__/__/____

RUNTIME ENVIRONMENT & CODE OPTIMIZATION

RUNTIME ENVIRONMENT

outcome :

(V) Different storage allocation strategies.

[/code]

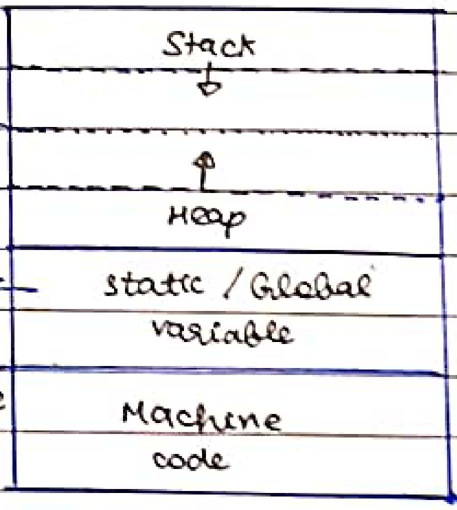
when we write a program code for a computer, the computer doesn't run the code. The operating system converts the program into a process.

Memory Layout :

It is basically the process structure.
What is javac ? It takes HLL code and converts it into machine executable code. only the machine code is not sufficient for the program to run. that is why the OS converts the machine executable code into a process.

(dynamic) (no fixed boundary)

store all variables that have the same lifetime as the process itself



stack goes downward, heap goes upward.

stack can go beyond boundary if heap is using less space and vice-versa.

Note : This process structure is the result of an evolution. The engineers nowadays are using this structure. Previously, each of these segments were tried out individually.

Storage Allocation strategies :

① **static** :

(i) Allocation is done at compile time.
(ii) Bindings do not change at run time.
(iii) one activation record per procedure.

program
f1()
f2()
f3()

static allocation
f1()
f2()
f3()

cons :

(i) Recursion is not supported.
(ii) size of data objects must be known at compile time.
(iii) Data structures cannot be created dynamically.

Pro :

The element which is provided with the static allocation gets the lifetime as same as the process itself.

Note : Global arrays are static by default

② **stack** :

whenever a new activation begins, the activation record is pushed onto the stack and whenever activation ends, the activation record is popped off.

stack
f3()
f2()
f1()

```
f1() {
    f2() {
        f3() { }
    }
}
```

con : Local variables cannot be retrieved once activation ends.

Pro : Recursion is supported.

③ **Heap** :

Allocation and deallocation can be done in any order.

con : Heap management is an overhead.

Note : In C programming First Fit is used.

Storage Allocation Strategies - summary

why engineers decided to use a compaction of all strategies instead of using them individually?

① permanent lifetime in case of static allocation.
② Nested lifetime in case of stack allocation
③ Arbitrary lifetime in case of heap allocation.

The compacted process structure is known as the RUNTIME ENVIRONMENT.

stack + heap + static

Summary :

✓ Different storage allocation strategies.

_/_/_

CODE OPTIMIZATION

Outcome :

1. Understanding the purpose of code optimization.
2. Objective of code optimization.
3. Different code optimization techniques.

Objectives of code optimization

1. The optimization must be correct and must not change the meaning of the program.
2. The compilation time must be kept reasonable.
3. The optimization process should not delay the overall compiling process.
4. Optimization should increase the speed and performance of the program

Purpose of code optimization

1. It is used to reduce the consumed memory space.
2. It is used to increase the compilation speed.
3. An optimized code facilitates re-usability.

Types of code optimization

1. Machine Independent :
   Improves the intermediate code.
2. Machine Dependent :
   # It involves CPU registers and may have absolute memory references rather than relative references.
   # It is performed after the target code has been generated.

_/_/__

Machine Independent optimizations :

① Loop optimizations

```
for (int i=0; i<10; i++)
        a = i * 2;
for (int j=0; j<10; j++)
        b = j + 3;
```

$\Rightarrow$

```
for (int i=0; i<10; i++)
{
        a = i * 2;
        b = i + 3;
}
```

② constant folding

In an expression, if the operands are known to be constants, all the operands can be replaced by the constant (evaluated value of the expression).

③ constant propagation

If an operand has a constant value, it will be replaced in every place where the operand is used.

④ operator strength Reduction

We try to settle down with lesser expensive operators instead of those operators which are mentioned in the HLL code.

⑤ Dead code Elimination

If the result of an instruction is never used, it is called as dead code. It can be eliminated.

⑥ common subexpression elimination

2 expressions are common if they produce the same result. In such cases, the expression is

evaluated once, and the result is used as a reference whenever the expression is next encountered.

(4) Algebraic simplification
[similar to constant folding & constant propagation]. Basically, particular operator-operand combinations are used to simplify algebraic expressions.

Machine Dependent optimizations

(1) Register Allocation
[completely dependent on machines - complete knowledge about architecture is necessary]

(2) Instruction Scheduling
If we schedule instructions based on the machine architecture, we can significantly improve performance.

(3) peephole optimizations
The whole intermediate code will not be considered. only a few lines of it will be evaluated through a peephole.
(a) Redundant LOAD and STORE : eliminated
(b) Flow control optimizations
(c) use of machine Idioms

Summary :
✓ purpose ✓ objective ✓ techniques

_/_/_

LOOP OPTIMIZATION – PART 1

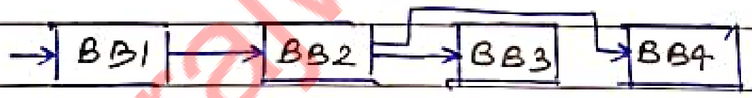
Outcome :

- ✓ Requisites for loop optimization.
- ✓ Understanding Basic Blocks, Program Flow Graph and control Flow Analysis.

Loop optimization :

* The loops must be detected
* Loops are detected through control Flow Analysis (CFA) using program flow graphs (PFG).
* In order to determine PFG, we first need to detect the basic blocks.

Basic Block : It is a sequence of 3 address statements where control enters at the beginning and leaves only from the end without any jumps / halts.



(control flow analysis)

Summary :

✓ Requisites ✓ BB, PFG and CFA

LOOP OPIMIZATION – PART 2

Outcome :

- ✓ How to determine the basic blocks.
- ✓ Illustration of control Flow Analysis.

How to find the Basic Blocks ?

Find the Leader. (starting statements of every basic block)

__/__/__

Identifying the Leader :

1. First statement is a leader.
2. Statement that is the target of a conditional or unconditional GOTO is a leader.
3. Statement that immediately follows a conditional or unconditional GOTO is a leader.

Illustration - Producing PFG

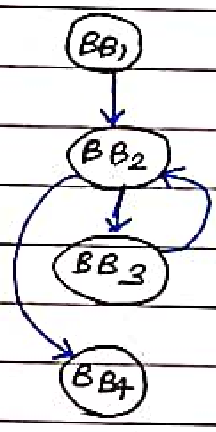
HLL code :

```
fact(a) {
    int f = 1;
    for(i=2; i<=a; i++)
        f = f * i;
}
```

TAC :

```
① f = 1              BB3
② i = 2
③ if(i<=a) GOTO 9    BB2
④ t1 = f * i
⑤ f = t1             BB1
⑥ t2 = i+1
⑦ i = t2
⑧ GOTO 3
⑨ GOTO <calling program>   BB4
```

PFG :



Summary :

- ✓ BB
- ✓ CFA illustration

LOOP OPTIMIZATION TECHNIQUES

Outcome :

- ✓ Different types of loop optimization techniques.

Loop optimization Technique - code Motion

* The number of statements within loop is reduced.
* Also known as Frequency reduction.

```
while (i < 1000)                          t = sin(x) / cos(x);
{                                   →     while (i < 1000)
    a = (sin(x) / cos(x)) * i;            {
    i++;                                      a = t * i;
}                                             i++;
                                          }
```

Note: Moving the expression with computation outside is also known as Loop Invariant Method.

Loop optimization Technique - Loop unrolling

* If possible, eliminate the entire loop.

```
for(int i=0; i<5; i++)          =>    printf("Hello");
    printf("Hello");                  printf("Hello"); (5 times)
```

Loop optimization Technique - Loop Jamming

* combine the loop bodies.
or Also known as loop fusion.

```
for(i=0; i<10; i++)
    a= i*2;                    =>   for(i=0; i<10; i++)
for (j=0; j<10; j++)                {
    b= j+3                              a= i*2;
                                        b= i+3;
                                    }
```

Note: The opposite transformation is called Loop Fission or Loop Distribution.

Loop optimization Technique - Loop Unswitching

* It lifts conditions out of loops, creating 2 loops.

```
for( i=0; i<100; ++i)          if (c)
{                              {
    if (c) f();                    for (i=0; i<100; ++i)
    else g();                          f();
}                              }
                               else
                               {
                                   for (i=0; i<100; i++)
                                       g();
                               }
```

Note: Also known as loop splitting.
Now condition is checked only once, instead of being checked 100 times.

__/__/____

Loop optimization Technique - Loop Peeling

Here, problematic iteration is resolved separately before entering the loop.

⇓

```
for (i=0; i<10; i++)            a[0] = ......
{                          ⇒    for (i=1; i<10; i++)
    if (i==0)  a[i] = ....      {   b[i] = ...
    else       b[i] = ....      }
}
```

Summary :

✓ Different types of loop optimization technique.

MACHINE INDEPENDENT OPTIMIZATION TECHNIQUES

Outcome :

✓ Different machine independent optimization technique

constant Folding

* Evaluation of expressions at compile time
* Applicable to the operands which are known to be constant.

$a = 10 * 5 + 6 - b; \Rightarrow a = 56 - b;$

constant propagation

If a variable is assigned a constant value, then subsequent uses of that variable can be replaced by the constant as long as no intervening assignment has changed the value of the variable

① a = 13.7 ⇒ a/4.5 as
b = a/4.5   13.7/4.5

② j = 1
if (j) GOTO L ⇒ GOTO L

Operator strength Reduction:

* It replaces an operator by a less expensive one.

$b = a * 2 \Rightarrow \underbrace{b = a << 1}_{\text{less expensive}}$

Dead code Elimination

If an instruction's result is never used, the instruction is considered dead and can be removed from the instruction stream.

$\cancel{t_1 = t_2 * t_3}$ But if t_1 holds the result of a function call, we cannot eliminate the instruction

Common Subexpression Elimination

Two operations are common if they produce the same result. In such a case, it is likely more efficient to compute the result once and reference it the second time rather than re-evaluate it.

$A = B + C$
$D = 2 + B + 3 + C$ $\Rightarrow$ $A = B + C$
$D = 2 + 3 + A$

Algebraic simplification

* Simplifications use algebraic properties or particular operator-operand combinations to simplify expressions.
* These optimizations can remove useless instructions entirely via algebraic identities.

$\cancel{A = A + 0}$
$\cancel{B = B * 1}$

Summary:

1. Different Machine independent optimization Techniques.

__/__/____

MACHINE DEPENDENT OPTIMIZATION TECHNIQUES

Outcome :

1. Machine Dependent Optimization Techniques

Register Allocation

* The most effective optimization for all architectures.
* solely depends on the number of available registers.
* Types : Local Allocation (for local variables)
  Global Allocation (for global variables)

Instruction scheduling

* Using this, the pipelining capability of the architecture can be used effectively.
* Instructions can be placed in the delay slots (like NOOP)

peep-hole optimization - Redundant LOAD & STORE

$x = y + z$

MOV y, Ro
ADD z, Ro
MOV Ro, x

a = b + c → MOV b, Ro
d = a + e   ADD c, Ro
            MOV Ro, a   (redundant)
            MOV a, Ro
            ADD e, Ro
            MOV Ro, d

⇒ MOV b, Ro
  ADD C, Ro
  ADD e, Ro
  MOV Ro, d

Peephole optimization - Flow control

1. Avoid Jumps on Jumps

L1 : GOTO L2
L2 : GOTO L3
L3 : GOTO L4
L4 :

⇒ L1 : GOTO L4
  L4 :

__/__/__

② Eliminate Dead code

define x 0

```
if (x)
{ ....
}
```

→ dead code } will never be executed

Peephole optimization - use of Machine Idioms

i = i+1 → MOV i, R_0

ADD i, R_0 ⟹ INC i

MOV R_0, i

summary :

✓ Machine Dependent optimization Techniques.

LIVENESS ANALYSIS

outcome :

✓ Understanding Liveness Analysis

✓ solved problem on Liveness Analysis

Liveness Analysis

what is Liveness ?

A variable is live if its value will be used before it gets overwritten.

why is it important ?

① Register allocation : It helps in deciding which variables to keep in registers and which to store in memory to optimize performance.

The variables which are used frequently are kept in registers, while the variables which are not so frequently used are permanently stored in the memory.

__/__/__

② Dead code elimination : It can be used to identify and remove code that computes values that are never used.

③ code scheduling : It is used to reorder instructions to minimize stalls and improve instruction-level parallelism.

Liveness Analysis - Algorithm

Input : Program Flow Graph
Output : Liveness information :

IN[B] : set of variables that are live at the beginning of the block.

OUT[B] : set of variables that are live after the block.

DEF/KILL[B] : set of variables that are defined/killed in the block. [killed = modified]
[basically variables present in the LHS of the equations].

USE/GEN[B] : set of variables that are used/generated in the block. [basically variables present in RHS but not LHS]

Algorithm :

① Initialization : IN and OUT sets are initially empty. DEF/KILL and USE/GEN will be initialized on observing the basic blocks.

② Worklist Initialization : Create a worklist containing all the basic blocks of the CFG.

③ **Iterative Dataflow Analysis :** while the worklist is not empty, perform the following steps for each basic block :

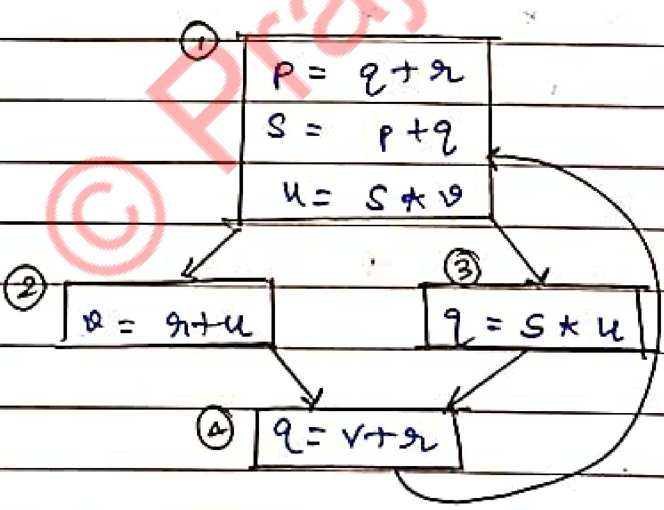
* calculate $IN[B] = USE[B] \cup (OUT[B] - DEF[B])$
* calculate $OUT[B] = \cup IN[S]$

④ **Final Liveness Information :** After the analysis has converged, the 'IN' sets represent the live variables at the entry point for each block

Q : A variable x is said to be live at a statement S_i in a program if the following 3 conditions hold simultaneously :

(GATE 2015)

① There exists a statement S_j that uses x.
② There is a path from S_i to S_j in the flow graph corresponding to the program.
③ The path has no intervening assignment to x including at S_i and S_j.



The variables which are live both at the statement in basic block 2 and at the statement in basic block 3 of the above control flow graph are

(a) p, s, u (b) r, s, u
(c) r, u (d) q, v

Basic Block	USE	DEF	IN	OUT
①	q, r, v	p, s, u	q, r, v	r, u, s
②	r, u	v	r, u	v, r
③	s, u	q	s, u	v, r
④	v, r	q	v, r	q, r, v

← Iteration ①

__/__/____

IN	OUT	IN	OUT
r, q, v	v, r, s, u	r, v, q	v, r, s, u
r, u	r, v	r, u	r, v
v, r, s, u	r, v	v, r, s, u	r, v
r, v	r, q, v	r, v	r, v, q

Iteration ② Iteration ③

⇓

clearly, iteration has converged.

$$\left.\begin{aligned} IN(2) &= \{r, u\} \\ IN(3) &= \{v, r, s, u\} \end{aligned}\right] \rightarrow$$ (provide liveness information)

$IN(2) \cap IN(3) = \boxed{\{r, u\} \Rightarrow (c)}$

Semantic Approach

Basic Blocks	p	q	r	s	u	v
2	×	×	✓	×	✓	×
3	×	×	✓	✓	✓	✓

summary :

✓ understanding Liveness Analysis.

✓ solved problem on Liveness Analysis.