Iterative Data flow analysis

Iterative solution to data-flow equations

- Approach is to calculate the in() and out() simultaneously for each node of the flow graph
- Gen() and kill() are essentially same
- Previously, we talked about forward equations which calculates out() based on in()
- The approach in here discuss a backward equation wherein in() is computed assuming out() and again out() is computed based on the new in()

Iterative solution to data-flow equations

- Confluence operator is “Union” function which computes the union of all definitions reaching a point

Iterative algorithm – reaching definitions

- in[B] = U out[P] for all P being a predecessor of B
- out [B] = gen [B] U {in[B] – kill [B]}
- A graph with n blocks has 2n equations which are solved assuming they are recurrences

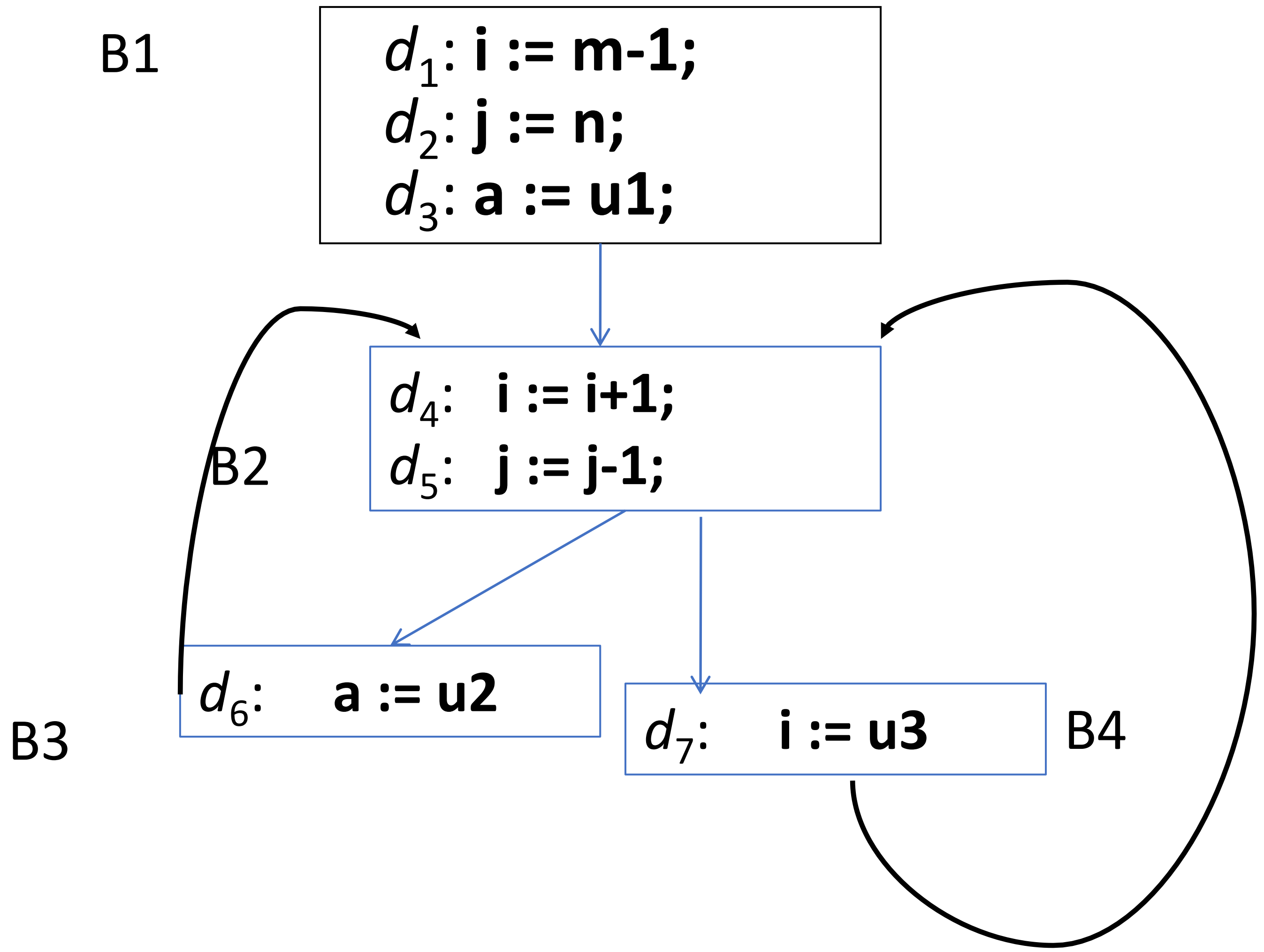
Algorithm

- Input: A flow graph for which kill[B] and gen[B] have been computed
- Output: in[B] and out[B] for each block B
- Approach: in[B] is Φ for all B and converging to the desired values of in[] and out[].

Algorithm

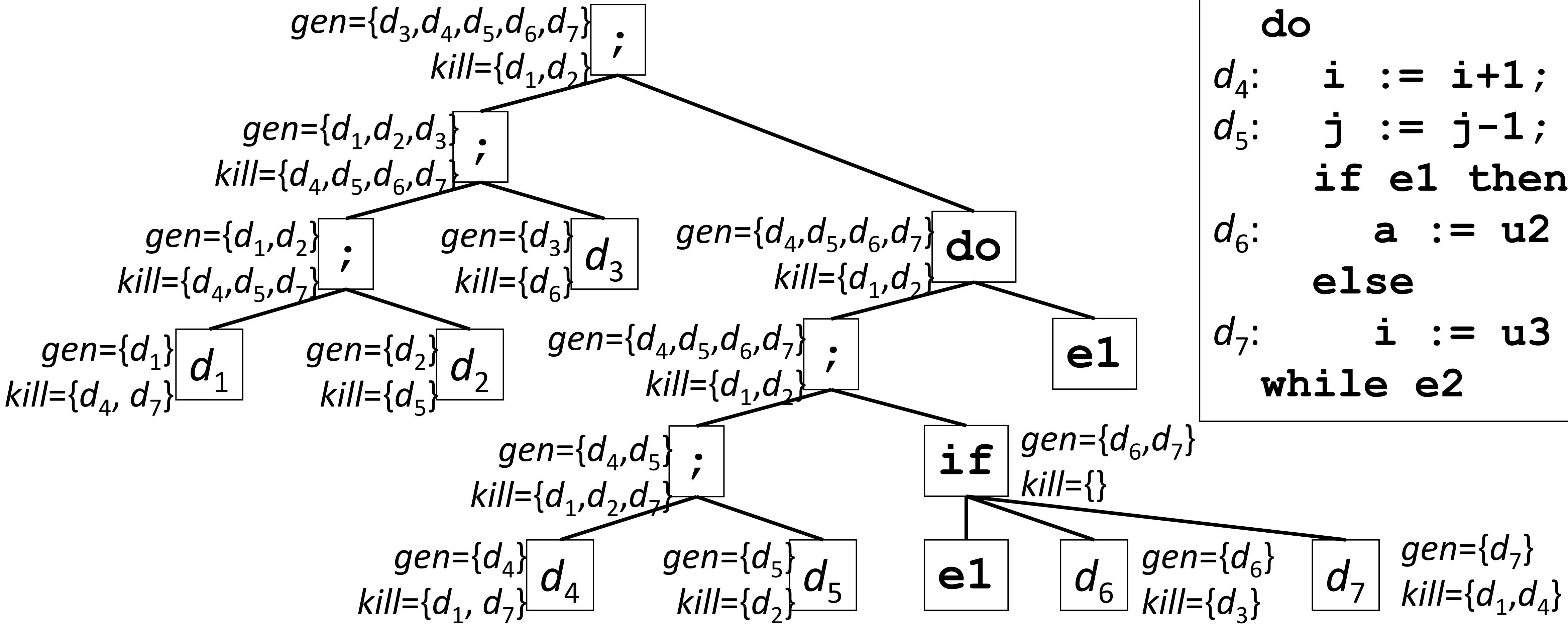
```
Initialize in[B] = Φ for all blocks B
for each block B
    out[B] = gen[B]
change := true
while change
        change := false
        for each block B
            in[B] = U out[P] where P is a predecessor
            oldout = out[B]
            out[B] = gen[B] U {in[B] – kill[B]};
            if out[B] ≠ oldout then change := true
        end
    end
```

Example



```
d1: i  := m-1;
d2: j  := n;
d3: a  := u1;
   do
d4:    i  := i+1;
d5:    j  := j-1;
       if e1 then
d6:       a  := u2
       else
d7:       i  := u3
   while e2
```

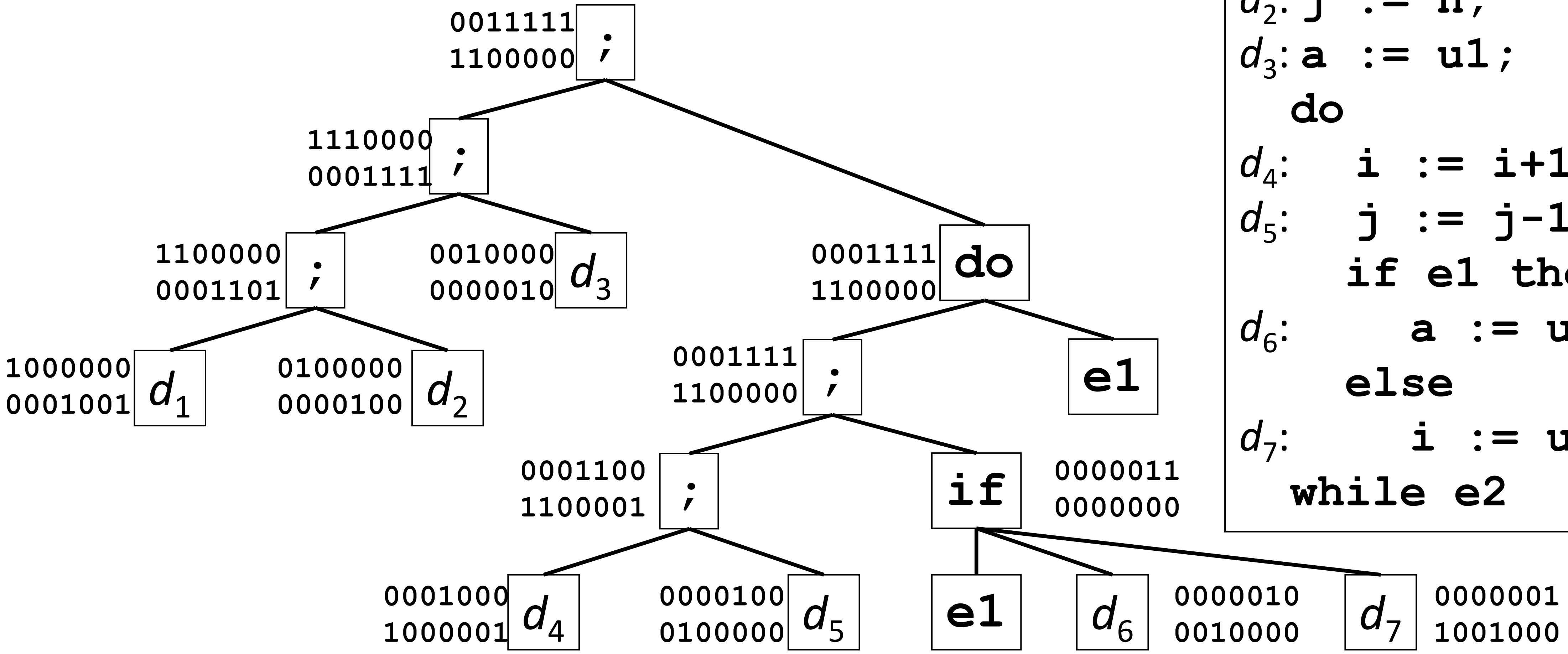
Example Reaching Definitions



gen={d_3,d_4,d_5,d_6,d_7} kill={d_1,d_2} ;
gen={d_1,d_2,d_3} kill={d_4,d_5,d_6,d_7} ;
gen={d_1,d_2} kill={d_4,d_5,d_7} ;
gen={d_3} kill={d_6} d_3
gen={d_4,d_5,d_6,d_7} kill={d_1,d_2} do
gen={d_1} kill={d_4, d_7} d_1
gen={d_2} kill={d_5} d_2
gen={d_4,d_5,d_6,d_7} kill={d_1,d_2} ;
e1
gen={d_4,d_5} kill={d_1,d_2,d_7} ;
if gen={d_6,d_7} kill={}
gen={d_4} kill={d_1, d_7} d_4
gen={d_5} kill={d_2} d_5
e1
d_6 gen={d_6} kill={d_3}
d_7 gen={d_7} kill={d_1,d_4}

```
d1: i  := m-1;
d2: j  := n;
d3: a  := u1;
  do
d4:   i  := i+1;
d5:   j  := j-1;
      if e1 then
d6:      a  := u2
      else
d7:      i  := u3
  while e2
```

Using Bit-Vectors to Compute Reaching Definitions



```
d1: i  := m-1;
d2: j  := n;
d3: a  := u1;
  do
d4:   i  := i+1;
d5:   j  := j-1;
      if e1 then
d6:      a  := u2
      else
d7:      i  := u3
  while e2
```

Example

Block	Gen	Kill
B1	{d1, d2, d3} – {111 0000}	{d4, d5, d6, d7} -
B2	{d4, d5} – {000 1100}	{d1, d2, d7}
B3	{d6} – {000 0010}	{d3}
B4	{d7} – {000 0001}	{d1, d4}

Example

- out [B2] = 000 1100
- in[B2] = out[B1] U out[B3] U out[B4]

  = 111 0000 + 000 0010 + 000 0001

  = 111 0011
- out [B2] = gen [B2] U {in[B2] – kill [B2]}

  = 000 1100 + (1110011 – 110 0001)

  = 000 1100 + (111 0011 & 001 1110)

  = 000 1100 + (001 0010) = 001 1110

Example

Block	**Initial**		**PASS 1**		**PASS 2**	
	in [B]	out [B]	in [B]	out [B]	in [B]	out [B]
B1	000 0000	111 0000	000 0000	111 0000	000 0000	111 0000
B2	000 0000	000 1100	111 0011	001 1110	111 1111	001 1110
B3	000 0000	000 0010	001 1110	000 1110	001 1110	000 1110
B4	000 0000	000 0001	001 1110	001 0111	001 1110	001 0111

Available Expression

- Expression x+y is available at a point ‘p’ if every path from the initial node to ‘p’ computes x+y and after the last evaluation there are no assignments to x, y
- Killing expressions is not the same as reaching definitions
- Used for identifying CSE

Available Expression

- Compute the set of generated expressions for each point in a block from the beginning to end of the block
- At the point prior to the block assume no expressions are available

Available expressions

- At point p if A – set of available expressions, and q is the point after p, where x: = y+z is done, we form expression available at A as
  - Add to A the expression y+z
  - Delete from A any expression involving x
- Same order should be maintained

Example

Statements	Available expressions
….	none
a := b+c	
….	b+c
b := a – d	
….	only a-d
c := b+c	
….	only a-d
d := a-d	
..	none

Algorithm for Available expressions

- U – set of all expressions appearing on the right of one or more statements
- e_gen[B] – generated by a block B
- e_kill[B] – set of expression in U killed by B
- out[B] = e_gen[B] U {in[B] – e_kill[B]}

Available Expression - algorithm

- in [B] = ∩ out[P] for P is the predecessor block
  - An expression is available at the input of a block only if it is available at the end of all its predecessors
- in[B1] = Φ

Algorithm – Available expressions

- Input: A flow graph G with e-kill[B] and e-gen[B] for each block
- Output: set in[B] for each block B

Algorithm

```
Initialize in[B1] = Φ
out[B1] = e_gen[B1]
for B ≠ B1 do out[B] := U – e_kill[B]
change := true
while change
            change := false
            for each block B ≠ B1
                  in[B] = ∩ out[P] where P is a predecessor
                  oldout = out[B]
                  out[B] =e_gen[B] U {in[B] – e_kill[B]};
                  if out[B] ≠ oldout then change := true
            end
     end
```

Live-Variable analysis

- Determine whether for any variable ‘x’ and point ‘p’ whether the value of x at p could be used along some path in the flow graph starting at p.
  - x is live at p else x is dead at p

Live-variable analysis

- in[B] - set of variables live at the point immediately before block B
- Out[B] – same at the point immediately after the block
- Def[B] – definitely assigned values in B prior to any use of that variable in B
- Use[B] – set of variables whose values may be used in B prior to definition of the variable

Live-variable analysis

- in[B] = use[B] U {out[B] – def[B]}
  - Variable is live coming into a block if either it is used before redefinition in the block or it is live coming out of the block and is not defined in the block
- out[B] = U in[S] for all S being successor of B
  - Variable is live coming out of a block iff it is live coming into one of its successors

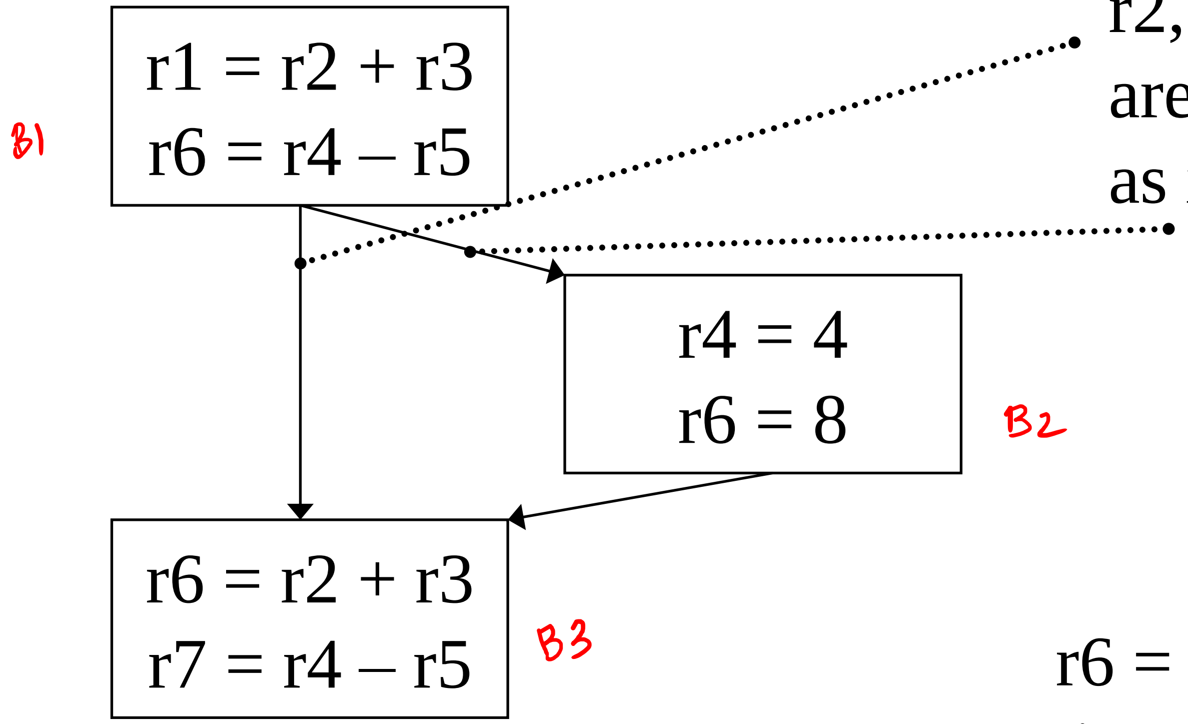
Algorithm

- Input: A flow graph with def and use computed for each block
- Output: out[B] – the set of variable live on exit from each block B

Algorithm

```
for each block B do in[B] = Φ
while changes to any of the in's occur do
  for each block B do begin
                  out[B] = U in[S]
                  in[B] := use[B] U (out[B] – def[B])
```

Example: Liveness



r2, r3, r4, r5 are all live as they are consumed later, r6 is dead as it is redefined later

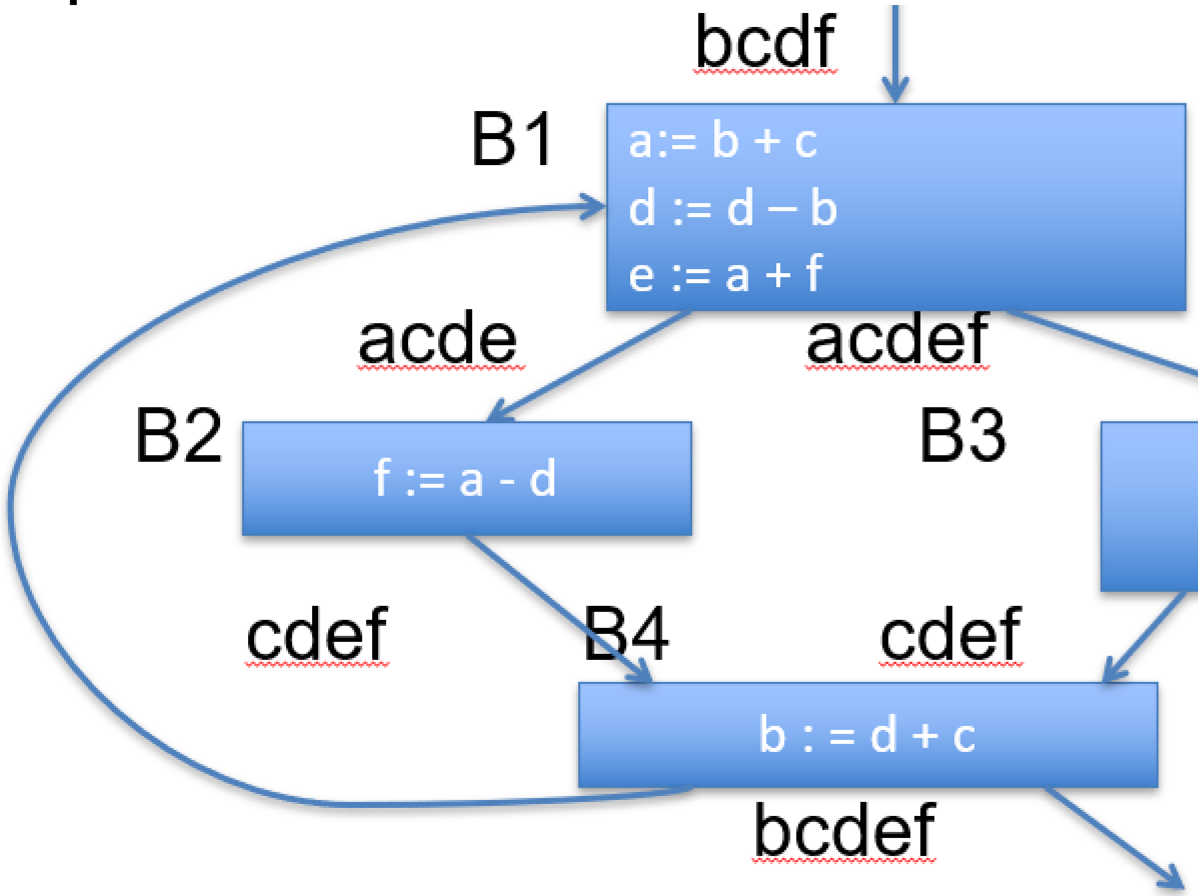
r4 is dead, as it is redefined. So is r6. r2, r3, r5 are live

r6 = r4 – r5 is useless,
it produces a dead value !!
Eliminate it

Computation of use [ ] and def[ ]

```
for each basic block BB do
    def[B] = $\Phi$ ;    use[B] = $\Phi$ ;
    for each statement (x := y op z) in sequential order, do
        for each operand y, do
            if (y not in def[B])
                use[B] = use[B] $\cup$ {y};
        endfor
        def[B] = def[B] $\cup$ {x};
    endfor
```

Example



Block	Use	Def
B1	{b, c, d, f}	{a, d, e}
B2	{a, d}	{f}
B3	{a, c, d, f}	{b, e}
B4	{c, d}	{b}

Example

- use[B1] = {b, c, d, f}
- out[B1] = Φ
- def[B1] = {a, d, e}
- in[B1] =

  = {b, c, d, f} U {{a, c, d, e, f} – { a, d, e}}

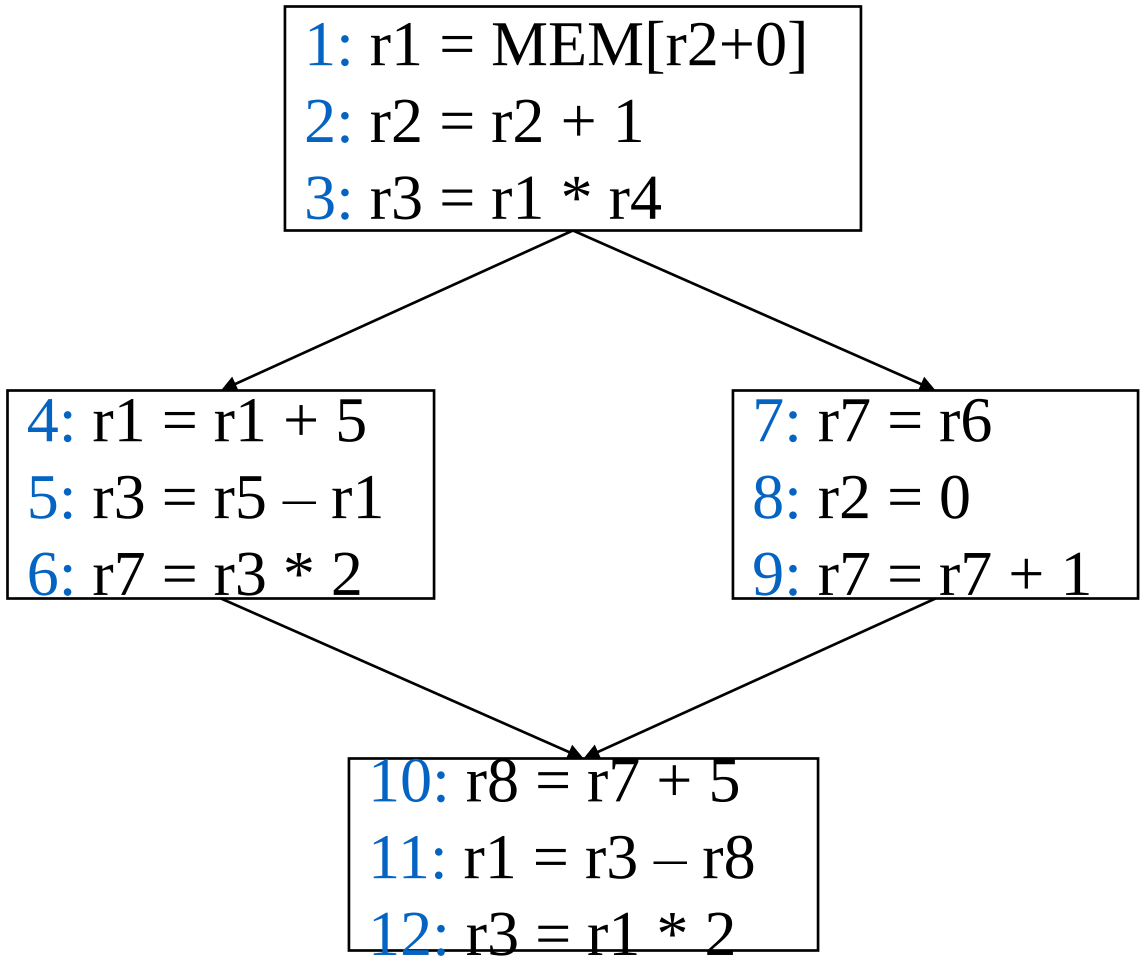
  = {b, c, d, f} U {c, f}   = {b, c, d, f}

DU/UD Chains

- Convenient way to access/use reaching definition information.
- Def-Use chains (DU chains)
  - Given a **def**, what are all the possible consumers of the definition produced
- Use-Def chains (UD chains)
  - Given a **use**, what are all the possible producers of the definition consumed

$\underline{x} = \underline{y + z}$

Example: DU/UD Chains



```
1: r1 = MEM[r2+0]
2: r2 = r2 + 1
3: r3 = r1 * r4

4: r1 = r1 + 5
5: r3 = r5 – r1
6: r7 = r3 * 2

7: r7 = r6
8: r2 = 0
9: r7 = r7 + 1

10: r8 = r7 + 5
11: r1 = r3 – r8
12: r3 = r1 * 2
```

DU Chain of r1:
- (1) -> 3,4
- (4) ->5

DU Chain of r3:
- (3) -> 11
- (5) -> 11
- (12) ->

UD Chain of r1:
- (12) -> 11

UD Chain of r7:
- (10) -> 6,9

Global Common Sub-expression elimination

- Available expression algorithm computes and informs whether an expression is common at point p
- After identifying this, we need to get rid of the common sub-expression

Algorithm

- Input: A flow graph with available expression information
- Output: A revised flow graph
- Method: For every statement ‘s’ of the form x:=y+z if y+z is available at the beginning of ‘s’ do the following

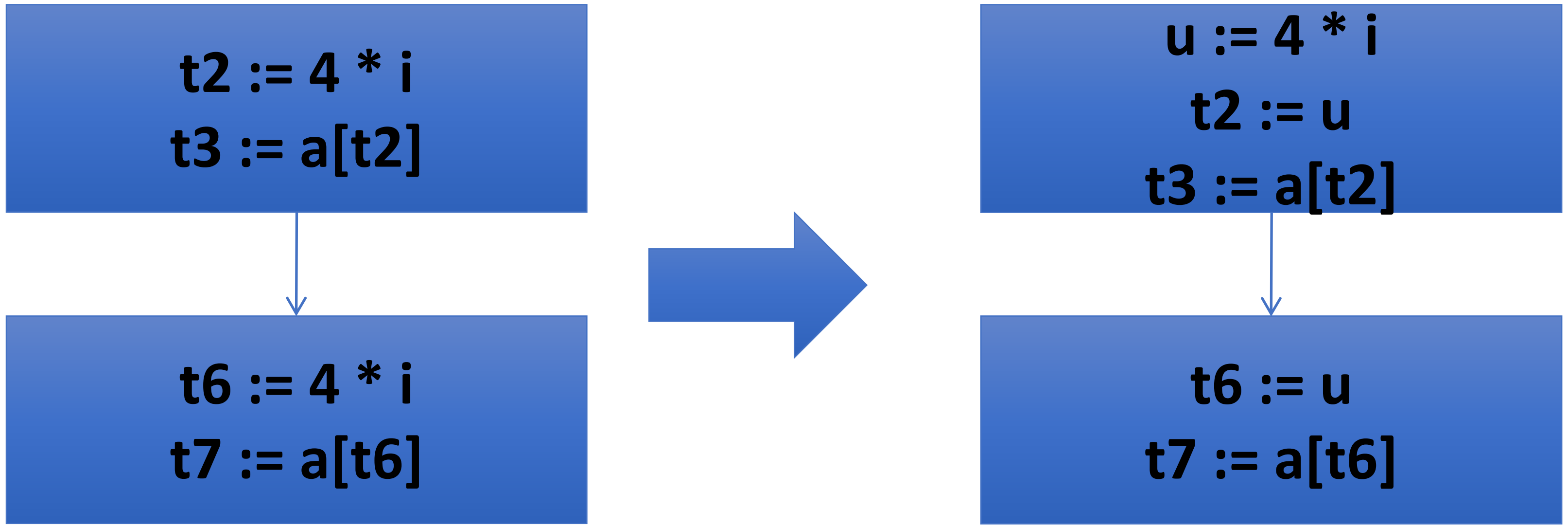
Algorithm

- Search backward from ‘s’ the expression ‘y+z’
- Create a new variable ‘u’
- Replace each statement w:=y+z found in (1) by
  - u := y+z
  - w:= u
- Replace statement ‘s’ by x := u

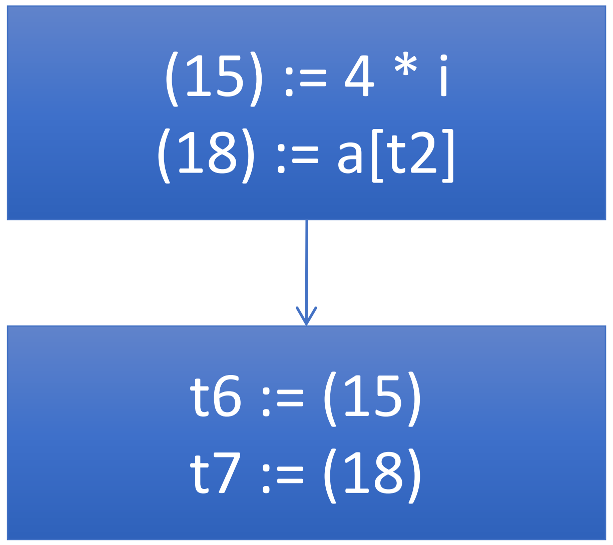
Observations of the algorithm

- Searching for ‘y+z’ can be done as a data-flow analysis problem
- Need not be optimized.. Results in copy statements
- The following is not handled – ‘b’ and ‘d’ are also same
    - a := x+y   vs   c := x+y
    - b := a*z   d := c*z

Example



Example – Value numbering based



Copy Propagation

- Copies gets generated due to elimination of common sub-expression
- S: $\underline{x:= y}$ ✓ $y \overset{u}{=} \boxed{l+z}$
- Determine where the value of ‘x’ is used
- Substitute ‘y’ in place of ‘x’ where ‘u’ is a statement that uses y

Copy Propagation

- Statement ‘s’ must be the only definition of x reaching u – UD chain could be used
- On every path from ‘s’ to ‘u’, there are no assignments to ‘y’ – new data-flow analysis problem

Forward Copy Propagation

- Forward propagation of RHS of assignment or mov’s.

```
r1  := r2
    .
    .
    .
r4  := r1 + 1
```

$\longrightarrow$

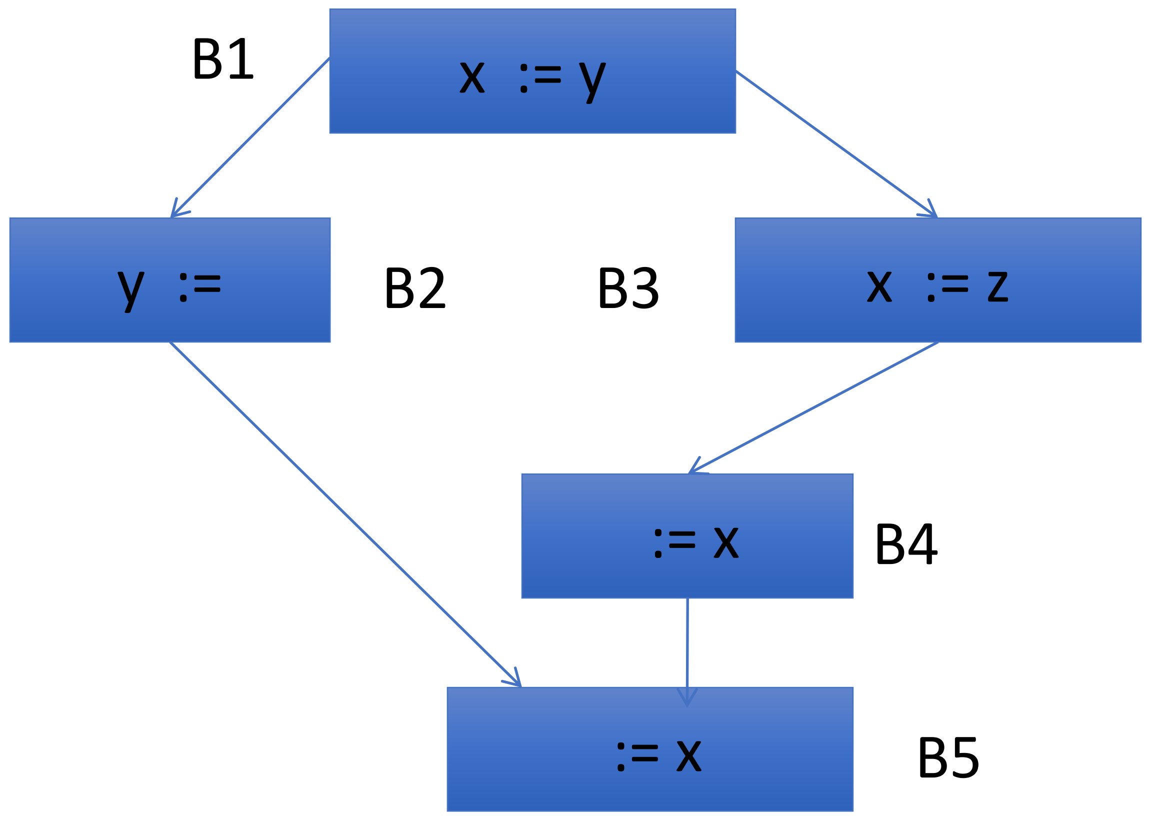
```
r1  := r2
    .
    .
    .
r4  := r2 + 1
```

  - Reduce chain of dependency
  - Possibly create dead code

Copy Propagation

- out[B] = c_gen[B] U (in[B] – c_kill[B])
- in[B] = ∩ out[P] where P is a predecessor block
- in[B1] = Φ – B1 is the initial block
- This is similar to the available expressions algorithm and the computation are also same.

Example



c_gen [B1] = {x := y }
c_gen[B3] = { x:= z}
c_kill [B2] = {x:= y}
c_kill[B1] = {x:=z}
c_kill[B3] = {x := y}
All other c_gen and
c_kill are Φ
No copy of x:=y or x:= z
reaches B5

Algorithm – Copy Propagation

- Input: A flow graph with ud-chains and c_in[B], du-chains that has the use of all definitions
- Output: revised flow graph

Algorithm

- For each copy ‘s’ x:=y, do the following
  - Determine the uses of ‘x’ that are reached by this definition
  - Determine whether for every use of ‘x’, ‘s’ is in c_in[B] where B is the block for this use and no definitions of ‘x’ or ‘y’ occur prior to this use
  - If the statement meets this condition, remove ‘s’ and replace all uses of ‘x’ by ‘y’

Constant Propagation

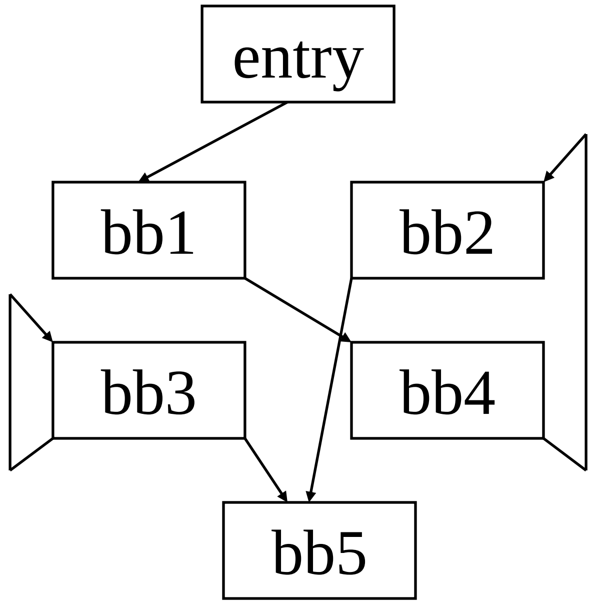
- Forward propagation of moves/assignment of the form

  d: `rx := L` where `L` is literal

  - Replacement of “rx” with “L” wherever possible.
  - d must be available at point of replacement.

Unreachable Code Elimination

```
Mark initial BB visited
to_visit = initial BB
while (to_visit not empty)
    current = to_visit.pop()
    for each successor block of current
        Mark successor as visited;
        to_visit += successor
    endfor
endwhile
Eliminate all unvisited blocks
```



Which BB(s) can be deleted?

Loop invariant computation

- UD-chains could be used to find out values that does not change as long as control stays within the loop
- Loop have at least one way to get back to the header from any block in the loop
- If x:= y+z is at a position in the loop and all possible definitions of ‘y’ and ‘z’ are outside the loop then y+z is loop invariant

Algorithm

- Input: A loop L consisting of a set of basic blocks having three address statements
- The set of three-address statements that compute the same value each time executed, from the time control enters the loop L until control leaves L

Algorithm

- Mark the statements whose operands are all either constants or have all reaching definitions outside L as ‘invariant’
- Repeat the following step until at some repetition no new statements are marked ‘invariant’
- Mark ‘invariant’ all those statements not previously so marked all of whose operands are either constant or having definitions reaching outside ‘L’ or have only one reaching definition which is marked invariant

Performing Code motion

- Applied to statements found to be loop invariant
- Statements are moved to the pre-header of the loop
- Some conditions need to be checked and applied to perform code motion

Conditions ‘s’ x:= y+z

- Block containing a statement ‘s’ must dominate all exit nodes of the loop – a successor node that is not in the loop
- No other statement in the loop assigns to ‘x’
- No use of ‘x’ in the loop is reached by any definition of x other than ‘s’.

Algorithm

- Input: a loop L with ud-chain information and dominator information
- Output: a revised loop with a pre-header and some statements moved to the pre-header (if any)

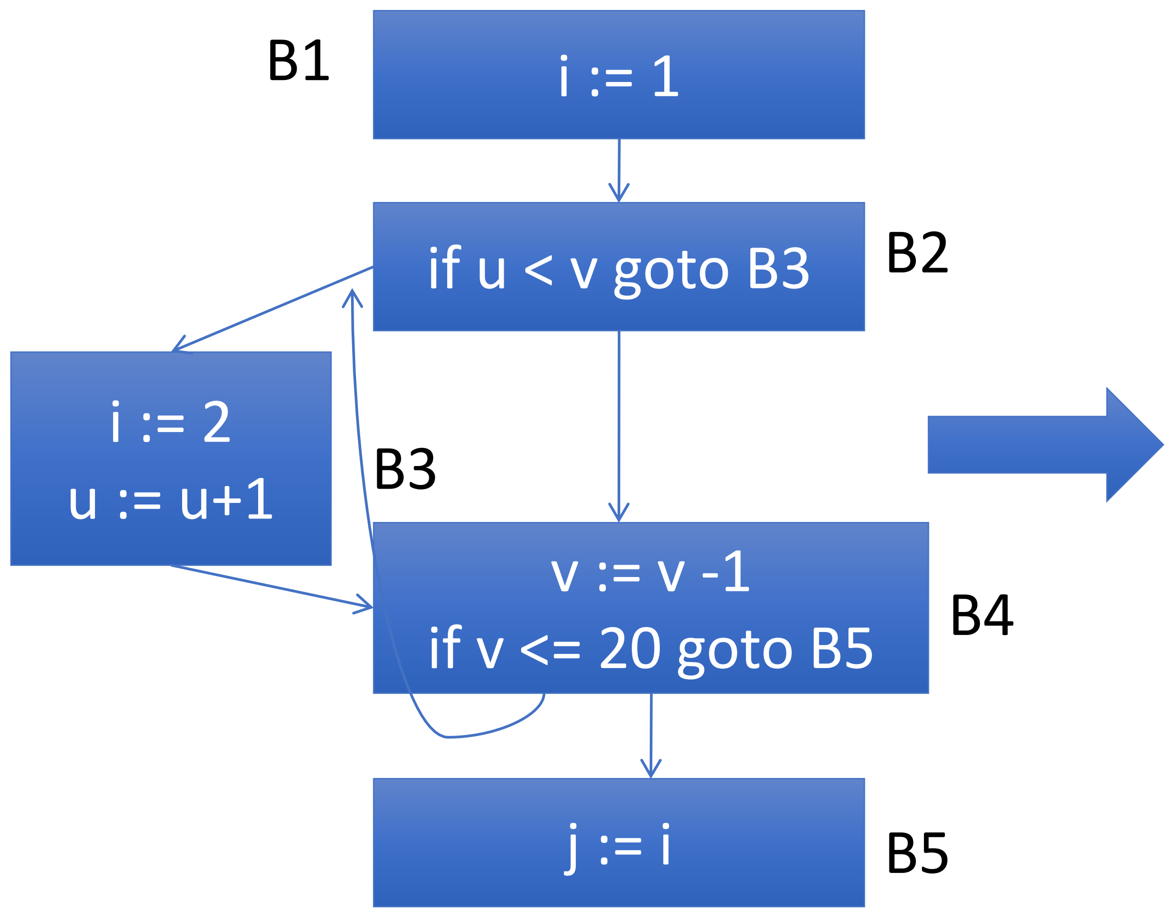
Algorithm

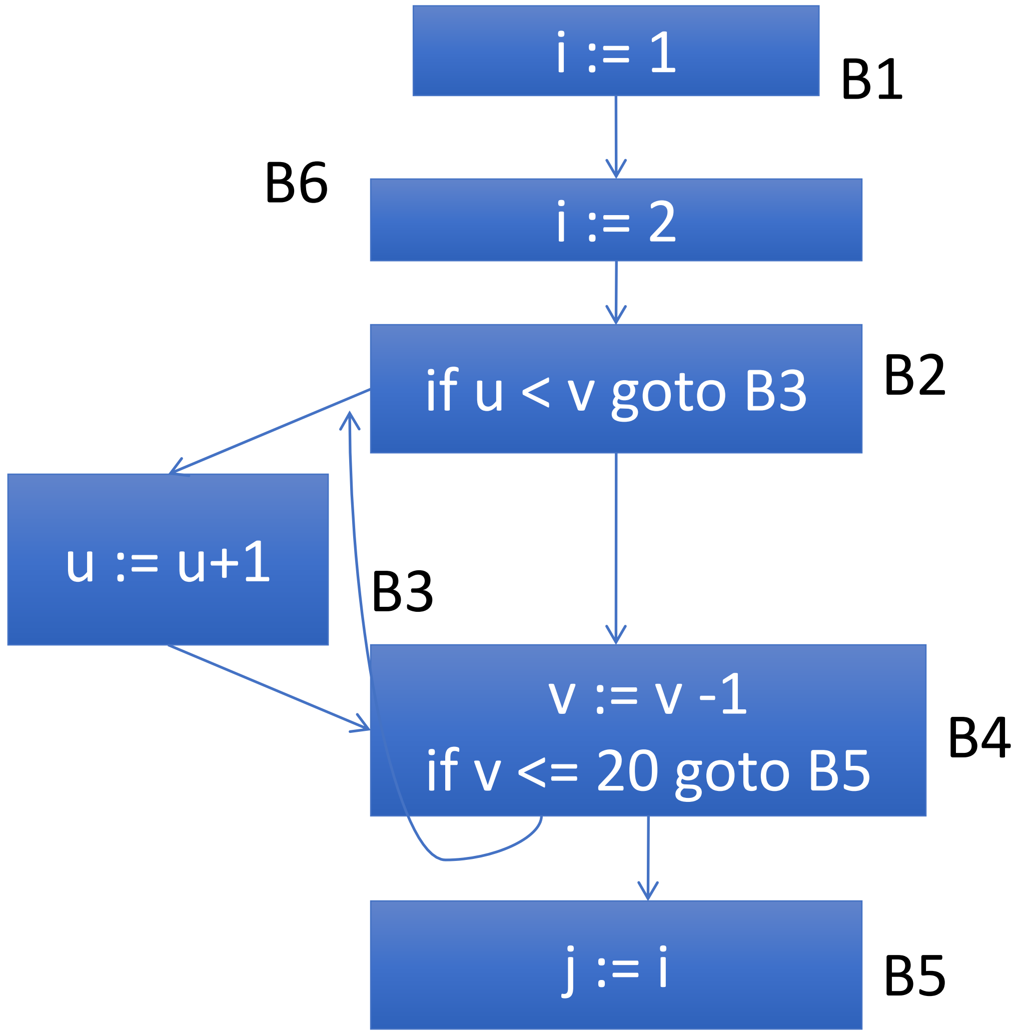
- Identify the loop invariant statements defining ‘x’
- For each statement ‘s’ defining ‘x’ find
  - That it is in a block that dominates all exits of L
  - That ‘x’ is not defined elsewhere in L
  - All uses in L of x can only be reached using statement ‘s’

Algorithm

- Move, in the order found by loop invariant algorithm, each statement ‘s’ to a newly created pre-header

Illegal code motion





Loop Transformation and Aliases

Induction variable elimination

- A variable ‘x’ is induction if every time the value of ‘x’ is changed by a constant ‘c’
- Look for basic induction variable i := i +/- c
- Look for derived induction variable ‘j’ which are defined in terms of the basic ‘i'

Induction variable identification algorithm

- Input: A loop L with reaching definitions and loop-invariant computation
- Output: a set of induction variables

Algorithm

- Find basic induction variable based on loop-invariant computation (i,I, 0)
- Search for a variable k having the following forms
  - k := j * b, k := b*j, k := j/b, k:= j +/-b, k:= b +/-j
  - b – constant, j is an induction variable

Algorithm

- Triple for k is (j, b, 0)
- Compute the triple and accumulate to the list of inductions variables
- Modify them to use additions / subtractions as against multiplication / division
- Replace it and this is called strength reduction

Strength reduction – induction variables

- Input: A loop L with reaching definition information and induction variables computed
- Output: A revised loop

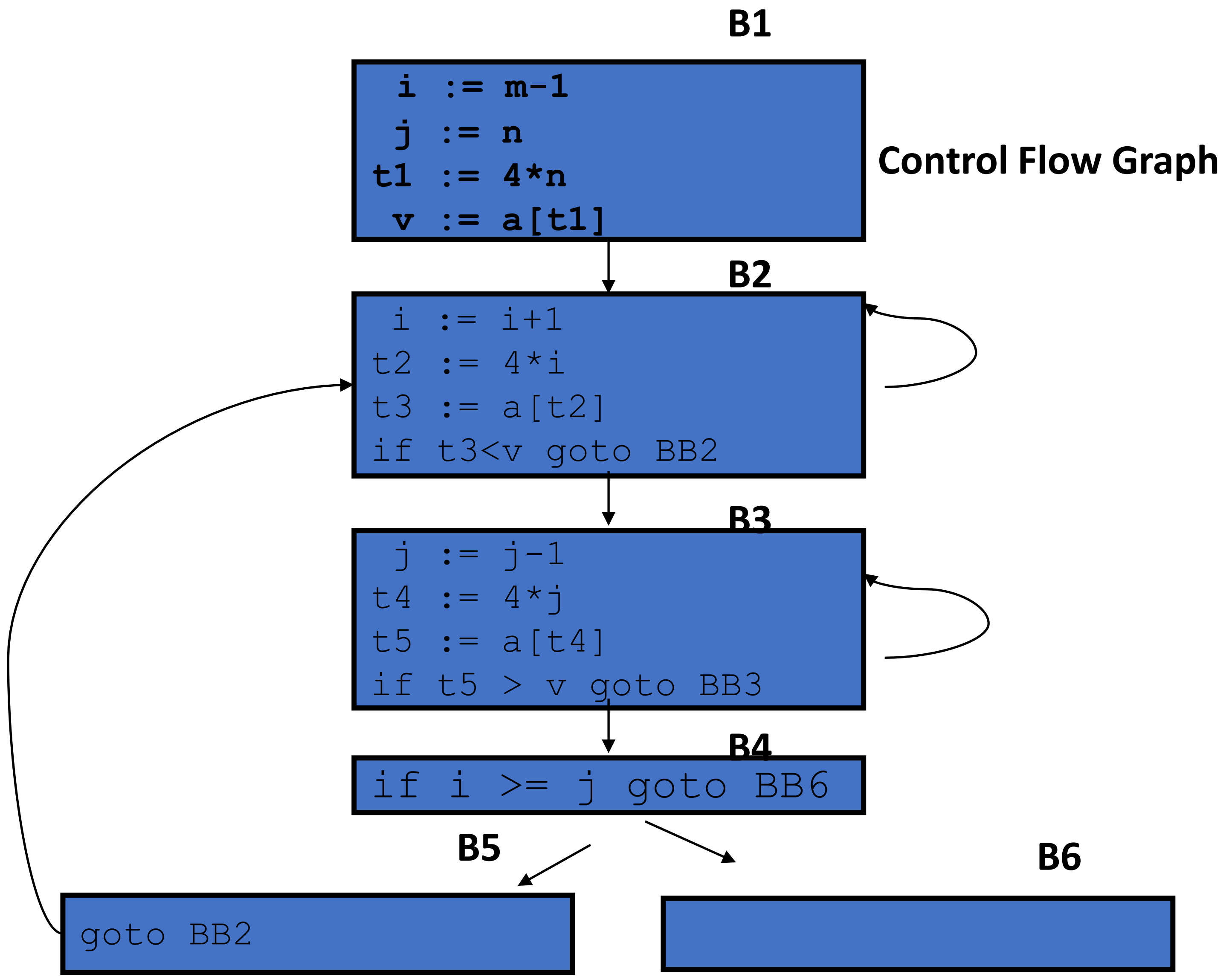
Algorithm

- For each induction variable i in turn, for every induction variable j in the family of i with triple (i, c, d): (j := i *c +d)
- Create a new variable ‘s’
- Replace the assignment to j by j:=s
- Immediately after each assignment i := i+n, append
    - s := s + c* n

Algorithm

- Place ‘s’ in the family of ‘i‘ with triple (i,c,d)
- s is initialized to c*i+d
  - s := c * i
  - s := s+d

Quicksort CFG



```
 i  := m-1
 j  := n
t1  := 4*n
 v  := a[t1]
```

```
 i  := i+1
t2  := 4*i
t3  := a[t2]
if t3<v goto BB2
```

```
 j  := j-1
t4  := 4*j
t5  := a[t4]
if t5 > v goto BB3
```

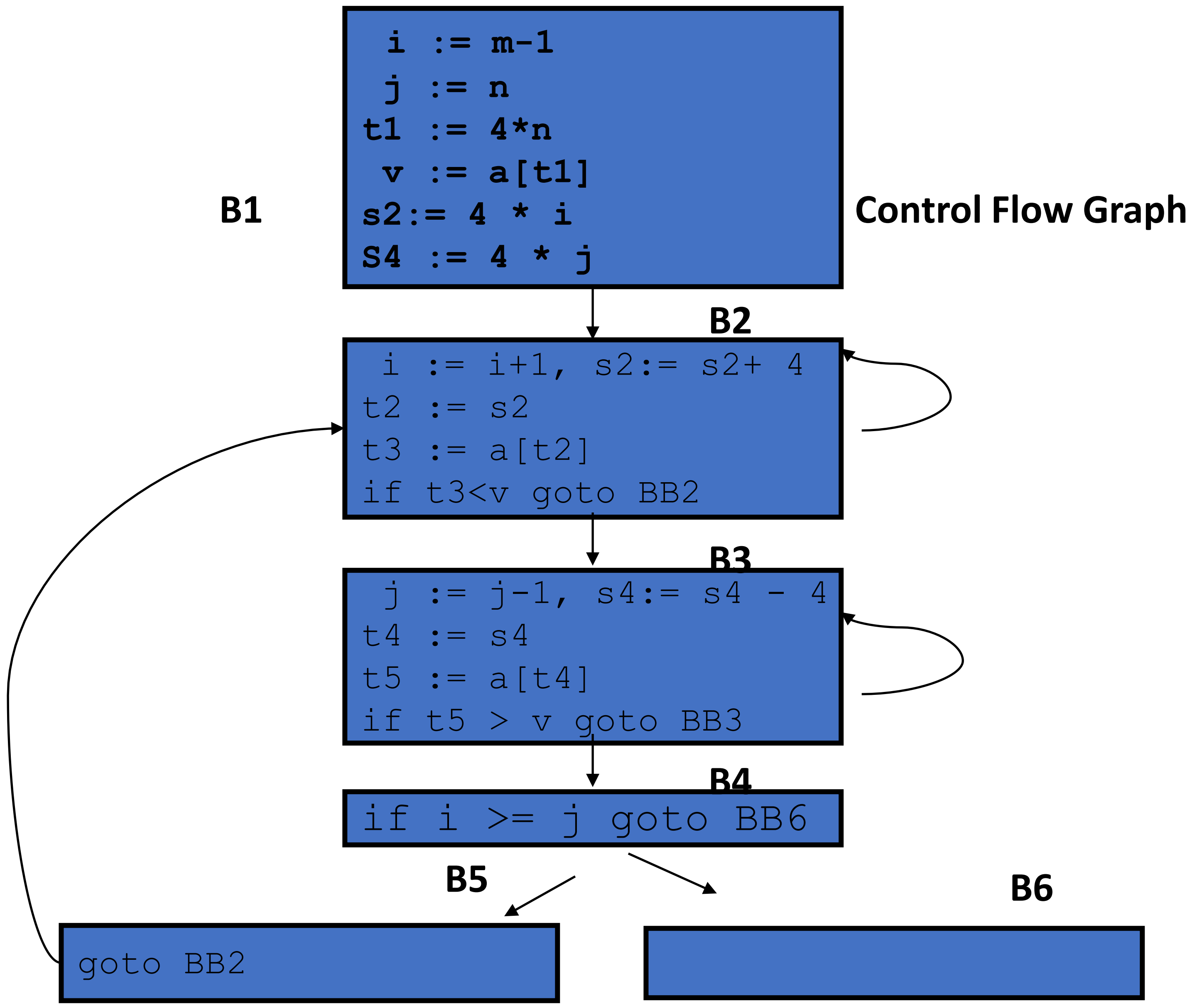
```
if i >= j goto BB6
```

```
goto BB2
```

Example

- B2 and B3 are inner loops
- Induction variable in B3 is ‘j’ and t4 (j, 4, 0)
- A new variable is construction s4
- t4 := 4 * j is replaced with t4:= s4
- Inserts the assignment s4 := s4 – 4 after j:= j-1

Quicksort CFG



```
 i  := m-1
 j  := n
t1  := 4*n
 v  := a[t1]
s2:= 4 * i
S4  := 4 * j
```

```
 i  := i+1, s2:= s2+ 4
t2  := s2
t3  := a[t2]
if t3<v goto BB2
```

```
 j  := j-1, s4:= s4 - 4
t4  := s4
t5  := a[t4]
if t5 > v goto BB3
```

```
if i >= j goto BB6
```

```
goto BB2
```

Elimination of induction variable

- Input: A loop L with reaching definition information, loop-invariant computation and live variable information
- Output: a revised loop

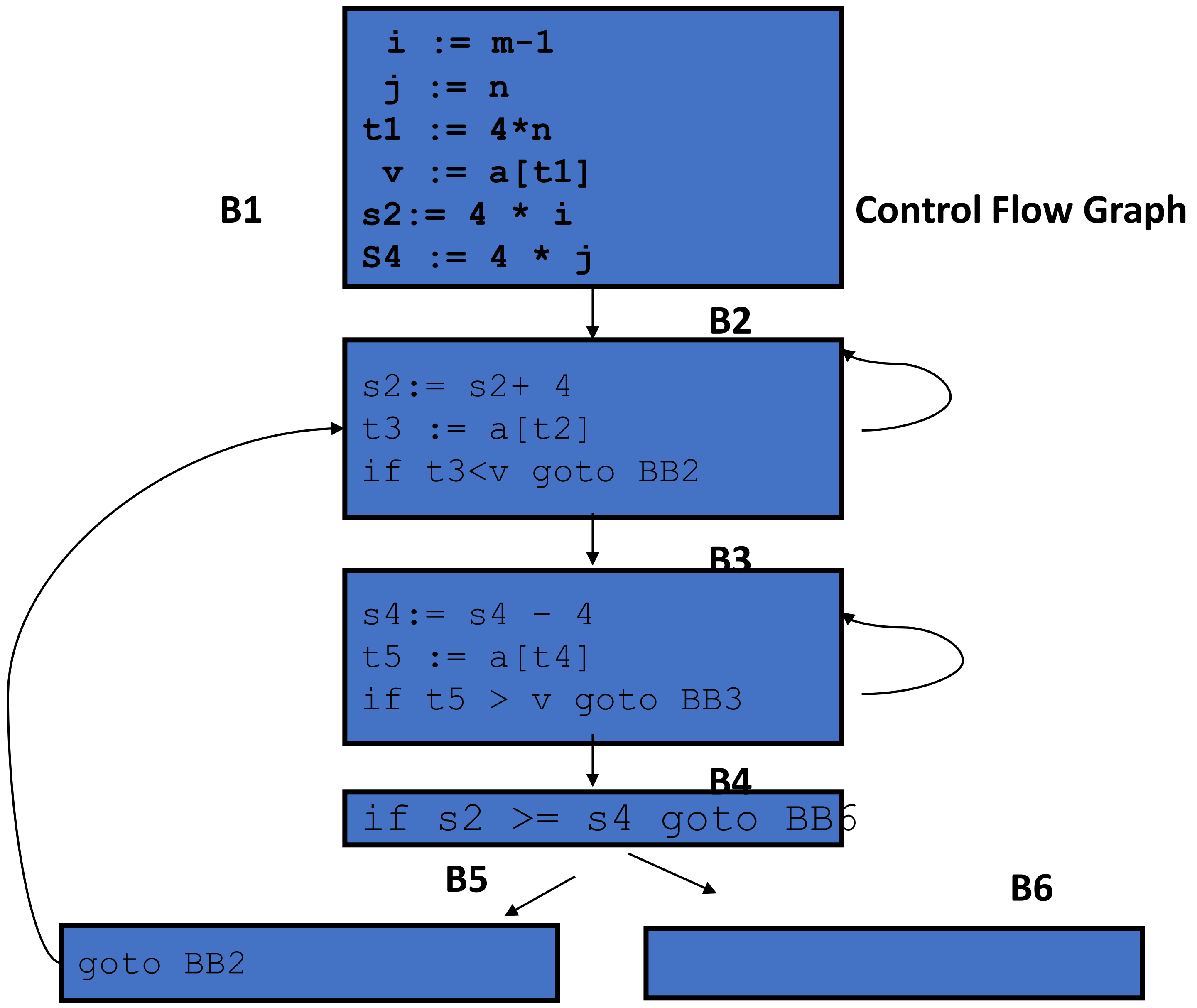
Algorithm

- Take some induction variable ‘j’ in ‘i‘s family with (i,c,d) and modify each test that ‘i‘ appears in to use ‘j’ instead. ‘c’ is positive.
  - if i relop x goto B is replaced as
  - r := c*x, r := r+d, if j relop r goto B

Algorithm

- if i1 relop i2 is also replace with new variable if j1 relop j2
- Delete all assignments to the eliminated induction variables from the loop L
- Consider every new statement j:= s
- Verify that no assignment to ‘s’ between the introduced statement and the use of ‘j’
- Replace all uses of j by uses of ‘s’ and delete j := s

Quicksort CFG



Dealing with aliases

- If two or more expressions denote the same memory address we say that the expressions are aliases of one another
- Presence of pointers makes data-flow analysis more complex
- Pointer p can point to is to assume that an indirect assignment through a pointer can potentially change any variable

$p = \&a$

$x = \underline{a + b}$

$*p += 1$

$y = \underline{a + b}$

Dealing with aliases

- Consider a language having preliminary data types
- If pointer ‘p’ points to a primitive data element, then any arithmetic operation on ‘p’ produces a value that may be an integer

Dealing with aliases

- If ‘p’ points to an array, addition/subtractive leads to ‘p’ somewhere in the array
- If ‘p’ points to other array, then the impact of this would have to be dealt by the optimizing compiler

Effects of pointer assignments

- Variables that could possibly be used as pointers are those declared to be pointers and temporaries that receive a value is a pointer plus or minus a constant

Pointer can point to

- If there is an assignment s: p := & a then immediately after s, ‘p’ points only to ‘a’. If a is an array, then p can point only to a after assignment of the form p := &a +/-c, where &a refers to &a[0]

Pointer can point to

- If there is an assignment, s: p:= q +/-c , p and q are pointers, then immediately after s, p can point to any array that q could point to before ‘s’
- If there is an assignment, s: q := p, p points to what q points to

Pointer can point to

- Any other assignment to p, there is no object that p could point to such an assignment is probably meaningless
- After any assignment to a variable other than p, p points to whatever it did before the assignment

Alias computation

- in[B] – (p, a) – set of variables {a} to which p could point at the beginning of B
- $trans_B$ – transfer function that defines the effect of block B
    - Takes a set of pairs, S of the form (p,a) and produces another set T

Alias computation

- $trans_B$ is computed for every statement and $trans_B$ is the union of $trans_S$

Rules for computing $trans_S$

- if s: p:= & a or p:= &a+/- c, where ‘a’ is array then
  - $trans_S$ (S) = (S – {(p,b) | any variable b}) U (p,a)
- If s: p: = q +/- c for pointer q and c is non-zero
  - $trans_S$ (S) = (S – {(p,b) | any variable b}) U {(p,b) | (q,b) is in S and b is any variable}

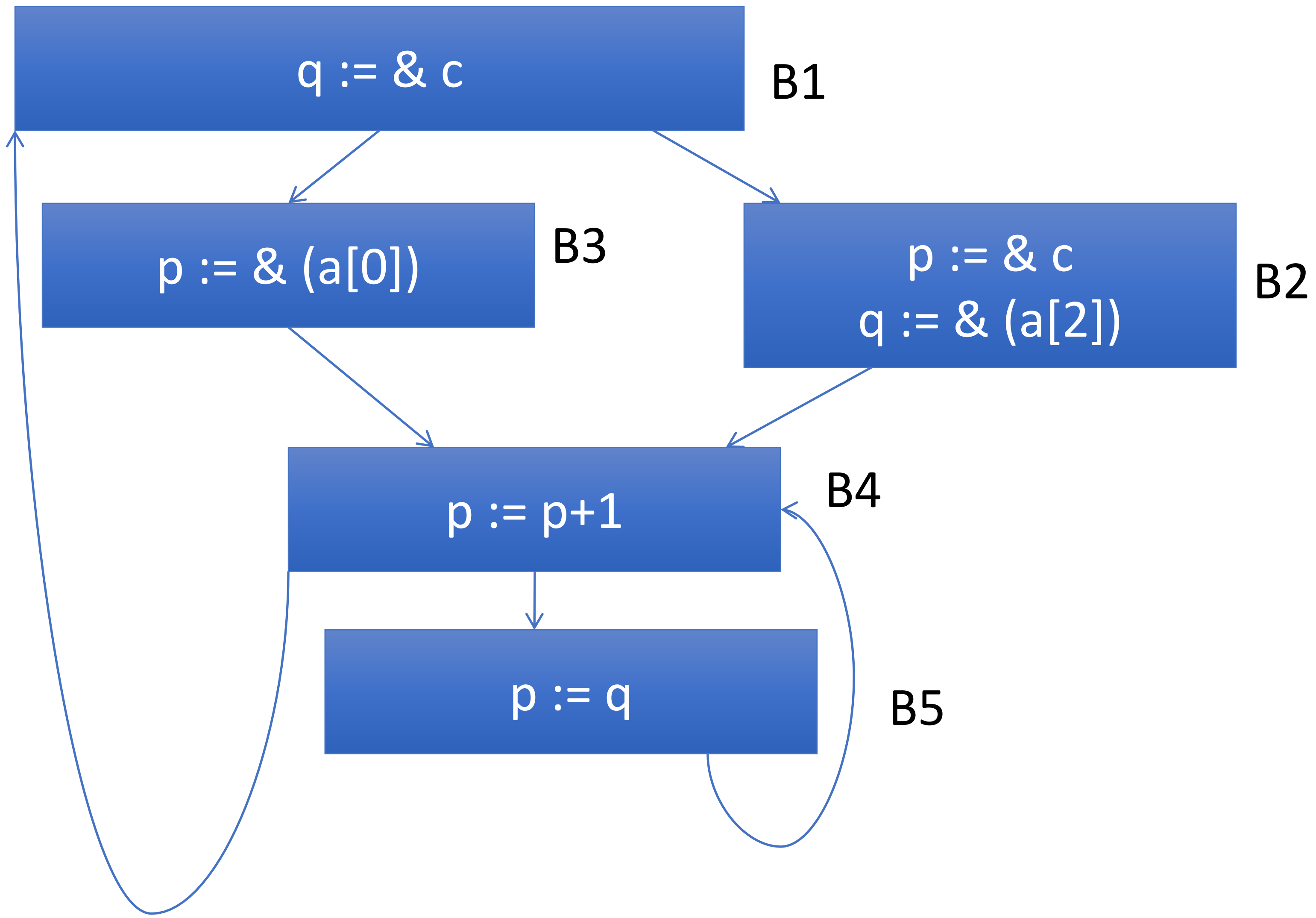
Alias computation

- If s: p:= q then
  - $trans_S$ (S) = (S – {(p,b) | any variable b}) U {(p,b) | (q,b) is in S}
- If s assigns to pointer p another expression then
  - $trans_S$ (S) = (S – {(p,b) | any variable b})
- If s is not an assignment to a pointer then
  - $trans_S$ (S) = S

Data-flow equations

- out[B] = trans_B (in[B])
- in[B] = U out(P) where P is a predecessor block
- trans_B (S) = trans_{sk} (trans_{sk-1} (trans_{sk-2} …)))

Example



Example

- Out[B1] = trans_{B1} (Φ)
- B1 has one statement and hence

Out[B1] = trans_{B1} (Φ) = {(q,c)}

- p:= & c replace all pairs of p with (p,c)
- q replaces (q,a)

Out[B2] = trans_{B2} ((q,c)) = {(p,c), (q,a)}

Example

Block	in[ ]	out [ ]	trans [ ]
B1	Φ	{(q,c)}	{(q,c)}
B2	{(q,c)}	{(p,c), (q,a)}	{(p,c), (q,a)}
B3	{(q,c)}	{(p,a), (q,c)}	{(p,a), (q,c)}
B4	{(p,a), (q,c), (p,c), (q,a)}	{(p,a), (q,c), (q,a)}	{(p,a), (q,c), (q,a)}
B5	{(p,a), (q,c), (q,a)}	{(p,a), (q,c), (p,c), (q,a)}	{(p,a), (q,c), (p,c), (q,a)}

Example – II pass

Block	in[ ]	out [ ]	trans [ ]
B1	Φ	{(p,a), (q,c)}	{(p,a), (q,c)}
B2	{(p,a),(q,c)}	{(p,c), (q,a)}	{(p,c), (q,a)}
B3	{(p,a),(q,c)}	{(p,a), (q,c)}	{(p,a), (q,c)}
B4	{(p,a), (q,c), (p,c), (q,a)}	{(p,a), (q,c), (q,a)}	{(p,a), (q,c), (q,a)}
B5	{(p,a), (q,c), (q,a)}	{(p,a), (q,c), (p,c), (q,a)}	{(p,a), (q,c), (p,c), (q,a)}

Usage

- For live variable analysis
- Dead variable analysis
- Reaching definitions

Summary

- Iterative data flow equations for reaching definitions, available expression and live variable analysis
- Algorithm and examples were discussed
- Algorithm for Common sub-expression and copy propagation
- Identified unreachable code and eliminated
- Code motion and loop invariant computation identified
- Loop optimizations
- Dealing with aliases and its impact