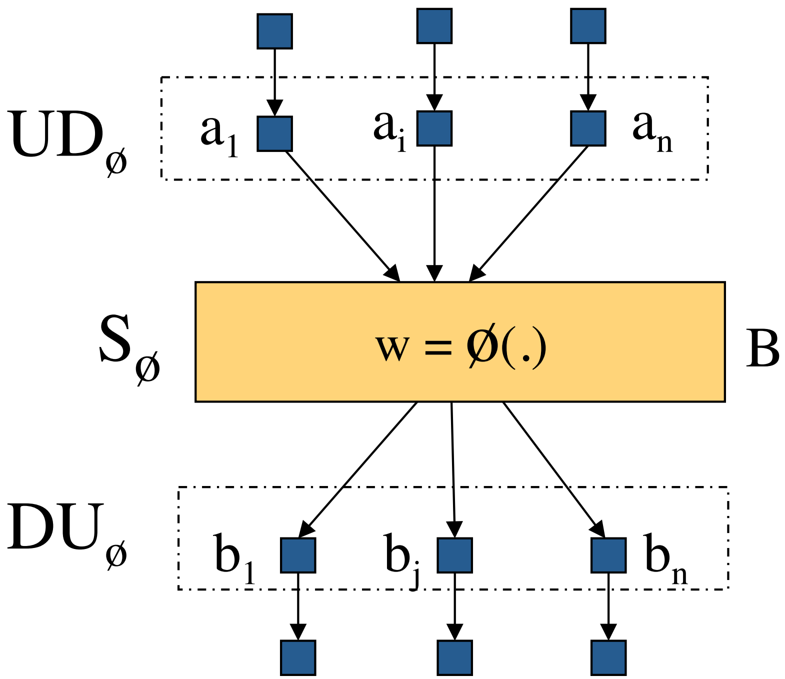
Single Reference Form (SRF)

- Presence of references in SRF is equivalent to a variable definition in SSA.
- Insert ø-functions as in SSA.
  - Cytron et al [1989]
- Perform reference analysis to compute the arguments of ø-functions.
  - Unlike in SSA, arguments in SRF are both sets: use-def and def-use.



Reference Analysis

- *Reference Analysis* is used to determine which references reach (or are reachable by) the result of ø-functions.
- The ø-function arguments become the elements in $UD_ø$ and $DU_ø$.



- ❑ Set $UD_ø$ is the set of references that reach statement $S_ø$.
- ❑ Set $DU_ø$ is the set of references that are reachable by w.

Reference Analysis (cont.)

$UD_1 = \{ w_3 \}$

$DU_1 = \{ a[i] \}$

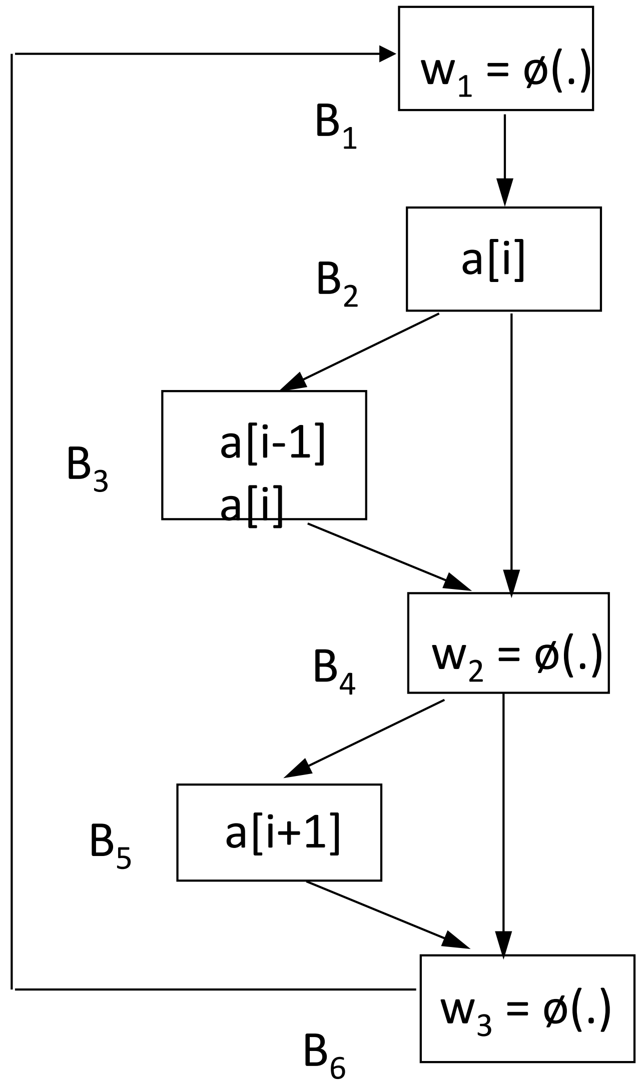
$w_1 = Ø(w_3, a[i])$

$UD_2 = \{ w_1 \}$

$DU_2 = \{a[i-1], \; w_2\}$

$UD_3 = \{ a[i] \}$

$DU_3 = \{ \; w_2 \}$



$UD_4 = \{a[i], a[i] \}$

$DU_4 = \{a[i+1], w_3 \}$

$w_2 = Ø(a[i], a[i], a[i+1], w_3)$

$UD_5 = \{ w_2 \}$

$DU_5 = \{ \; w_3 \}$

$UD_6 = \{a[i+1], w_2 \}$

$DU_6 = \{ w_1 \}$

$w_3 = Ø(a[i+1], \; w_2, w_1)$

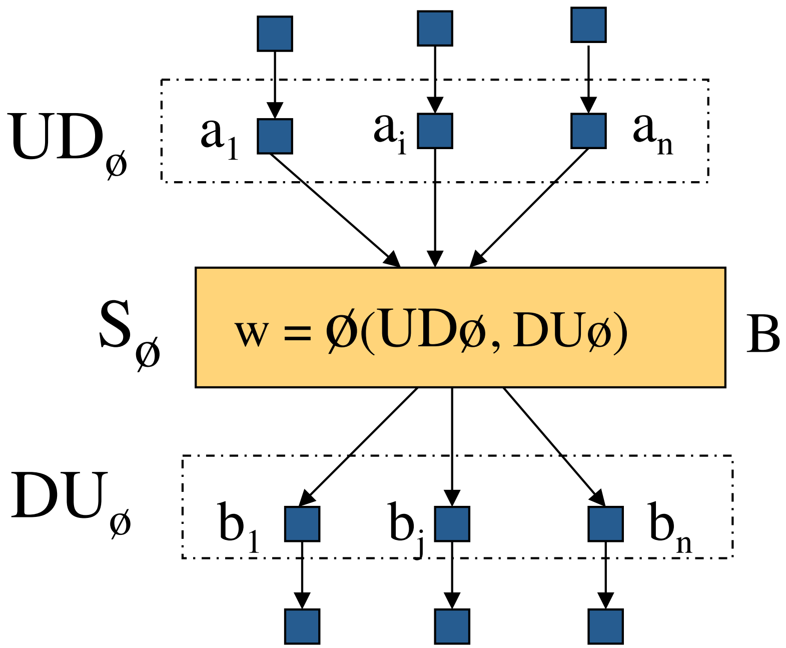
Reference Equations

- Ø-functions form a system of assignment equations.
  - $w_1 = Ø(w_3, a[i])$
  - $w_2 = Ø(a[i], a[i], a[i+1], w_3)$
  - $w_3 = Ø(a[i+1], w_1, w_2)$
- The system usually has circular dependencies.
  - Estimates for the values of w_1, w_2 and w_3 must be computed.
  - Determine the best evaluation order for the equations which minimizes the number of cycles to break in the dependency graph.
  - Have to design a compiler ! Pick the one at the tail of the loop first and follow backward to the head of the loop.

Computing ø-functions

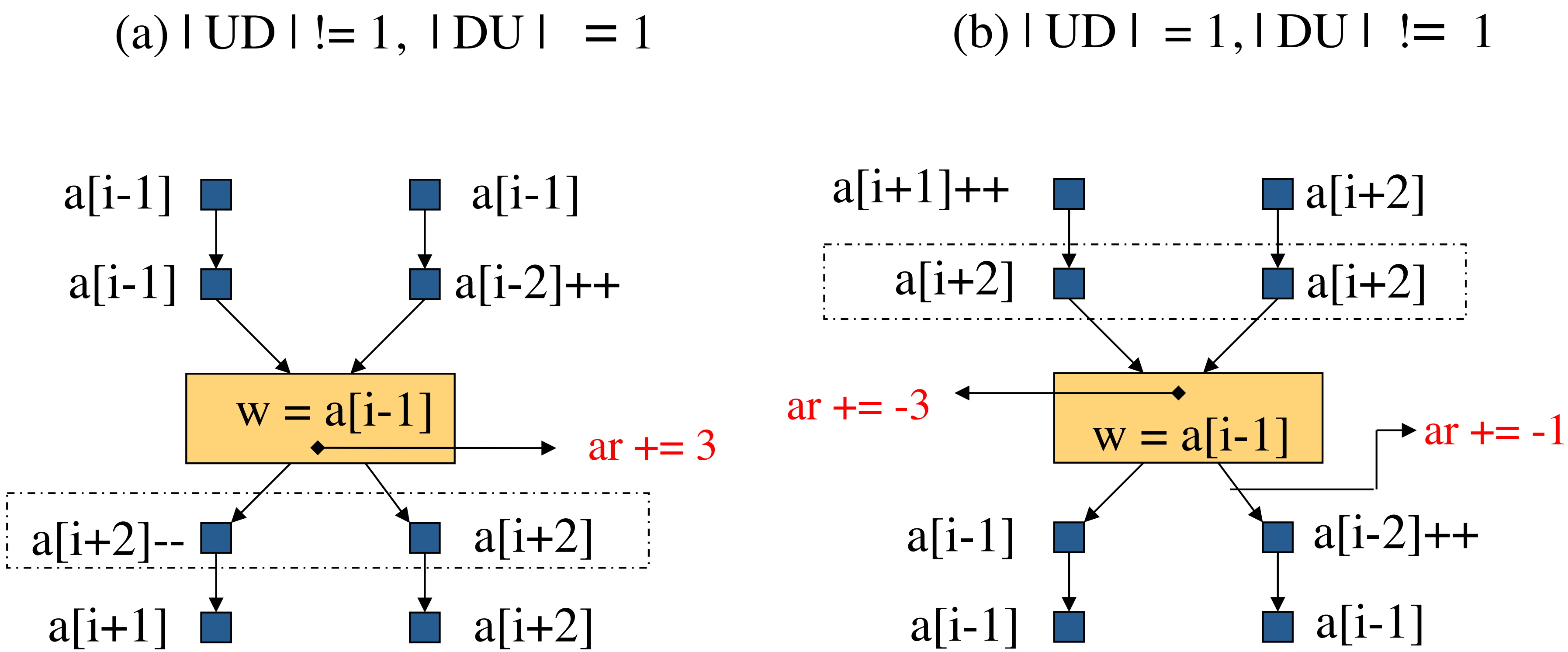
- Determine the result w of the ø-functions.

- Minimize cost(a,b) = $\begin{cases} 0, & \text{if } |d(a,b)| \leq 1 \text{ and a is a real reference} \\ & \text{if } |d(a,b)| = 0 \text{ and a is the result of a } \varnothing\text{- function} \\ 1, & \text{otherwise} \end{cases}$

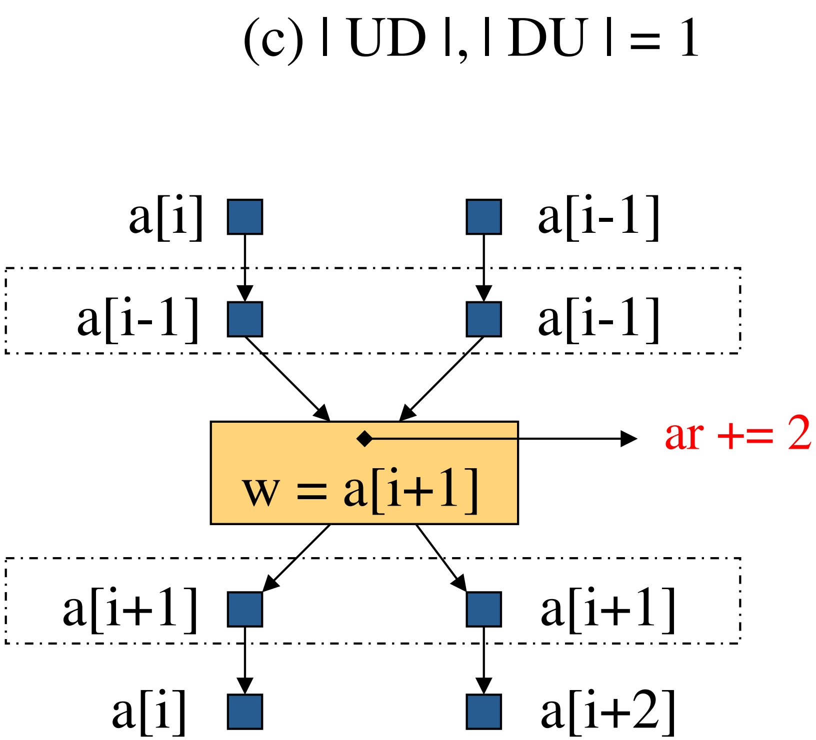


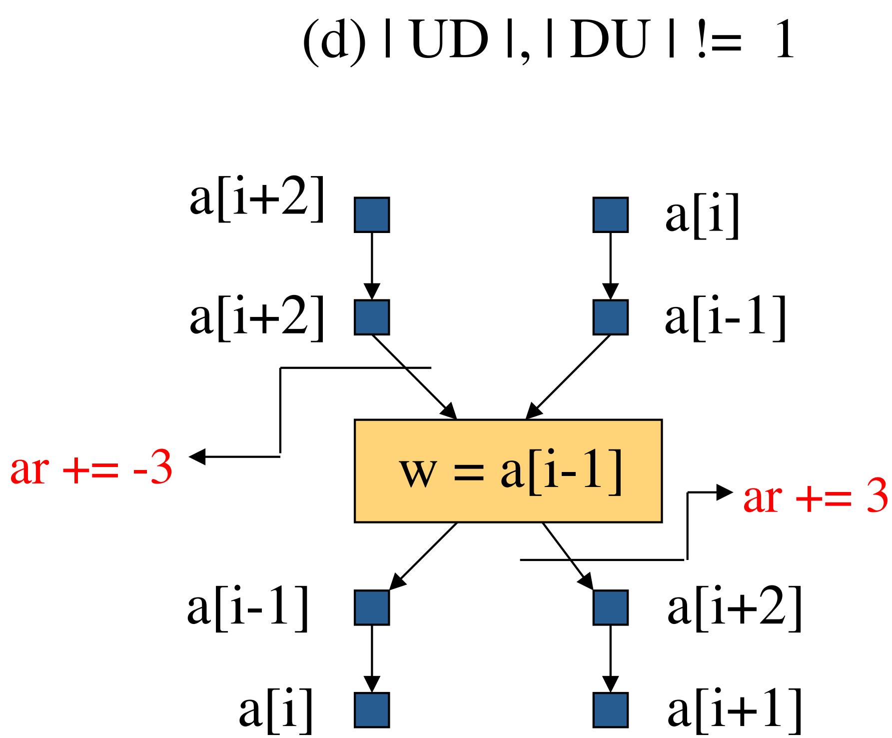
$$\min_{w} \left(\sum_{i=1}^{|UD_{\varnothing}|} cost(a_i, w) + \sum_{j=1}^{|DU_{\varnothing}|} cost(w, b_j) \right)$$

Computing ø-functions (cont.)

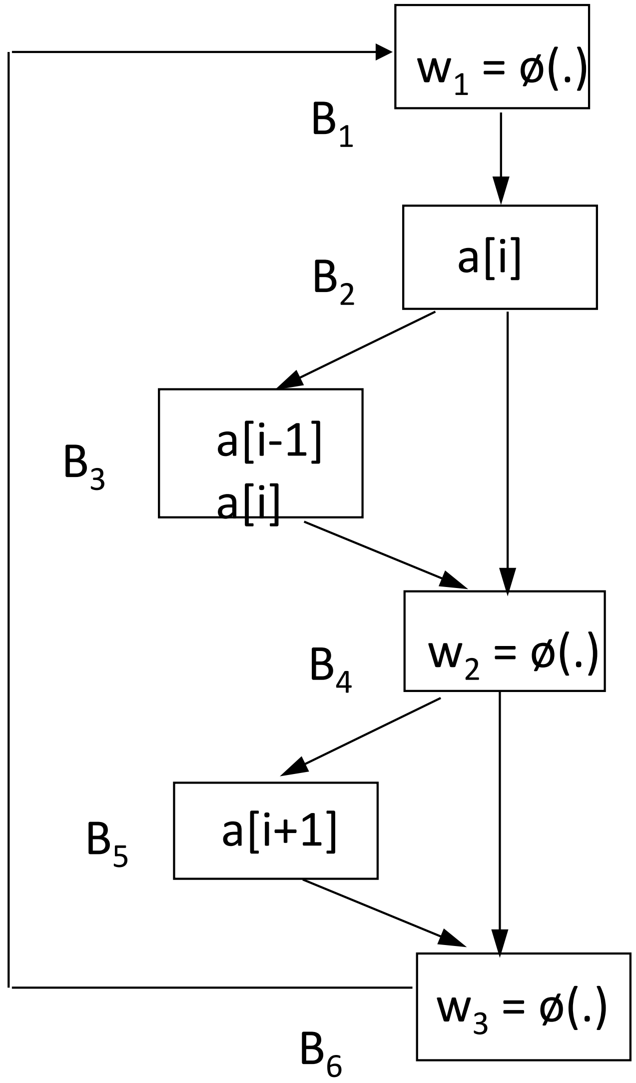


Computing ø-functions (cont.)





Solving Reference Equation System



$w_1 = \emptyset(w_{3,}\ a[i])$

$w_2 = \emptyset(a[i],\ a[i],\ a[i+1],\ w_3)$

$w_3 = \emptyset(a[i+1],\ \ w_2,\ w_1\ )$

<u>Solution:</u>

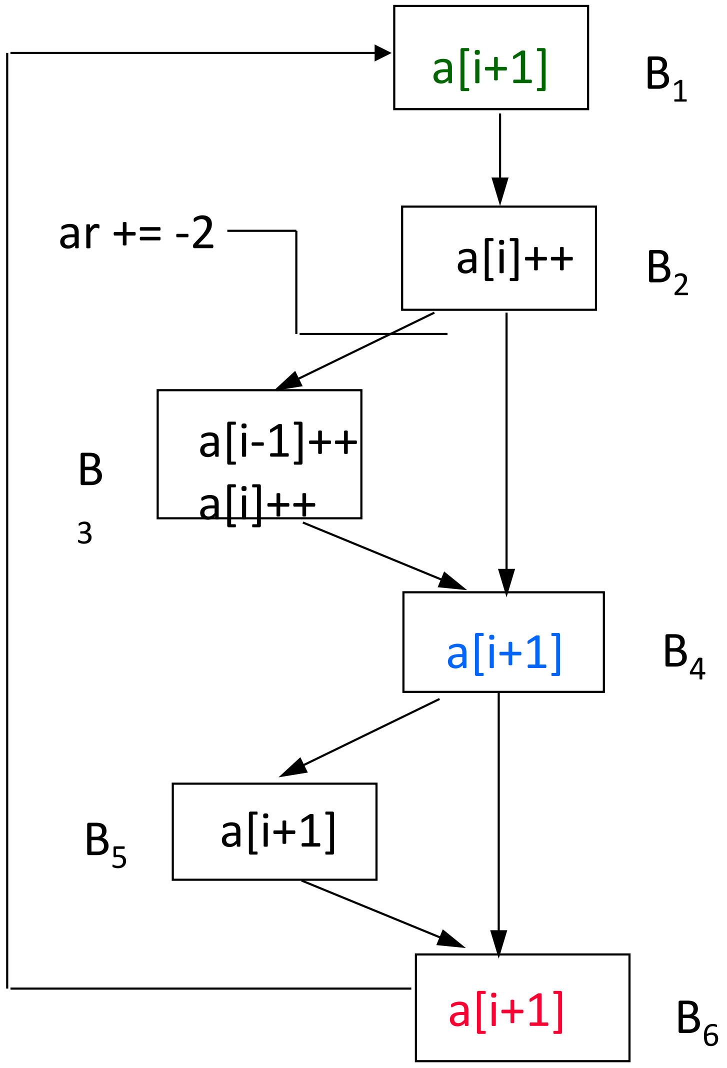
(1) $w_3 = \emptyset(a[i+1]\ ) = a[i+1]$

(2) $w_2 = \emptyset(a[i],\ a[i],\ a[i+1],\ a[i+1]) = a[i+1]$

(3) $w_1 = \emptyset(a[i+1],\ a[i]) = a[i+1]$

SAME FOR CONSECUTIVE ITERATIONS

Update Instruction/Mode Insertion




```
p = &a[1];
for (i = 1; i < N-1; i++) {
  avg = *p++ >> 2;
  if (i % 2) {
    p += -2;
    avg += *p++ << 2;
    *p++ = avg * 3;
  }
  if (avg < error)
    avg -= *p - error/2;
}
```