Code Optimization

M.B.Chandak

Lecture notes on Language Processor

Approaches:

- Local Optimization
- Loop Optimization
- Peep Hole optimization

Basis:

- Optimized code should generate same results as original code.

Trade off:

- Time spent in optimization and outcome should be balanced

Local optimization:
1. Elimination of Common Sub-expression

- Common Sub-expressions
- Repeated in code and executed more than once
- **Example**

A = B+C*D

X = Y+C*D

- **Common expression can be found from TAC**

T1 = C*D

T2 = B+T1

T3 = C*D

T4 = Y + T3

Idea: To identify and replace

Implementation: Removal of Common SE

DAG: Directed Acyclic Graph

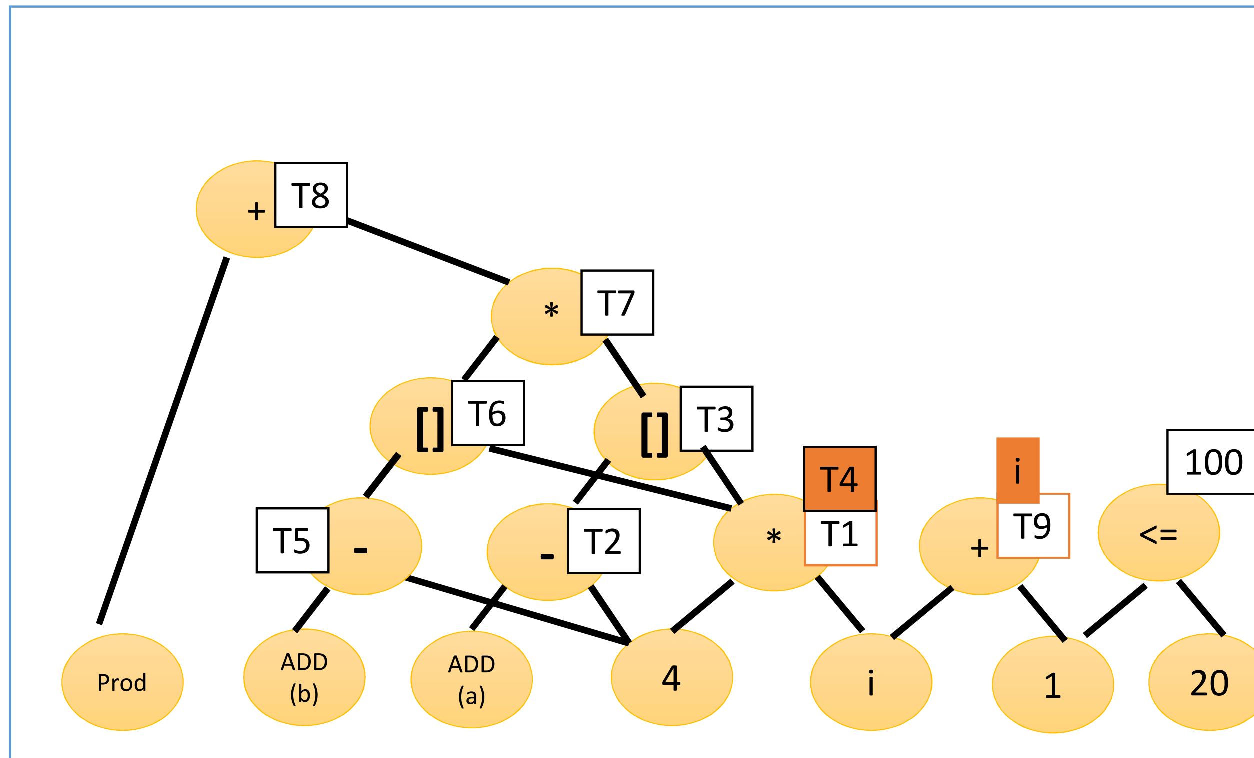
- Construction: From leaf node to root node
- Root node = Operator, Leaf node=Generally operands
- Left hand side rule is used to construct DAG

STEPS

- Given Program Segment – Construct TAC – Construct DAG – identify Common Sub Expressions – Remove Common SE – Rewrite TAC [optimized]

Examples: DAG

Label	Code
100	T1 = 4 * I
101	T2 = ADD(A) – 4
102	T3 = T2[T1]
103	T4 = 4 * I
104	T5 = ADD[B] – 4
105	T6 = T5[T4]
106	T7 = T3*T6
107	T8=PROD + T7
108	T9 = I + 1
109	I = T9
110	IF (I<=20) GOTO 100



Example

Label	Code
100	T1 = I * 4
102	T2 = ADD(A) - 4
103	T3 = T2[T1]
104	X = T3
105	T4 = J * 4
106	T5 = ADD(P) – 4
107	T6 = T5[T4]
108	T7 = I * 4
109	T8 = ADD(A) – 4
110	T9 = T8[T7]
111	Z = T9

Local Optimization – Part II

Topics:

Copy propagation, Dead Store elimination, Constant propagation, Variable propagation, Dead code elimination, Algebraic transformation

Code Optimization

1 COPY PROPAGATION

It is code transformation technique used to improve the intermediate code. In copy propagation, if statement **x=y** exists in code, then use "y" instead of "x" in statements following x=y in the TAC.

Consider following program segment:

```
int a[20]
int main()
{
   a[0] = 3;
   a[1] = 4;
}
```

The Three Address Code for the above program segment will be: Assume bpw = 4

```
10:     t1 = 0 * 4// value of "i" represents reference, which is 0 in statement a[0]=3
20:     t2 =  add(a) – C //C = bpw
30:     t2[t1] = 3
40:     t3 = 1 * 4 // value of "I" represents reference, which is 1 in statement a[1]=3
50:     t4 = add(a) – C // C =bpw
60:     t4[t3] = 4
```

In the above code, copy propagation is possible for statement 10 and 40. Value of t1=0 and value of t3=4. The three address code after replacing copy propagation, will be:

```
10:     t1 = 0
20:     t2 =  add(a) – C //C = bpw
30:     t2[0] = 3
40:     t3 = 4
50:     t4 = add(a) – C // C =bpw
60:     t4[4] = 4
```

1.1 Dead Store elimination

In the above code statement 10 and 40, represents dead store. The value of dead store memory locations are not used in the three address code and can be used for optimization.

The optimized code is represented as:

```
20:     t2 =  add(a) – C //C = bpw
30:     t2[0] = 3
50:     t4 = add(a) – C // C =bpw
60:     t4[4] = 4
```

2 CONSTANT PROPAGATION

In the above example, the use of constant “0” was propagated in place of t0 and constant “4” was propagated in place of t3. This kind of propagation is referred as constant propagation.

It is also called as “constant folding optimization”.

2.1 Variable Propagation: Consider following program segment

```
int demo(int a, int b, int c)
{
    int d,e,f;
    d = a;
    if(a>10)
      e=d+b;
   else
      e=d+c;
 f =d * e;
}
```

The Three Address code for above program segment will be:

```
10:   d = a
20    if (a>10) goto 40
30    goto 60
40    e = d+b
50    goto 70
60    e = d+c
70    f = d*e
```

The TAC will be modified as after performing of copy propagation:

```
10:   d = a
20    if (a>10) goto 40
30    goto 60
40    e = a+b  //variable propagation
50    goto 70
60    e = a+c //variable propagation
70    f = a*e  //variable propagation
```

The statement number 10 represents “dead store”, which can be also eliminated to perform further optimization.

```
20    if (a>10) goto 40
30    goto 60
40    e = a+b  //variable propagation
50    goto 70
60    e = a+c //variable propagation
70    f = a*e  //variable propagation
```

3 DEAD CODE ELIMINATION

The part of program segment which is never reached during execution is called as "dead code". This code can be eliminated to perform optimization.
Consider the following program segment

```
int debug;
int demo(int a, int b, int c)
{
   int x,y,z;
   debug=0
   if(debug=1)
      z=a+b+c;
  v=a+b+c;
}
```

The Three Address code for above program segment will be:

```
10:    debug=0
20:    if(debug=1) goto 30
30:    goto 70
40:    t1=a+b
50:    t2=t1+c
60:    z=t1
70:    t3=a+b
80:    t4=t3+c
90:    v=t4
```

After copy propagation:

```
10:    debug=0
20:    if(0=1) goto 30
30:    goto 70
40:    t1=a+b
50:    t2=t1+c
60:    z=t1
70:    t3=a+b
80:    t4=t3+c
90:    v=t4
```

Dead Code elimination: Since "debug=0", the "if condition" will not be true any time. The code of statement "20,30,40,50" will be never executed and hence can be eliminated.
The three address code after dead code elimination:

```
10:    debug=0
70:    t3=a+b
80:    t4=t3+c
90:    v=t4
```

Example 2: Dead Code Elimination

```
int demo(int a, int b, int c)
{
   int x,y,z;
   x = a+b
   y = x/c
   return(y)
   x=y+z;
   z=z+1
   return(z)
}
```

In the above code segment the "return(y)" statement will generate transfer of control to main program. This will generate "dead code" in the segment.

The Three Address Code for above segment will be:

```
10:   t1=a+b
20:   x=t1
30:   t2=t1/c
40:   y=t2
50:   return(y)
60:   goto 120
70:   t1=y+z
80:   x=t1
90:   t2=z+1
100:  z=t2
110:  return(z)
120:  exit
```

The optimized TAC after elimination of dead code:

```
10:   t1=a+b
20:   x=t1
30:   t2=t1/c
40:   y=t2
50:   return(y)
```

4 ALGEBRAIC TRANSFORMATIONS

The quality of intermediate code can be improved by taking advantage of Algebraic transformations. Some of the commonly used identities are:

Name of identity	Example	Example
Additive identity	x+0=x	y=x+0 means y=x
Multiplicative identity	x*1=x	y=x*1 means y=x
Multiplication with zero	x*0=0	y=x*0 means y=0

This transformation reduces computation time. For example in place of addition instruction, assignment instruction will be used, which requires less time during execution. Sometimes it is also possible to replace arithmetic instructions with copy instruction.

4.1 Strength Reduction Transformation

This transformation replaces the costly operations with less expensive operations.

Example:

y = x * 2 can be replaced with y = x+x

y = x * 32 can be replaced with y = x << 5

y = x / 8 can be replaced with y = x >> 3

5 ELIMINATION OF LOOP INVARIANT

An invariant instruction is represent a segment of code which generates same value during each iteration of the loop. If it is present in the loop, it will cause increase in execution time.

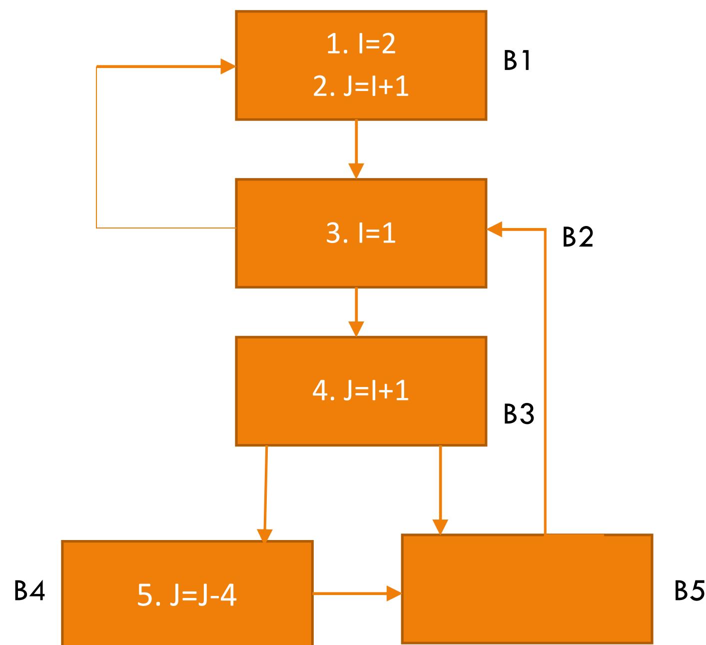
For computation of loop invariant, it is necessary to find out information about "Reaching Definitions", which can be generated with the help of "User-Defined Chaining" or IN-OUT analysis.

Given a program segment, to find out loop invariant, following steps are performed:

1. Program Segment To Three Address Code
2. Three Address Code to Program Flow Graph
3. Program Flow Graph to Loop detection process
4. Performing Reaching definitions to generate ud-chains
5. Find Loop invariant
6. Remove the loop invariant and store it in pre-header block, before the start of loop.

Example:

Given Program Flow Graph, perform IN-OUT analysis (1,2,3,4,5 represents definitions number)



Step1: Find Predecessor, GEN[B] and KILL[B] information

Block	Predecessor	GEN[B]	KILL[B]
B1	B2	{1,2}	{3,4,5}
B2	B1, B5	{3}	{1}
B3	B2	{4}	{2,5}
B4	B3	{5}	{2,4}
B5	B3, B4	{}	{}

Step 2: Perform IN-OUT Analysis

Initially: IN[B] = {} AND OUT[B]=GEN[B]

Block	IN[B]	OUT [B]
B1	{}	{1,2}
B2	{}	{3}
B3	{}	{4}
B4	{}	{5}
B5	{}	{}

Step 3: Perform further analysis using formulation and continue analysis till stable state is reached. (results of OUT[B] values of two iterations are similar)

IN[B] = U OUT[P] ➔ Union(OUT of all the predecessor blocks)
OUT[B] = {IN[B] – KILL[B]} U GEN[B]

Block	Predc	IN[B]	OUT [B]		**IN[B]**	**OUT[B]**	
B1	B2	{3}	{1,2}		**{2,3}**	**{1,2}**	
B2	B1, B5	{1,2}	{2,3}	**Phase 2**	**{1,2,3,4,5}**	**{2,3,4,5}**	
B3	B2	{2,3}	{3,4}		**{2,3,4,5}**	**{3,4}**	
B4	B3	{3,4}	{3,5}		**{3,4}**	**{3,5}**	
B5	B3, B4	{3,4,5}	{3,4,5}		**{3,4,5}**	**{3,4,5}**	

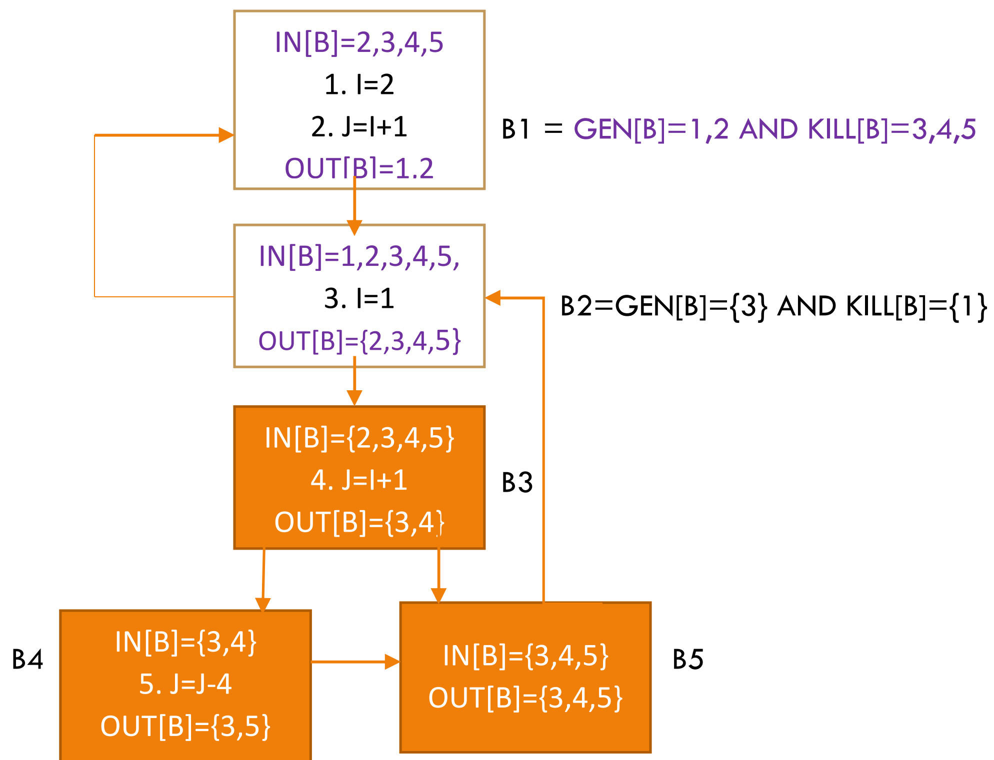
Block	Predc	IN[B]	OUT[B]		IN[B]	OUT[B]
B1	B2	{2,3}	{1,2}	Phase 3	{2,3,4,5}	{1,2}
B2	B1, B5	{1,2,3,4,5}	{2,3,4,5}		{1,2,3,4,5}	{2,3,4,5}
B3	B2	{2,3,4,5}	{3,4}		{2,3,4,5}	{3,4}
B4	B3	{3,4}	{3,5}		{3,4}	{3,5}
B5	B3, B4	{3,4,5}	{3,4,5}		{3,4,5}	{3,4,5}

Sample calculation phase 3:

Out(B1) = IN[B1] – KILL[B1] U GEN[B1] = {2,3,4,5} – {3,4,5} U {1,2} = {1,2}

After Phase 3, stable state is reached.

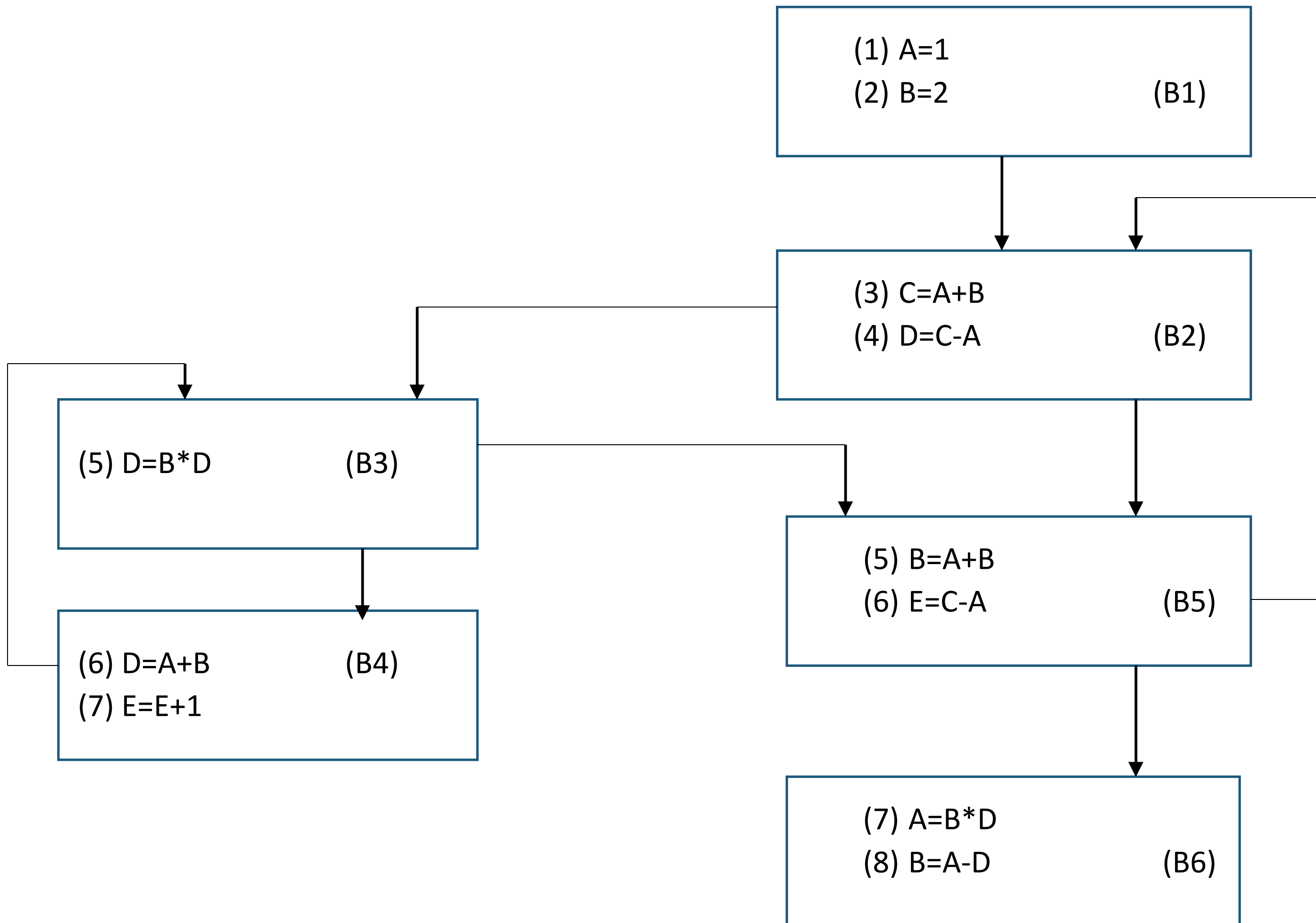
Reaching Definitions within loop or ud-chain of variables:



// ud-chain or reaching definitions //

//Students to complete remaining part from class notes //

Example:2 Perform IN-OUT Analysis and find ud-chain for any three "quads"



Block	Predecessor	GEN[B]	KILL[B]	IN[B]	OUT[B]
B1	{}	{1,2}	{8,10,11}	{}	{1,2}
B2	B1,B5	{3,4}	{5,6}	{}	{3,4}
B3	B2,B4	{5}	{4,6}	{}	{5}
B4	B3	{6,7}	{4,5,9}	{}	{6,7}
B5	B2,B3	{8,9}	{2,7,11}	{}	{8,9}
B6	B5	{10,11}	{1,2,8}	{}	{10,11}

Phase 2: Using IN-OUT in above table and formulation, compute IN-OUT till stable state is reached (OUT[B] RESULTS of two consecutive steps is same)

//** students to complete the remaining part **//