22/10/18

Code Optimization

It is a technique which tries to improve the code while eliminating unnecessary code lines and arranging the statements in such a sequence that speed up the program execution without wasting the recourses.

The adv. of code optimization are:-
1. Execute faster
2. Efficient memory usage
3. Yields better performance

Techniques for code optimization

1. Common sub expression
2. Constant folding
3. Code moment or code motion
4. Dead code elimination
5. Induction variable and reduction in strength.

Common sub-expression

$t_6 = 4*i$	$t_6 = 4*i$
$x = a[t_6]$	$x = a[t_6]$
$t_7 = 4*i$	$t_8 = 4*j$
$t_8 = 4*j$	$t_9 = a[t_6]$
$t_9 = a[t_8]$	$a[t_6] = t_9$
$a[t_7] = t_9$	$a[t_8] = x$
$t_{10} = 4*j$	goto B_2
$a[t_{10}] = x$	
goto B_2	
Before optimization	After optimization

Before optimization

Within the one block → local.
Within many blocks → global.

We are eliminating the variables having same values as previously assigned to some other variable. In short, we are eliminating the common expression.

Where our start expression will be the start of a block and where we encounter some jump or loop ending expression → there will be the end of the block.

The common sub-expression is an expression appearing repeatedly in the code which is computed previously. This technique replace redundant expressions each time it is encountered

Constant folding:

Eg: pie = 3.14
r = 5
Area = pie * r * r;

Eg: length = $\left(\frac{22}{7}\right) * d$

It refers to a technique of evaluating the expression whose operands are known to be constant at compile time itself.

Code movement or code motion technique

It is a technique of moving a block of code outside a loop, if it wouldn't have any difference if it is inside or outside the loop.

Before code movement

Eg: ① for (int i = 0; i<=10; i++)
{
x = y+z;
a[i] = 6 * i;
}

After code movement

②
x = y+z;
for (int i=0; i<=10; i++)
{
a[i] = 6 * i;
}

Dead code elimination

It include eliminating those code segments which are either never executed or unreachable.
If variable is live at a point in a program if its value can be used subsequently otherwise it is dead at that point.

Eg: If we are assigning a variable true and we are checking that variable is false then there are

Some execution related to the condition. So that condition will never be executed. So we have to remove these code (dead code).

```
int i = 1
=
=
if i = 0 is never get executed (we have to eliminate this)
then =
```

⑤ Induction variable and reduction in strength

Those variables whose value is increased or decreased by same constant value are known as induction variable.

As cost of multiplication is more that addition → we will replace mult with add if possible is know as reduction in strength.

The induction variables leads to strength reduction and eliminate computation. It is replacement of expression that are expensive with cheaper and simple one.

* Cause of redundancy (Read from book)
* Blocks concept

Loop optimization

① Loop (code) movement or code motion.
② Induction variable
③ Loop invariant
④ Loop unrolling
⑤ Loop fusion and loop jamming

③ Loop invariant

The computation inside the loop is avoided & thereby computational overhead on compiler is avoided.

```
for (int i=0 ; i<=10 ; i++)
    {
        p = a/b + i
    }
```

Before optimization

```
t = a/b
for (int i=0; i<=10; i++)
    {
        p = t + i
    }
```

After optimization

25/10/18

④ Loop unrolling:

→ Reduce the number of loops to half.

→ The number of jumps and test can be reduced while writing the code two times.

Before optimization

```
int i = 1;
while (i ≤ 100)
{
  a[i] = b[i]
  i++
}
```

After optimization

```
int i = 1
while (i ≤ 100)
{
  a[i] = b[i]
  i++
  a[i] = b[i]
  i++
}
```

All these optimization is done by compiler.

⑤ Loop fusion and loop jamming

Some adjacent loops can be fused into one loop to reduce loop overhead & improve run-time performance.

Basic Block and Flow Graph.

The basic block is a sequence of consecutive statements which are always executed in sequence without halt or possibility of branching.

if a < b then 1 else 0

```
① if (a<b) goto ⑤
② t1 = 0
③ t1 = 1
④
```

a = b + c + d

x1 = b + c

x2 = x1 + d

a = x2

Rules to convert 3-address code to Basic block

Rule 1 - Determine the leader:

(a) The first statement is a leader.

(b) Any target statement of conditional and unconditional goto is a leader.

(c) Any statement that immediately follows goto is a leader.

Rule 2 - The basic block is formed starting at the leader statement and ending just before the next leader statement appearing

① i = 1 (L) — B_1

② j = 1 (L) — B_2

③ $t_1 = 10 * i$ (L)

④ $t_2 = t_1 + j$

⑤ $t_3 = 8 * t_2$

⑥ $t_4 = t_3 - 88$

⑦ a[t_4] = 0.0

⑧ j = j + 1

⑨ if j <= 10 goto ② — B_3

⑩ i = i + 1 (L) — B_4

⑪ if i < 10 goto ②

⑫ i = 1 (L) — B_5

⑬ $t_5 = i - 1$ (L)

⑭ $t_6 = 8.8 * t_5$

⑮ a[t_6] = 1.0

⑯ i = i + 1

⑰ if i <= 10 goto ⑬ — B_6

There are total 6 basic blocks

The basic block doesn't have any jump statement among them. When the first statement is executed, all the inst^n in the same basic block will be executed in their sequence of appearance without losing the flow control of the program.

→ First we will have 3-address code
→ We will convert it to DAG, to apply rules to it
→ We will then convert it to 3 add code
→ Data flow analysis

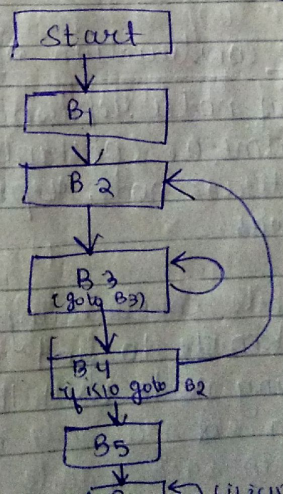
Flow graph :

A flow graph is a directed graph in which the flow control information is added to basic blocks.

Rule 1 → The basic blocks are the nodes to the flow graph.

Rule 2 → The block whose leader is the first statement is called initial block.

Rule 3 → There is a direct edge from block B_1 to block B_2, if B_2 immediately follow B_1 in the given sequence



In flow graph, we will replace goto ③ or any number to block numbers. as ③ lies in 3rd block so we will write goto ③

Optimization of basic blocks

① Representation of basic block using DAG

DAG is a useful data structure for implementing transformation on basic block. A basic block can be optimized by the construction of DAG.

→ A DAG can be constructed for a block and certain transformation such as common subexpression elimination and dead code elimination can be applied for performing the local optimization.

→ A DAG is constructed from 3-address statements.

Rules for construction of DAG

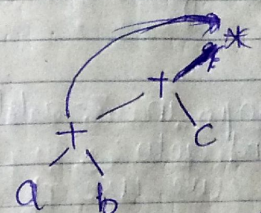
Rule 1 – Leaf nodes represent identifiers, name or constants and interior node represent operators.

Rule 2 → While constructing DAG, there is a checkmate to find if there is an existing node with the same children. A new node is created only when such a node doesn't exist. This action allow us to detect common sub-expression and eliminate the recomputation of the same.

ⓐ $(a+b) * (a+b+c)$

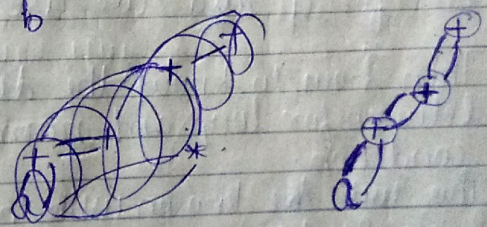
ⓑ $(((a+a) + (a+a)) + ((a+a) + (a+a))$

ⓐ



$t_1 = a+b$
$t_2 = t_1+c$
$t_3 = t_1 * t_2$

ⓑ



$t_1 = a+a$
$t_2 = t_1+t_1$
$t_3 = t_2+t_2$

29/10/18

Loop optimization

① Detect loops.
② Tow detecting loops.
→ CF in PFG.

(CF → control flow
PFG → program flow graph)

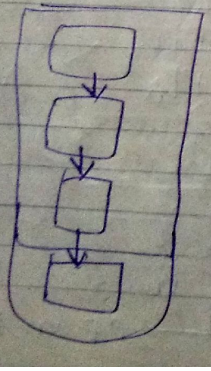
→ CF analysis of PFG.

③ For PFG
↳ we need BB.

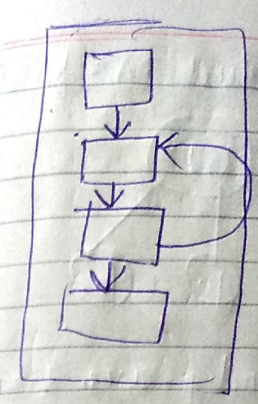
④ For BB → we find out leader.

→ We represent loops in 3-address code using if and goto.
And for if also we use if.

Then how to know the difference between for loop and normal "if" condition.
This will be known with the help of basic blocks.



No loop, then normal if statement



loop exists
(Eg. here there is no normal if condition, but there is loop).

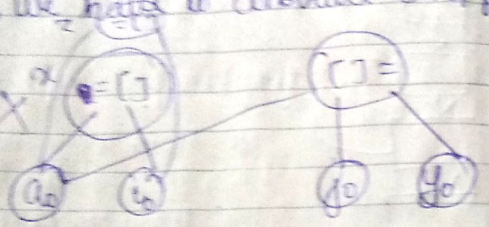
For detecting loops → we need program flow graph → for PFG we need basic blocks (BB) → for Basic blocks (BB) → we need leaders
(This is the chaining rule).

Eg:- $x = a[i]$ ①
$a[j] = y$ ②
$z = a[t]$

① If we are putting array's value into variable, we will be using =[]
It will have 2 children (a_0 and i_0) and value is being stored in x.

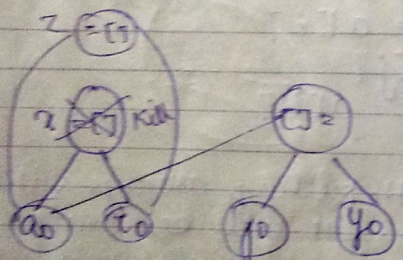
② If we are putting variable value into array, we will be using []=
and it will have 3 children
(a_0, j_0, y_0).

Now, we happy to construct DAG for this



If value of i and and j are same then we can kill x node.

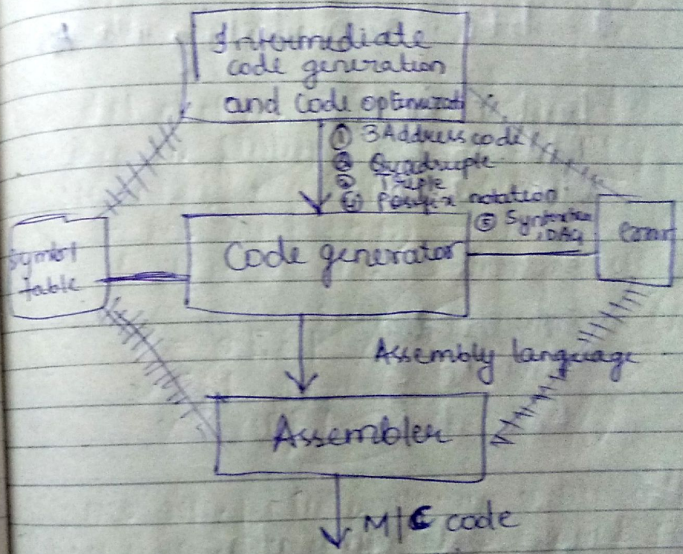
Now, in this case we are not using x further, as we are replacing it with z (z = a[i]), so here also we will kill x node.



(x is killed because, it is not being used further).

Ex: b = a+12
x = b[i]
b[j] = y

Code generation:



Issues in Code generation

① If input is different from the above ⑤, then code generation won't happen nicely.

② The O/P of the code generation, should be such that assembler can

convert it into machine code.

③ Instruction selection

④ Register allocation

a = b+c → converting into AL

Load c R_0
Add R_0 b
Store a R_0

d = a + e

Add R_0 e
Store d R_0

01/11/18

A Simple Code Generator

→ If we have 3-address code, then we have to convert it into assembly language code.

→ Register Descriptor :- It tracks which register, which variable is there

Track < Register, variablename >

Address Descriptor

Tracks which variable is in which location.

Tracks < variablename, location >.

We have a function here, GetReg().

Suppose we have 3 reg. $R_1 R_2 R_3$.

A = B + C

Load R_1, B
Load R_2 C
Add R_3 R_1 R_2

We are performing operation by loading variables in registers then cost is less, but if we are directly performing with memory location then cost is more. This all depends on machine architecture.

Get Reg ():

→ Find register R for computing A = B op C.

op ⇒ operation.

① If B is already in the register

and B is live after the block then return register R.

② If ① fails, return an empty register if available.

③ If ② fails, then use a heuristic to find an occupied register and empty that register.

Basis on which heuristic (factor) is based:-

① A register whose content are referenced farthest in future.

② The number of next user is smaller i.e, used for few time.

Here we will use command,

MOV C R_1.

Moving content of R_1 to location C then update the address descriptor.

④ Update register & address descriptor at every step.

Here we are assuming that no register is globally assigned. We are talking in terms of 1 block

Algorithm:-

① For each quadruple A = B op C do the following steps:-

① Find a location L to perform B op C. Usually a register can return by Get Reg() function.

② Where is the location of B.

③ B found using address descriptor. If B is both in memory & register use register for the operation and if B is in memory then load it in register.

③ Where is the location of C. Same as B we will find for C.

④ Generate → op R_3 (R_1 R_2)

op

dest. source.

⑤ Update descriptor (address & register both) for R_1, R_2 and R_3

Register Descriptor | Address Descriptor

As b is not used further so we will not store it

	R1	R2	R3	a	b	c	d	t	u	v
t = a - b	a	b		a	b	c	d			
				Initial.						
Load R1 a Load R2 b Sub R2 R1 R2										
u = a - c	a	t		a, R1	b	c	d	R2		
Load R3 c Sub R3 R1 R3										
v = t + u	a	t	u	a, R1	b	c	d	R2	R3	
Add R2 R2 R3										
a = d	a	v	u	a, R1	b	c	d	-	R3	R2
Load R1 d										
	a d	v	u	R1	b	c	R1 d	-	R3	R2

As value of a has been updated, so a is no more in memory location.

	R1	R2	R3	a	b	c	d	t	u	v
d = v + u	d	v	u	-	b	c	R1	-	R3	R2
Add R1 R2 R3										
Exit	-	-	-	-	b	c	d	-	u	v

Machine Independent Optimization

- loop jamming
- Constant folding, etc

Machine dependent Optimization

(After machine code generation)

- Peep hole optimization
- Register Allocation
- Addressing mode.

Peephole Optimization

(Small peephole window, we will slide it & find if we can reduce the code)

redundant.

1. Eliminate load and store statements.
2. Eliminate unreachable code.
3. Flow of control optimization.
4. Algebraic simplification and reduction in strength.

(1) Load R0 a
Store a R0
} Eliminate this type.

(2)
```
i = 1;
if (i == 0)
```
} Unreachable code. We have to eliminate.

(3) Jump over Jump statements.

→ Algebraic simplification

(4) `x = x * 1;`
`x = x / 1;` } Eliminate this type.

(5) `y = x^2` } Don't use this, instead of this we will use `y = x * x`.

→ Reduction in strength.

—— x ——