19-03-'19

UNIT-V: MACHINE INDEPENDENT OPTIMIZATION

Machine Independent Optimization

→ Optimization with respect to Quick Sort

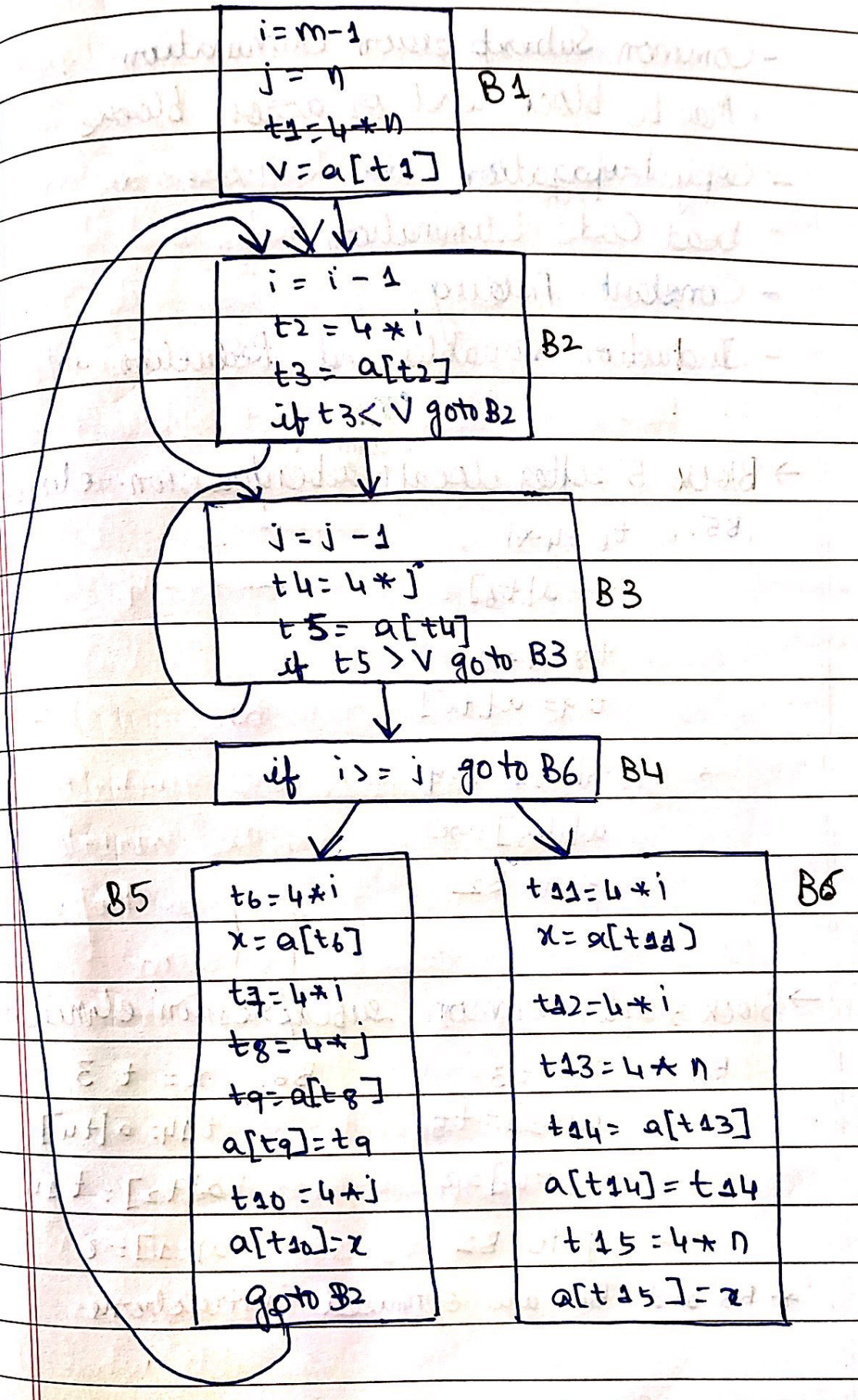
* Quick Sort Algorithm: Text Book

* Three Address Code:

1.	i = m-1	16.	t7 = 4*i
2.	j = n	17.	t8 = 4*j
3.	t1 = 4*n	18.	t9 = a[t8]
4.	v = a[t1]	19.	a[t9] = t9
5.	i = i+1	20.	t10 = 4*j
6.	t2 = 4*i	21.	a[t10] = x
7.	t3 = a[t2]	22.	goto (5)
8.	if t3<v goto (5)	23.	t11 = 4*i
9.	j = j-1	24.	x = a[t11]
10.	t4 = 4*j	25.	t12 = 4*i
11.	t5 = a[t4]	26.	t13 = 4*12
12.	if t5>v goto (9)	27.	t14 = a[t13]
13.	if i>=j goto (23)	28.	a[t1] = t14
14.	t6 = 4*i	29.	t15 = 4*n
15.	x = a[t6]	30.	a[t15] = x

P.T.O. →

* Flow Graph for Quick Sort:



→ Semantic Preserving Transformation

- Common Subexpression Elimination
  - May be block level or across blocks
- Copy Propagation
- Dead Code Elimination
- Constant Folding
- Induction Variables and Reduction in Strength

→ Block 5 after local common subexpression elimination:

B5: $t_6 = 4 * i$
$x = a[t_6]$
$t_8 = 4 * j$
$t_9 = a[t_8]$
$a[t_6] = t_9$
$a[t_8] = x$
goto B2

→ Block 5 after common subexpression elimination

B5: $x = t_3$
$a[t_2] = t_5$
$a[t_4] = x$
goto B2

B6: $x = t_3$
$t_{14} = a[t_4]$
$a[t_2] = t_{14}$
$a[t_1] = x$

* B5 and B6 are optimized instructions

* Common Subexpression Elimination introduces copy instructions.
  eg. $x = t_3$ not used in future => dead variable

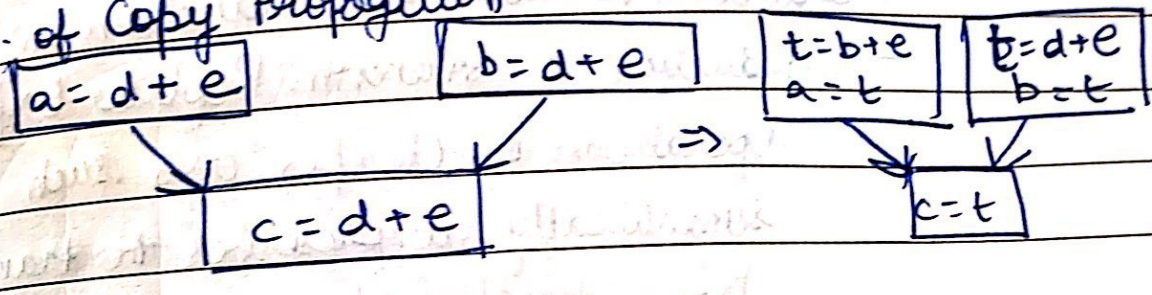
→ After copy propogation

B5: a[t3] = t5

a[t4] = t3

goto B2

→ Eg. of Copy Propogation



→ Dead Code Elimination

Eg. flag = false

if (flag) { } while (flag) { } } Never Executed

* No dead code in our flow graph

→ Code Motions

while (i <= limit - 2)
{
}

Loop Invariant Computation
- Same result independent of the number of times a loop is executed

* Eliminate invariant computations, compute after the loop or before the loop depending on the code.

t = i <= limit - 2

while (t)
{ }

* No code motions in our flow graph.

→ Constant folding

Eg: Replace 2×4 by 8

* No constant folding in our flow graph.

→ Induction Variables and Reduction in Strength

* Reduction in Strength: Replace Expensive Operations by cheaper ones such that semantically it preserves the transformation, i.e before transformation and after transformation the functionality of the code is same.

B3: j = j - 1 —— 3

t4 = 4 * j —— t4 - 4

t3 = a[t4]

if t5 > v goto B3

* Induction variables are generally stored in registers. They are the variables that are used mostly in loops, i.e., the loops depend on them.

B2: i = i - 1

t2 = 4 * i

t3 = a[t2]

if t3 < v goto V2

→ Here i and t2 are induction variables

→

Data Flow Analysis

- At a particular point 'p' in a block, we have to check whether the variable is a live variable or a dead variable.

Available Expressions

- Forward Problem
- ∩
- An expression $x+y$ is available at a point p if every path from the initial node to p evaluation $x+y$, and after like the last such evaluation, prior to reaching P, there are no subsequent assignments to x and y.
- A block 'kills' $x+y$, if it assigns to x or y and does not subsequently recompute $x+y$.
- A block 'generates' $x+y$ if it definitely evaluates $x+y$, and does not subsequently re define x or y.

- Eg.

d1: a = f + 1	d5: b = a + 4
d2: b = a + 7	d6: f = e + c
d3: c = b + d	d7: e = b + d
d4: a = d + c	d8: d = a + b
	d9: a = c + f
	d10: c = e + a

↳ exp in block B → exp out of block

Total: f + 1, a + 7, b + d, d + c, ~~d + c~~, a + 4, e + c, ~~b + d~~, a + b, c + f, e + a

EGEN(B) = {b+1, b+d, d+c}

EKILL(B) = {a+4, e+c, b+d, a+b, e+a, a+7}

* Live Variable : Depends on ~~previous~~ next blocks, i.e, successor blocks
→ so it is backward function.

* Other Variables : Depend on previous blocks, i.e., predecessor blocks
→ so it is forward function.

→ Data Flow Equations:

*Transform Functions for Available Expressions

IN[B] = ∩ OUT[P], B not initial
P is a predecessor of B

OUT[B] = egen[B] ∪ (in[B] - ekill[B])

IN[B1] = ∅

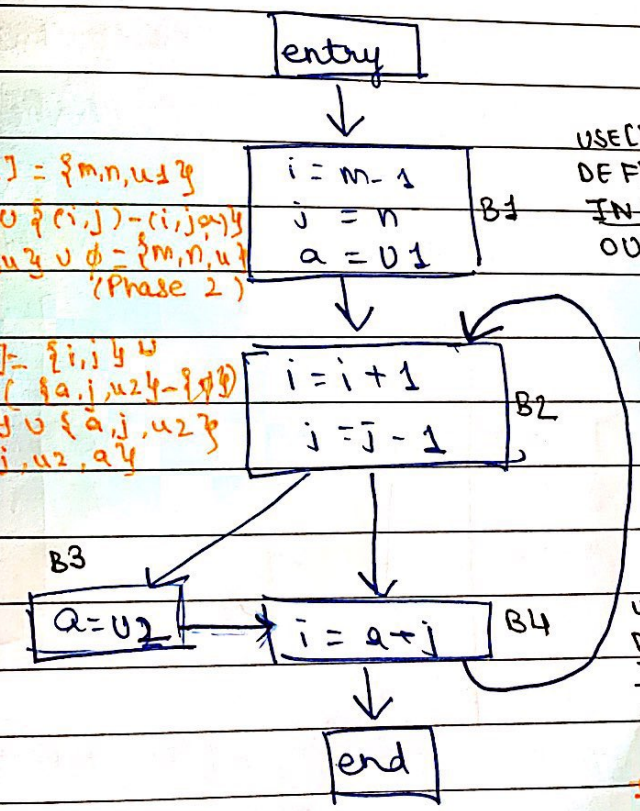
IN[B] = U for all B ≠ B1

B1 - initial (or) entry block and is special : nothing is available when the program execution begins

Live Variable Analysis

- Backward Problem.
- Confluence Operator - ∪
- IN[B] - set of live variables at the beginning of B
  OUT[B] - set of variables live just after B
  DEF[B] - set of variables defined 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 any definition of the variable
- $OUT[B] = \bigcup_{S \text{ is a successor of } B} IN[S]$
  $IN[B] = USE[B] \cup (OUT[B] - DEF[B])$
  $IN[B] = \phi$ for all B (initialization only)

Transfer Functions



USE[B1] = {m, n, u1}
DEF[B1] = {i, j, a}
~~IN[B1] = {m, n, u1} ∪ {φ - DEF[B1]}~~
IN[B1] = {m, n, u1}
OUT[B1] = {φ} (initialization)

Phase 2 = {i, j}

IN[B1] = {m, n, u1}
∪ {(i, j) - (i, j, a)}
- {n, n, u2} ∪ φ = {m, n, u1}
(Phase 2)

USE[B2] = {i, j}
DEF[B2] = {φ}
IN[B2] = {i, j}
OUT[B2] = {i, j}

Phase 2 = {a, i, u2}

Phase 2

IN[B2] = {i, j} ∪
{a, j, u2 - {φ}}
= {i, j} ∪ {a, j, u2}
= {i, j, u2, a}

USE[B3] = {u2}
DEF[B3] = {a}
IN[B3] = {u2}
OUT[B3] = {φ}
Phase 2 = {

USE[B4] = {a, j}
DEF[B4] = {i}
IN[B4] = {a, j}
OUT[B4] = {φ}
Phase 2 = {i, j, u2, a}
IN[B4] = {a, j} ∪
(i, j, u2, a} - {i}
= {a, j} ∪ {a, j, u2}
= {a, j, u2}

→ Repeat until no changes

After 3 repetitions, no changes:

USE[B1] = {m, n, u1}

DEF[B1] = {i, j, a}

IN[B1] = {m, n, u1, u2}

OUT[B1] = {i, j, u2, a}

USE[B2] = {i, j}

DEF[B2] = { }

IN[B2] = {i, j, u2, a}

OUT[B2] = {a, j, u2}

USE[B3] =

DEF[B3] =

IN[B3] =

OUT[B3] =

USE[B4] =

DEF[B4] =

IN[B4]

OUT[B4]