Array Reference Allocation

Sandro Rigo

Institute of Computing - UNICAMP

Overview

- Introduction
- Motivation
- Indexing Distance
- Live Range Growth
- Single Reference Form (SRF)
- Results
- Implementation on IMPACT

Introduction

- Array Reference Allocation Using SSA-Form and Live Range Growth
  - (Cintra'00) Presented at ACM SIGPLAN LCTES 2000, Vancouver.
  - It extends previous work in the area by enabling efficient allocation in the presence of control- flow instructions.
  - Tested in an optimizing compiler from Conexant Systems Inc

Motivation

- **Embedded systems executing specialized programs encompass a considerable share of the processors produced every year.**

  - **These systems have hard performance, power consumption and code size constraints.**

  - **Most embedded processors offer specialized addressing modes.**

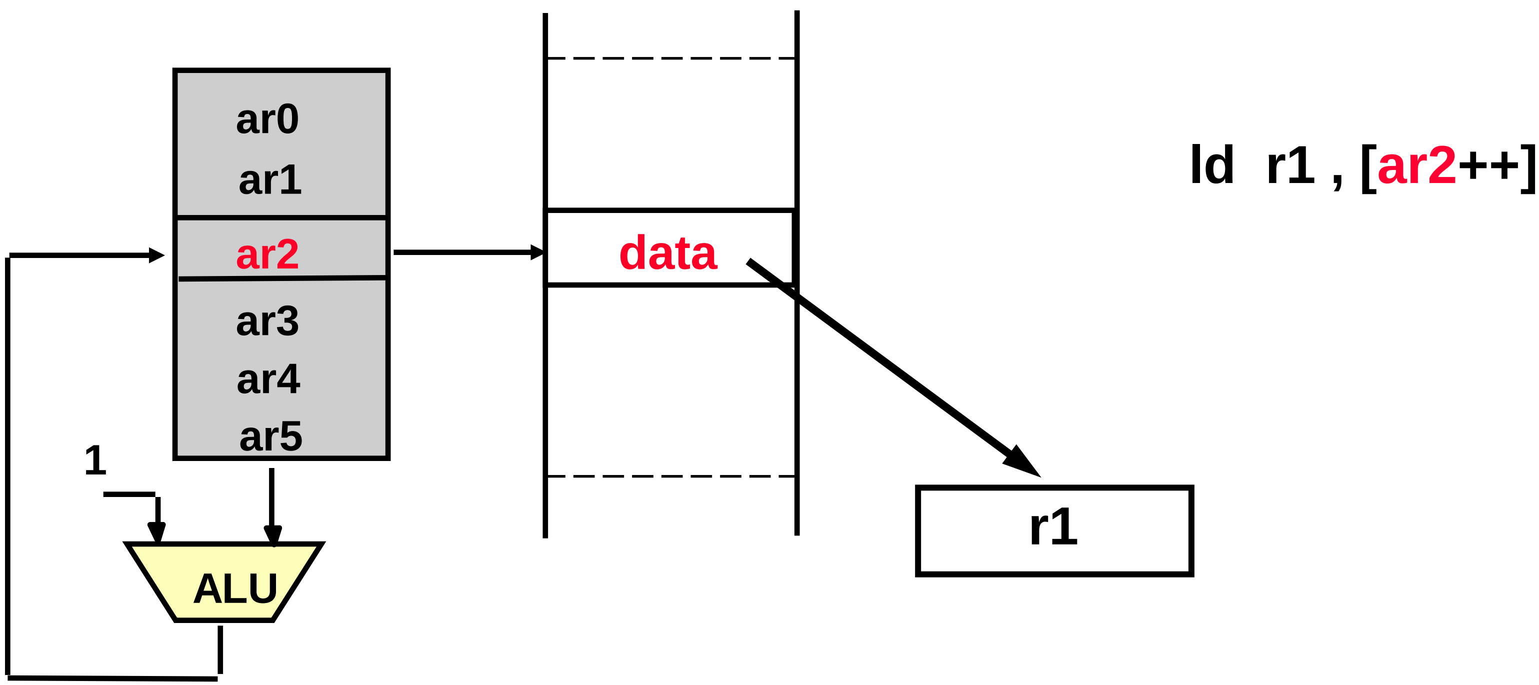
Indirect Addressing

- **Address computation is expensive.**
  - **One out of 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.
  - Available in the ISA of most embedded processors.



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;
}
```

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 triples 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 → 4**
  - (2) = i + 1 and (4) = i
  - d (2,4) = | i - (i + 1) | = 1
- **distance 3 → 5**
  - (3) = i - 1 and (5) = i + 2
  - d (3,5) = | (i + 2) - (i - 1) | = 3
- **distance 6 → 1**
  - (6) = i - 1 and (1) = i - 2
  - d (6,1) = | (i - 2) + 1 - (i - 1) | = 0

The Multidimensional Case

- **Triples for indices at dimension k: r1 = (a_k, i, b_k) and r2 = (a_k, i, c_k)**

- **Dimensional shift:**

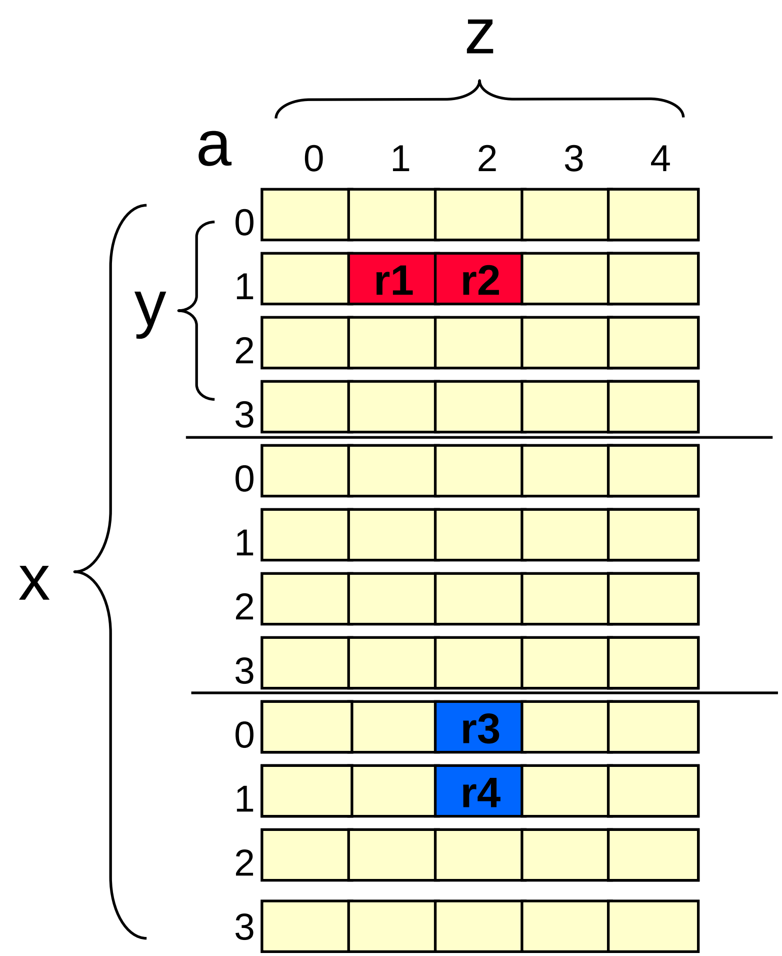
$$D_k = \begin{cases} 1 & \text{if } k = n \\ \prod_{j = k+1}^{n} size_j & \text{otherwise} \end{cases}$$

- **Indexing distance:**

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

The Multidimensional Case (cont.)

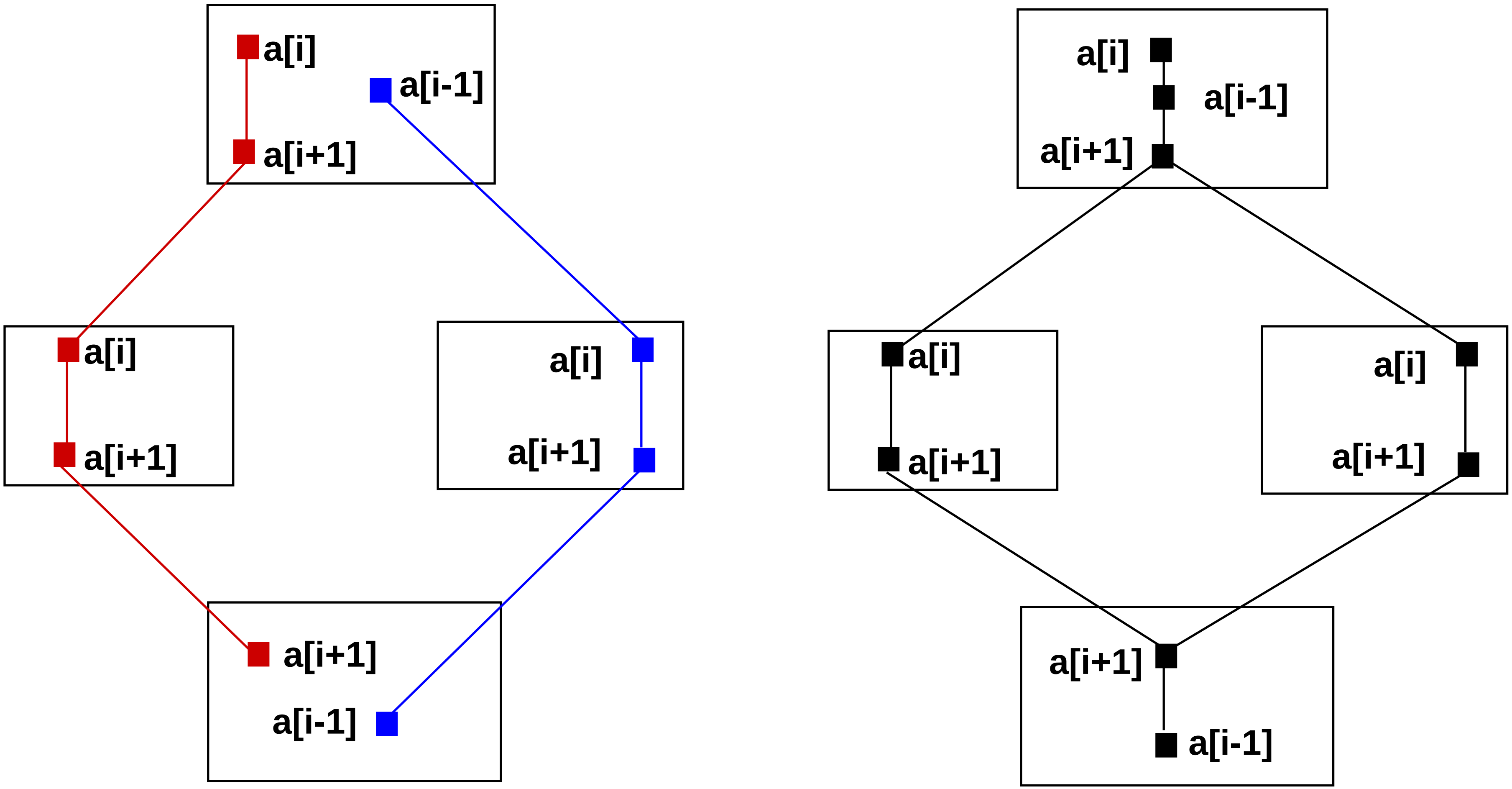
- **Let v[3][4][5] be a tridimensional vector.**
- **The dimensional shifts for v are:**
  - **D_1 = 4 * 5 = 20**
  - **D_2 = 5**
  - **D_3 = 1**
- **Consider r1 = v[0][1][1] and r2 = v[0][1][2]:**
  - **d(r1,r2) = |2 - 1| * D_3 = 1**
- **Consider r3 = v[3][0][2] and r4 = v[3][1][2]:**
  - **d(r1,r2) = | 1 - 0 | * D_2 = 5**



Live Range Growth

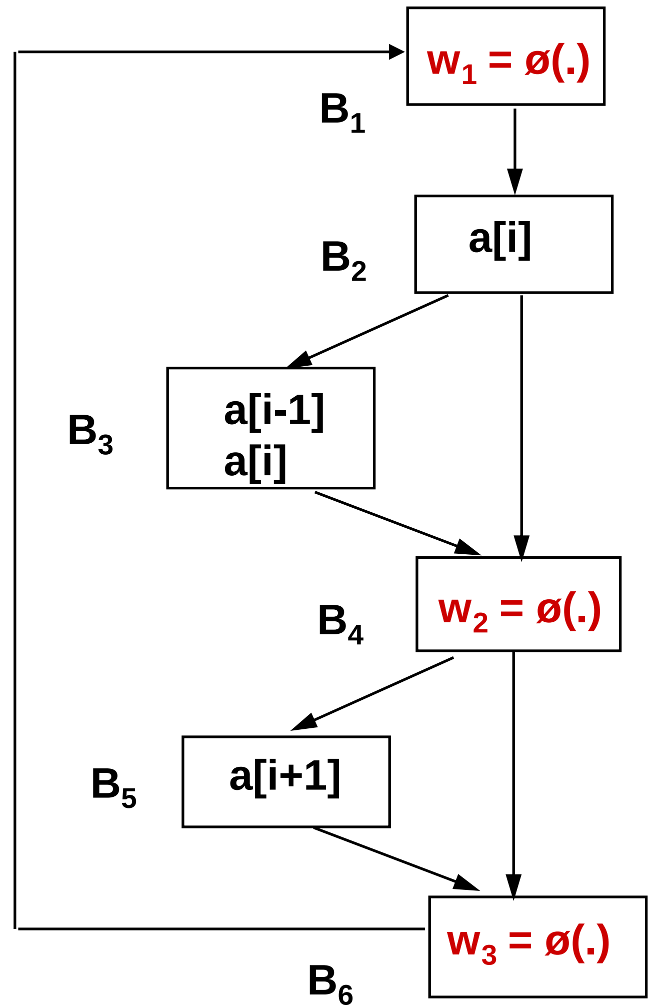
- **Pointer arithmetic is usually cheaper than memory spilling.**
- **To decide between auto-increment/decrement or an update instruction, we have to know (<u>at compile time</u>) which single reference reaches any other reference.**
  - **Have to decide at each join block which single reference leaves 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.)



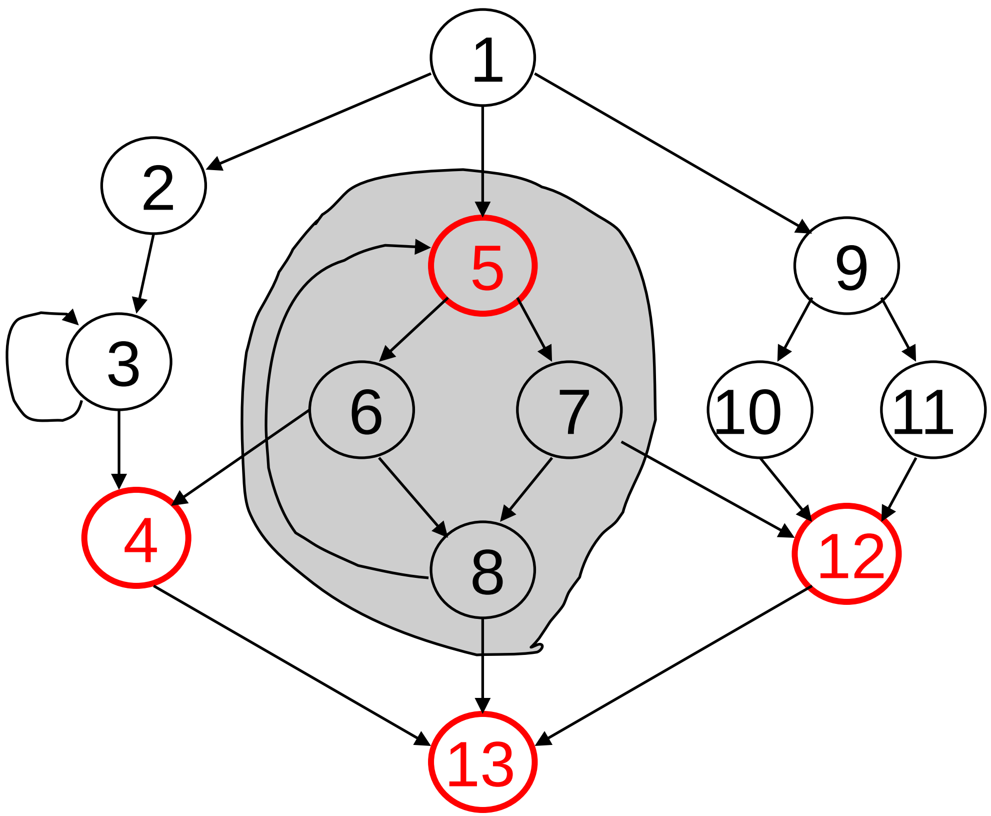
Single Reference Form (SRF)

- **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.**



Single Reference Form SRF (cont.)

- **The Dominance Frontier (DF) of a set of nodes, which have array references, shows us the nodes where we need to insert Ø-functions**

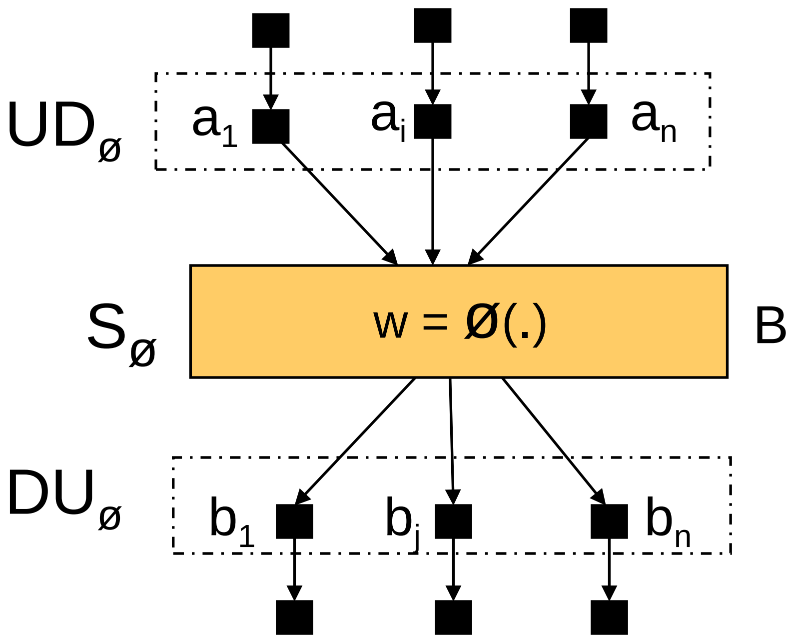


- The DF of 5 is: (4,5,12,13)

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.)

$UD_1 = \{ w_3 \}$

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

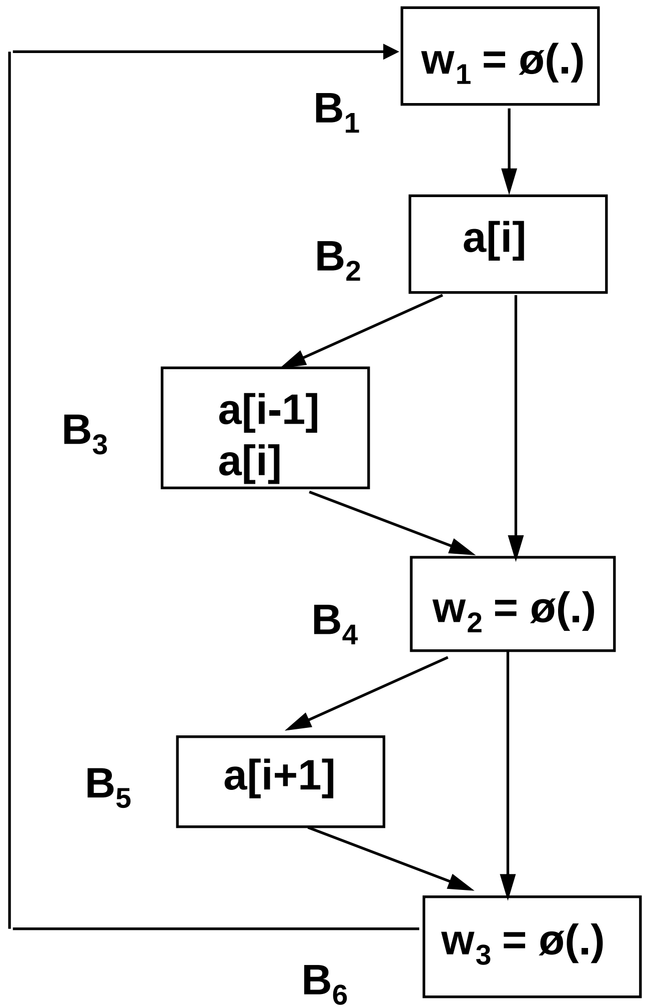
$w_1 = Ø(w_3, a[i])$

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

$DU_3 = \{ w_2 \}$

$UD_5 = \{ w_2 \}$

$DU_5 = \{ w_3 \}$



$UD_2 = \{ w_1 \}$

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

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

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

$w_2 = Ø(a[i], a[i], a[i+1], w_3)$

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

$DU_6 = \{ w_1 \}$

$w_3 = Ø(a[i+1], w_2, w_1)$

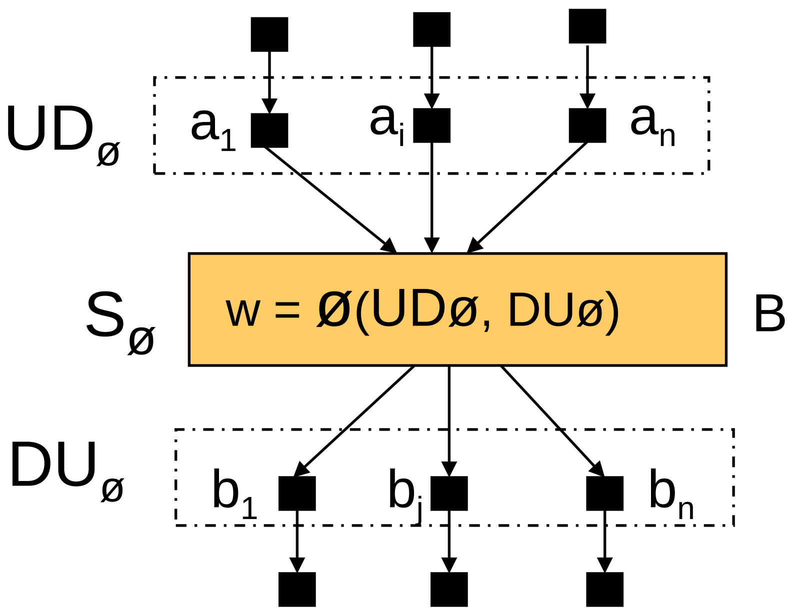
Reference Equations

- 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) =**

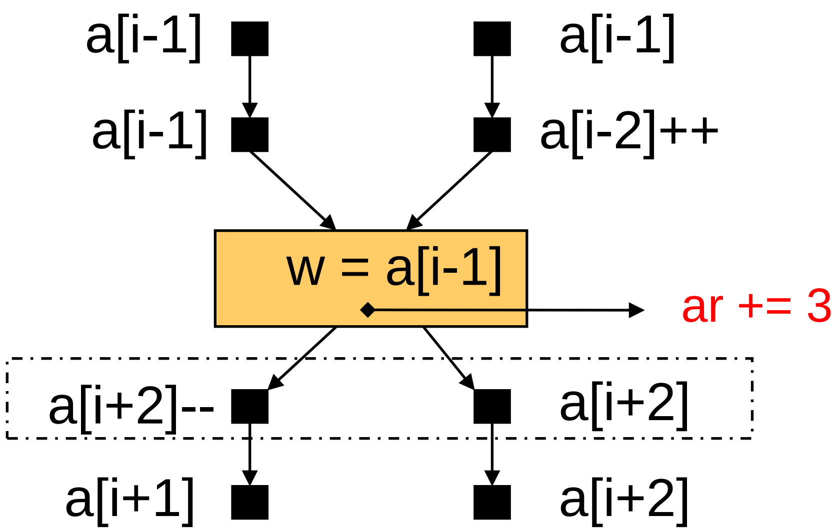
$$\mathrm{cost}(a,b) = \begin{cases} 0, & \text{if } |d(a,b)| \le 1 \text{ and a is a real reference} \\ & \text{if } |d(a,b)| = 0 \text{ and a is the result of a Ø- function} \\ 1, & \text{otherwise} \end{cases}$$



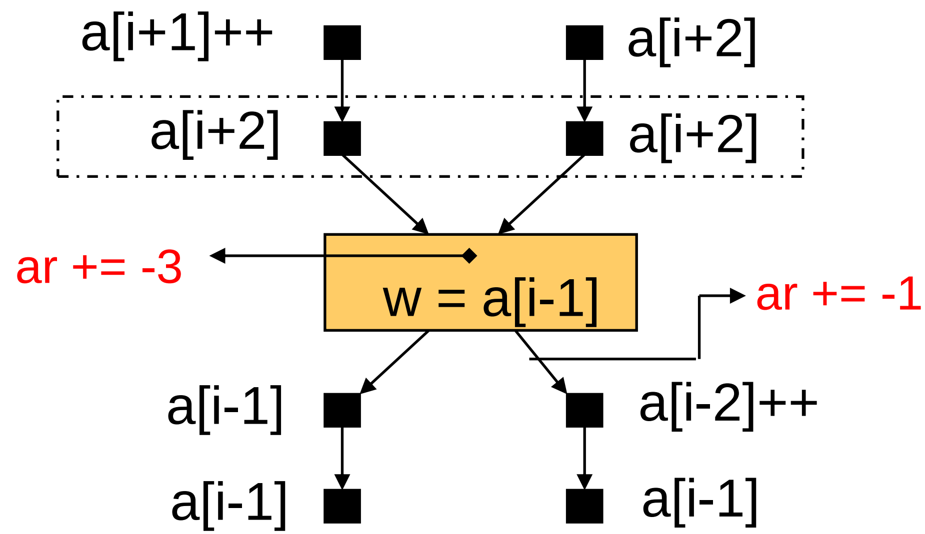
$$\min_{w} \left(\sum_{i=1}^{|UD_{ø}|} \mathrm{cost}(a_i, w) + \sum_{j=1}^{|DU_{ø}|} \mathrm{cost}(w, b_j) \right)$$

Computing ø-functions (cont.)

(a) | UD | != 1, | DU | = 1

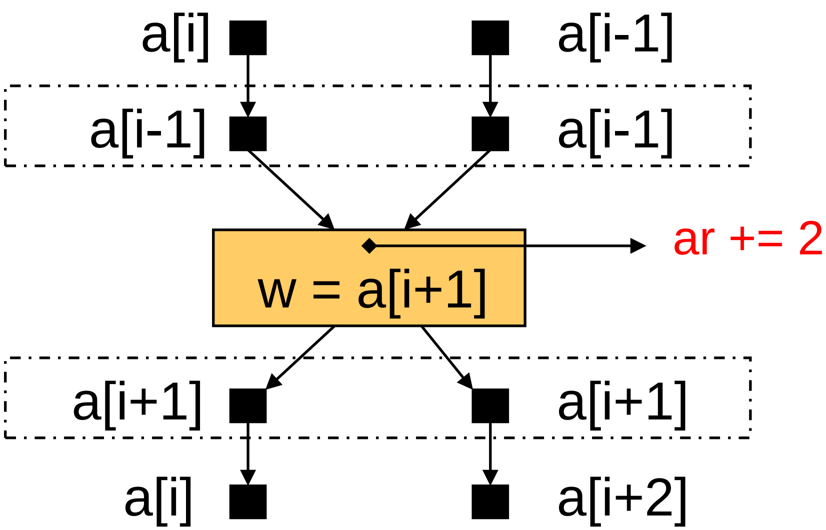


(b) | UD | = 1, | DU | != 1

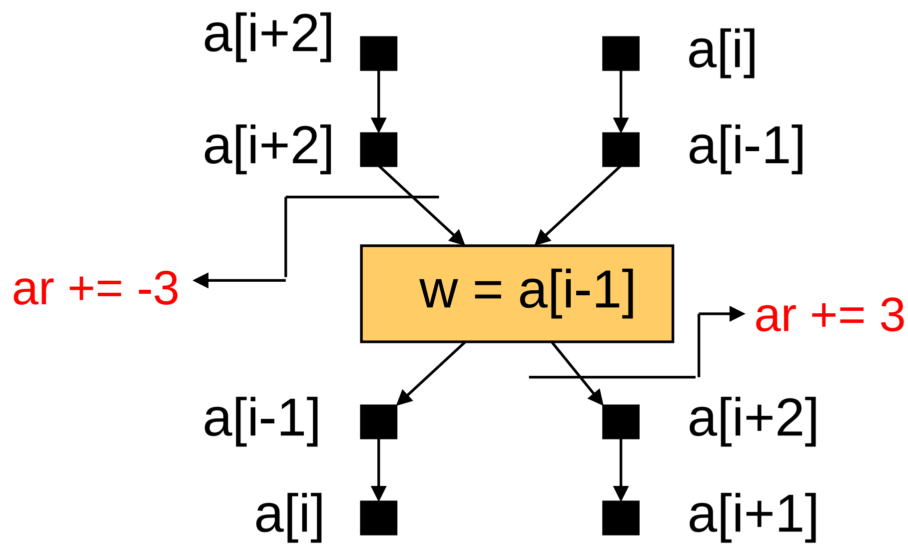


Computing ø-functions (cont.)

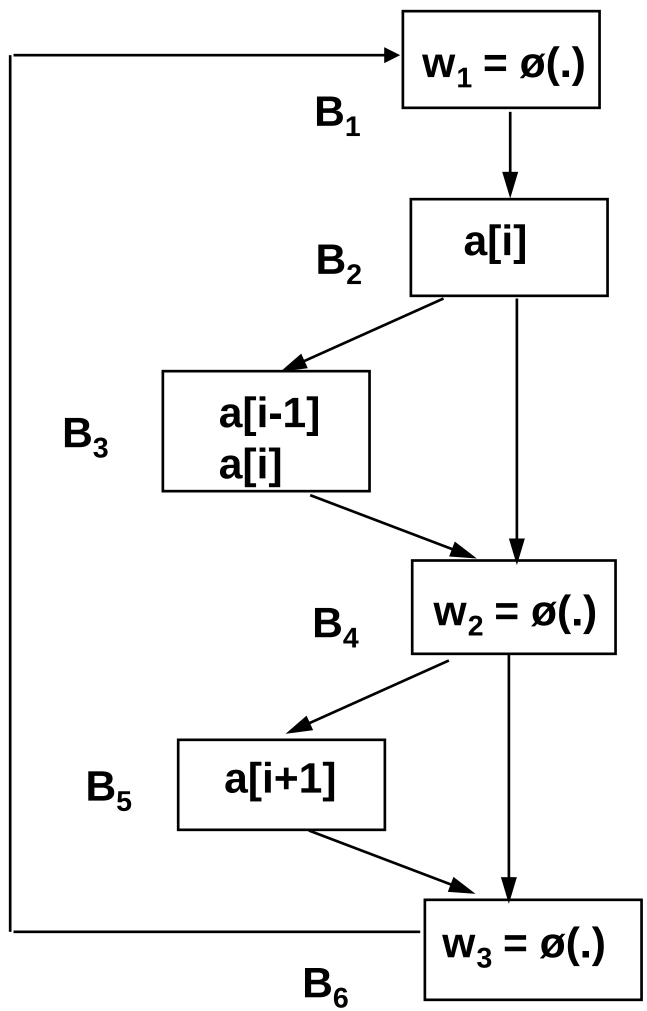
(c) | UD |, | DU | = 1



(d) | UD |, | DU | != 1



Solving Reference Equation System



$w_1 = Ø(w_3, a[i])$

$w_2 = Ø(a[i], a[i], a[i+1], w_3)$

$w_3 = Ø(a[i+1], w_2, w_1)$

Solution:

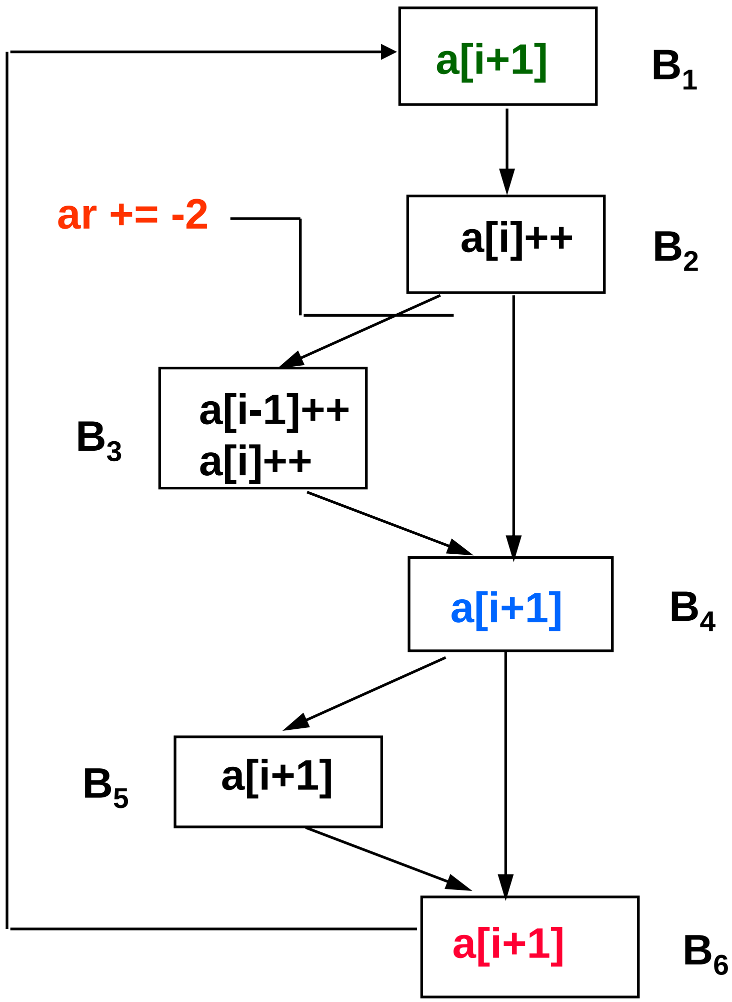
(1) $w_3 = Ø(a[i+1]) = a[i+1]$

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

(3) $w_1 = Ø(a[i+1], a[i]) = a[i+1]$

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;
}
```

Experimental Results

Program	Priority-based Cycles	Priority-based Size	LRG Optimized Cycles	LRG Optimized Size	Comparison Speedup	Comparison Size
conv enc	4331	4667	3943	4647	9%	0%
conv olution	1220	2068	1042	2077	17%	1%
dot_product	165	1305	160	1269	3%	-2%
biquad_N_sections	1380	2980	1218	2905	13%	-2%
fir_array	1471	2626	1263	2666	16%	2%
fir2dim	7684	4546	6728	4566	14%	1%
lms_array	2276	3644	1919	3665	18%	1%
mat1_x3	1202	2668	1113	2705	7%	2%
matrix1	34657	3057	30520	3135	13%	3%
n_complex_updates	2985	3300	2336	3410	27%	4%
n_real_updates	1855	2716	1452	2785	27%	3%
fft	173931	10103	165549	10097	5%	0%
autcor	179633	4003	167238	3990	7%	0%
fir8	280324	5143	256476	5088	9%	-1%
latsynth	3115	3408	3050	3402	2%	0%
fir_lms2	3454	3353	3317	3298	4%	-1%
latanal	703662	3425	691662	3411	2%	0%
AVERAGE					11.4%	0.6%

Results Comparison

Program	*IG based speedup*	*Greedy speedup*
convenc	9%	9%
convolution	17%	17%
dot_product	3%	2%
biquad_N_sections	13%	13%
fir_array	16%	16%
fir2dim	14%	14%
lms_array	18%	18%
mat1_x3	7%	6%
matrix1	13%	13%
n_complex_updates	27%	27%
n_real_updates	27%	27%
fft	5%	3%
autcor	7%	7%
fir8	9%	9%
latsynth	2%	1%
fir_lms2	4%	4%
latanal	2%	1%
TOTAL	11,35	11,00

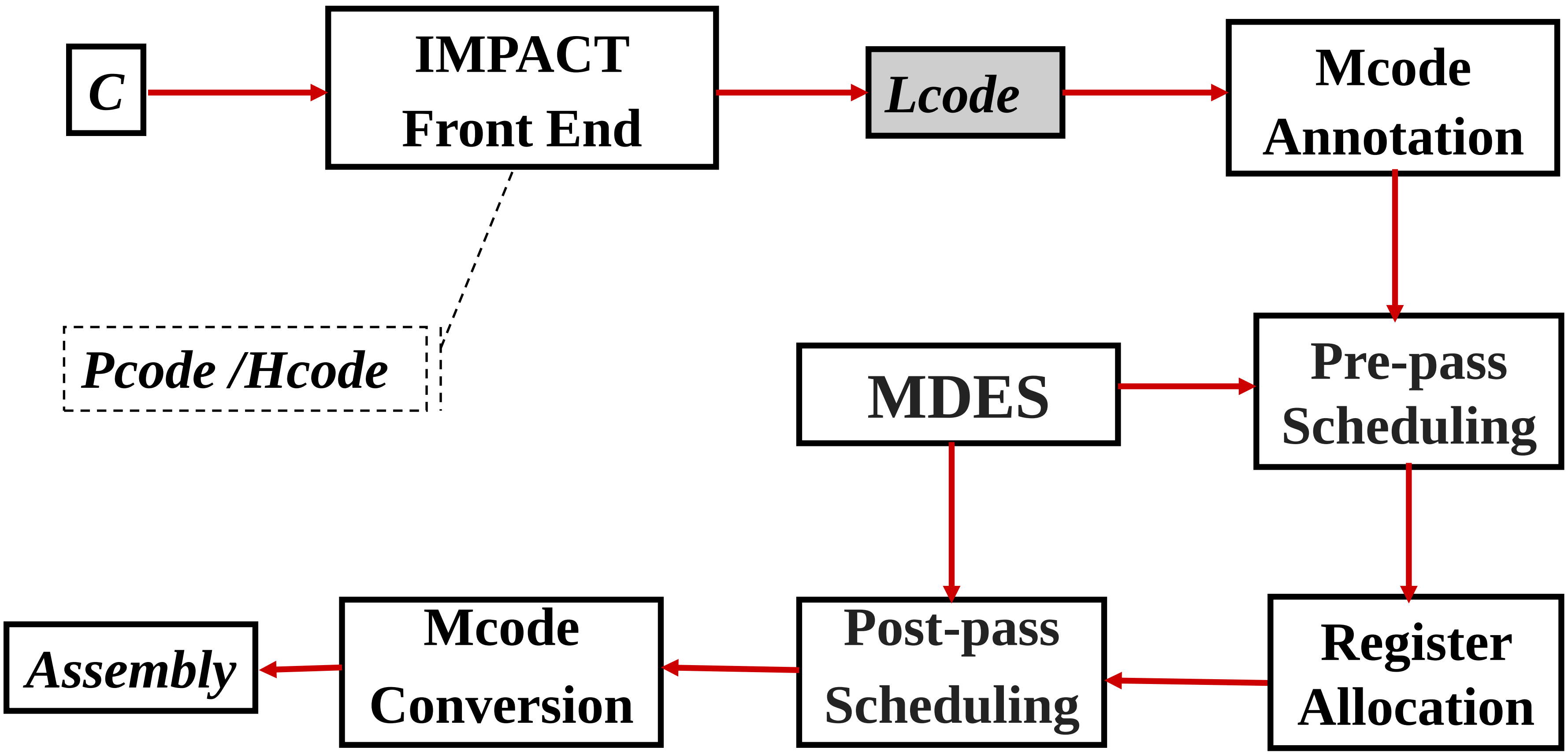
- **Compiler from Conexant Systems Inc.**
  - **Optimizing DSP compiler**
  - **Performs all Dragon Book optimizations**
    - **Induction Variable Elimination**
    - **Dead Code Removal**
    - **Graph coloring based register allocation**
    - **etc.**
- **Benchmarks**
  - **DspStone Benchmarks**
  - **Conexant Benchmarks**

Conclusions and Future Work

- An algorithm to perform Global Array Reference Allocation.
  - Uses SRF to minimize number of update instructions.
  - Average speed-up is 11%, insignificant size overhead.

- Moving to other processors.
  - IMPACT/Trimaran for a VLIW DSP architecture (MESCAL) - In Progress.
  - GNU gcc for the Motorola 68K.

IMPACT Compiler Framework



IMPACT - ARA Implementation

- **Lcode:**
  - □ It is the main intermediate representation of IMPACT;
  - □ We can implement ARA before classical (Dragon Book) optimization.
  - □ Loop information available: Induction Variable, Nesting Level, Back Edges, Ind. Var. Operations and profile.
- **Missing Information on Lcode:**
  - □ Type information: Used to identify array access, multidimensional arrays.

IMPACT - Lcode

```
 (cb 3 8.000000 [(flow 1 5 4.000000)(flow 0 4
4.000000)] <(iteration_header (i 1)(i 1))(iter_8 (f2
1)(f2 1))>)
    (op 7 add_u [(r 3 i)] [(mac $LV i)(i -40)])
    (op 8 mul [(r 4 i)] [(r 1 i)(i 4)])
    (op 9 ld_i [(r 5 i)] [(r 3 i)(r 4 i)])
    (op 10 asr [(r 6 i)] [(r 5 i)(i 2)])
    (op 11 mov [(r 2 i)] [(r 6 i)])
    (op 12 rem [(r 7 i)] [(r 1 i)(i 2)])
    (op 13 beq [] [(r 7 i)(i 0)(cb 5)])
  (cb 4 4.000000 [(flow 1 5 4.000000)])
    (op 14 add_u [(r 8 i)] [(mac $LV i)(i -40)])
    (op 15 sub [(r 9 i)] [(r 1 i)(i 1)])
    (op 16 mul [(r 10 i)] [(r 9 i)(i 4)])
    (op 17 ld_i [(r 11 i)] [(r 8 i)(r 10 i)])
    (op 18 lsl [(r 12 i)] [(r 11 i)(i 2)])
    (op 19 add [(r 2 i)] [(r 2 i)(r 12 i)])
    (op 20 mul [(r 13 i)] [(r 2 i)(i 3)])
    (op 21 add_u [(r 14 i)] [(mac $LV i)(i -40)])
    (op 22 mul [(r 15 i)] [(r 1 i)(i 4)])
    (op 23 st_i [] [(r 14 i)(r 15 i)(r 13 i)])
```

IMPACT - Hcode

```
 (BB 3 (PROFILE 4.000000 (4 1 4.000000))
        (Aadd (var P_avg_6_18___1) (lshft (index (cast ((INT)(P))
(var P_a_6_7___1)) (sub (var P_i_6_15___1) (signed 1))) (signed 2)))
        (assign (index (cast ((INT)(P)) (var P_a_6_7___1)) (var
P_i_6_15___1)) (mul (var P_avg_6_18___1) (signed 3)))
        (GOTO 4) )
 (BB 4 (PROFILE 8.000000 (5 1 6.000000) (6 0 2.000000))
        (IF (lt (var P_avg_6_18___1) (signed 2)) (THEN 5) (ELSE 6)
(EXPR_PRAGMA "IFELSE\$i_2")))
```

IMPACT - ARA Implementation

- **Changes on Hcode**
  - **Hcode adds an new attribute to load/store operations;**
  - **This attribute is used on Lcode to identify array access.**

- **Changes on Lcode:**
  - **Lcode: We added dominance frontier computation.**
  - **Lopti: ARA optimization files. ARA is called from l_optimize before Dragon Book opti.**

IMPACT - Modified Lcode

```
    (cb   3   8.000000   [(flow   1   5   4.000000)(flow   0   4   4.000000)]
<(iteration_header (i 1)(i 1))(iter_8 (f2 1)(f2 1))>)
    (op 7 add_u [(r 3 i)] [(mac $LV i)(i -40)])
    (op 8 mul [(r 4 i)] [(r 1 i)(i 4)])
    (op 9 ld_i [(r 5 i)] [(r 3 i)(r 4 i)] <(ARRAY_ACCESS_ATTR)>)
    (op 10 asr [(r 6 i)] [(r 5 i)(i 2)])
    (op 11 mov [(r 2 i)] [(r 6 i)])
    (op 12 rem [(r 7 i)] [(r 1 i)(i 2)])
    (op 13 beq [] [(r 7 i)(i 0)(cb 5)])
  (cb 4 4.000000 [(flow 1 5 4.000000)])
    (op 14 add_u [(r 8 i)] [(mac $LV i)(i -40)])
    (op 15 sub [(r 9 i)] [(r 1 i)(i 1)])
    (op 16 mul [(r 10 i)] [(r 9 i)(i 4)])
    (op 17 ld_i [(r 11 i)] [(r 8 i)(r 10 i)] <(ARRAY_ACCESS_ATTR)>)
    (op 18 lsl [(r 12 i)] [(r 11 i)(i 2)])
    (op 19 add [(r 2 i)] [(r 2 i)(r 12 i)])
    (op 20 mul [(r 13 i)] [(r 2 i)(i 3)])
    (op 21 add_u [(r 14 i)] [(mac $LV i)(i -40)])
    (op 22 mul [(r 15 i)] [(r 1 i)(i 4)])
    (op 23 st_i [] [(r 14 i)(r 15 i)(r 13 i)] <(ARRAY_ACCESS_ATTR)>)
```

IMPACT - ARA Implementation

- **In progress:**
  - **Implementing Live Range Growth.**

- **Future Steps:**
  - **Convert array access instructions on auto-increment / update instructions;**
  - **Evaluate performance;**
  - **Extension: use of modifier registers.**