I. CODE OPTIMIZATION

- Optimization is the process of transforming a piece of code to make more efficient (either in terms of time or space) without changing its output or side-effects.
- The only difference visible to the code's user should be that it runs faster and/or consumes less memory.
- Optimizations are classified into two categories. They are
    1. **Machine independent** optimizations
    2. **Machine dependant** optimizations

Machine independent optimizations - Machine independent optimizations are program transformations that improve the target code without taking into consideration any properties of the target machine.

Machine dependant optimizations - Machine dependant optimizations are based on register allocation and utilization of special machine-instruction sequences.

The criteria for code improvement transformations

1. Simply stated, the best program transformations are those that yield the most benefit for the least effort.
2. The transformation must preserve the meaning of programs. That is, the optimization must not change the output produced by a program for a given input, or cause an error that was not present in the original source program.
3. A transformation must, on the average, speed up programs by a measurable amount.
4. The transformation must be worth the effort. It does not make sense for a compiler writer to expend the intellectual effort, to implement a code improving transformation and to have the compiler expend the additional time on compiling source programs if this effort is not repaid when the target programs are executed.

1. 1 The Principle Sources of Optimization

- A transformation of a program is called local if it can be performed by looking only at the statements in a basic block; otherwise, it is called global.
- Many transformations can be performed at both the local and global levels. Local transformations are usually performed first.

Function-Preserving Transformations / Semantics Preserving Transformations

- There are a number of ways in which a compiler can improve a program without changing the function it computes. The transformations are
    1. **Common sub expression elimination,**
    2. **Copy propagation,**
    3. **Dead-code elimination**
    4. **Constant folding**

1. Common Sub expressions elimination

- *An occurrence of an expression E is called a common sub-expression, if E was previously computed and the values of variables in E have not changed since the previous computation*.
- We can avoid re-computing the expression if we can use the previously computed value.

Example

t1: = 4*i **t2: = a [t1]** **t3: = 4*j** **t4: = 4*i** **t5: = n** **t6: = b [t4] + t5**	The common sub expression t4: =4*i is eliminated as its computation is already in t1 and the value of "i" is not been changed from definition to use.	**t1: = 4*i** **t2: = a [t1]** **t3: = 4*j** **t5: = n** **t6: = b [t1] + t5**

2. Copy Propagation

- Assignments of the form f = g called copy statements, or copies for short.
- *The idea behind the copy-propagation transformation is to use g for f, whenever possible after the copy statement f: = g.*
- Copy propagation means use of one variable instead of another. This may not appear to be an improvement, but as we shall see it gives us an opportunity to eliminate some element.

Example

x = pi;
……
A = x*r*r;
The optimization using copy propagation can be done as follows:
A = pi*r*r;
Here the variable x is eliminated.

3. Dead-Code Eliminations - A variable is live at a point in a program if its value can be used subsequently; otherwise, it is dead at that point.

- *A related idea is dead or useless code, statements that compute values that never get used*.
- While the programmer is unlikely to introduce any dead code intentionally, it may appear as the result of previous transformations. An optimization can be done by eliminating dead code.

Example

```
i = 0;
if (i=1)
{
a=b+5;
}
```

Here, 'if' statement is dead code because this condition will never get satisfied. We can eliminate both the test and printing from the object code.

4. Constant folding

- More generally, *deducing at compile time that the value of an expression is a constant and using the constant instead is known as constant folding.*
- One advantage of copy propagation is that it often turns the copy statement into dead code.

Example

a = 3.14 / 2;
can be replaced by
a = 1.57; thereby eliminating a division operation.

1. 2 Optimization of Basic Blocks

- There are two types of basic block optimizations. They are,
  1. **Structure-Preserving Transformations**
  2. **Algebraic Transformations**

Structure-Preserving Transformations

- The primary Structure-Preserving Transformation on basic blocks are
  1. **Common sub-expression elimination**
  2. **Dead code elimination**
  3. **Renaming of temporary variables**
  4. **Interchange of two independent adjacent statements.**

1. Common sub-expression elimination

- Common sub expressions need not be computed over and over again. Instead they can be computed once and kept in store from where it's referenced when encountered again.

Example

a : = b + c **b : = a - d** **c : = b + c** **d : = a - d**	The 2nd and 4th statements compute the same expression b+c and a-d. Hence the Basic block can be transformed to	**a : = b + c** **b : = a - d** **c : = a** **d : = b**

2. Dead code elimination

- It's possible that a large amount of dead (useless) code may exist in the program.
- This might be especially caused when introducing variables and procedures as part of construction or error-correction of a program – once declared and defined, one forgets to remove them in case they serve no purpose.
- Eliminating these will definitely optimize the code.

3. Renaming of temporary variables

- *A statement t =b + c where t is a temporary name can be changed to u = b + c where u is another temporary name, and change all uses of t to u.*
- In this we can transform a basic block to its equivalent block called normal-form block.

4. Interchange of two independent adjacent statements

Two statements
t1 := b + c

t2 := x + y

can be interchanged or reordered in its computation in the basic block when value of t1 does not affect the value of t2.

5. Algebraic Transformations

- Algebraic identities represent another important class of optimizations on basic blocks. This includes simplifying expressions or replacing expensive operation by cheaper ones.
- Another class of related optimizations is constant folding. Here we evaluate constant expressions at compile time and replace the constant expressions by their values. Thus the expression 2*3.14 would be replaced by 6.28.
- Associative laws may also be applied to expose common sub expressions.

  Example
  - **x:=x+0** or **x = x *1** can be removed.
  - **x:=y**2** can be replaced by a cheaper statement **x:=y*y**
  - The compiler writer should examine the language carefully to determine what rearrangement of computations are permitted, since computer arithmetic does not always obey the algebraic identities of mathematics. Thus, a compiler may evaluate x*y - x*z as x*(y-z) but it may not evaluate a+(b-c) as (a+b)-c.

1.3 Loop Optimizations

- The loops where programs tend to spend the bulk of their execution time. The running time of a program may be improved if we decrease the number of instructions in an inner loop, even if we increase the amount of code outside that loop.
- Three techniques are important for loop optimization
  1. **Code motion**, which moves code outside a loop;
  2. **Induction-variable elimination**, which we apply to replace variables from inner loop.
  3. **Reduction in strength**, which replaces and expensive operation by a cheaper one, such as a multiplication by an addition.

1. Code Motion

- An important modification that decreases the amount of code in a loop is code motion.
- This transformation takes an expression that yields the same result independent of the number of times a loop is executed (a loop-invariant computation) and places the expression before the loop.
- Note that the notion “before the loop” assumes the existence of an entry for the loop.
- For example, evaluation of limit-2 is a loop-invariant computation in the following while statement:

  while (i <= limit-2) /* statement does not change limit*/

  Code motion will result in the equivalent of

  t= limit-2;

  while (i<=t) /* statement does not change limit or t */

2. Induction Variables

- Loops are usually processed inside out.

- When there are two or more induction variables in a loop, it may be possible to get rid of all but one, by the process of induction-variable elimination.

Example:

As the relationship t4:=4*j surely holds after such an assignment to t4 in Fig. and t4 is not changed elsewhere in the inner loop around B3, it follows that just after the statement j:=j-1 the relationship t4:= 4*j-4 must hold. We may therefore replace the assignment t4:= 4*j by t4:= t4-4. The only problem is that t4 does not have a value when we enter block B3 for the first time. Since we must maintain the relationship t4=4*j on entry to the block B3, we place an initializations of t4 at the end of the block where j itself is initialized, shown by the dashed addition to block B1 in second Fig. The replacement of a multiplication by a subtraction will speed up the object code if multiplication takes more time than addition or subtraction, as is the case on many machines.

3. Reduction In Strength

- Reduction in strength replaces expensive operations by equivalent cheaper ones on the target machine.
- Certain machine instructions are considerably cheaper than others and can often be used as special cases of more expensive operators.

For example,

1. x^2 is invariably cheaper to implement t as x*x than as a call to an exponentiation routine.
2. Fixed-point multiplication or division by a power of two is cheaper to implement as a shift. Floating-point division by a constant can be implemented as multiplication by a constant, which may be cheaper.

4. Loop Unrolling

- The loop exit checks cost more CPU time. Loop unrolling tries to get rid of the checks completely or to reduce the number of checks.
- If you know a loop is only performed a certain number of times, or if you know the number of times it will be repeated is a multiple of a constant you can unroll this loop.

Example:

```
// old loop
for(int i=0; i<3; i++) {
        color_map[n+i] = i;
}

// unrolled version
int i = 0;
colormap[n+i] = i;
i++;
colormap[n+i] = i;
i++;
colormap[n+i] = i;
```

II. CODE GENERATION

- The final phase in compiler model is the code generator.
- It takes as input an intermediate representation of the source program and produces as output an equivalent target program.

2.1 Issues in the Design of a Code Generator

- The design of the code generator is dependent on the specifics of the intermediate representation, the target language and the run-time systems such as,
  1. **Input to the code generator**
  2. **Target program**
  3. **Memory management**
  4. **Instruction selection**
  5. **Register allocation**
  6. **Evaluation order**

1. Input to the code generator

- The input to the code generation consists of the intermediate representation of the source program produced by front end, together with information in the symbol table to determine run-time addresses of the data objects denoted by the names in the intermediate representation.
- The many choices for the Intermediate representations include,
  1. **Linear representation** such as postfix notation
  2. **Three address representation** such as quadruples, triples or indirect triples
  3. **Virtual machine representation** such as stack machine code
  4. **Graphical representations** such as syntax trees and DAG.
- Prior to code generation, the front end must be scanned, parsed and translated into intermediate representation along with necessary type checking. Therefore, input to code generation is assumed to be error-free.

2. The Target program

- The Instruction Set Architecture (ISA) of the target machine has a significant impact on the difficulty of designing a good code generator.
- The most common target-machine architectures are RISC, CISC and Stack based.
- The output of the code generator is the target program. The output may be
  1. **Absolute machine language** - It can be placed in a fixed memory location and can be executed immediately.
  2. **Relocatable machine language** - It allows subprograms to be compiled separately.
  3. **Assembly language** - Code generation is made easier.

3. Memory management

- Names in the source program are mapped to addresses of data objects in run-time memory by the front end and code generator.

- It makes use of symbol table, that is, a name in a three-address statement refers to a symbol-table entry for the name.
- Labels in three-address statements have to be converted to addresses of instructions.

  For example,

  j: goto i where i is a label

  generates jump instruction as follows,

  if i < j, a backward jump instruction with target address equal to location of code for quadruple i is generated.

  if i > j, the jump is forward. We must store on a list for quadruple i the location of the first machine instruction generated for quadruple j. When i is processed, the machine locations for all instructions that forward jumps to I are filled.

4. Instruction Selection

- The instructions of target machine should be complete and uniform.
- Instruction speeds and machine idioms are important factors when efficiency of target program is considered.
- The quality of the generated code is determined by its speed and size.

5. Register allocation

- Instructions involving register operands are shorter and faster than those involving operands in memory.
- The use of registers is subdivided into two sub problems,
  1. **Register allocation** – the set of variables that will reside in registers at a point in the program is selected.
  2. **Register assignment** – the specific register that a variable will reside in is picked.
- Certain machine requires even-odd register pairs for some operands and results.

6. Evaluation order

- The order in which the computations are performed can affect the efficiency of the target code.
- Some computation orders require fewer registers to hold intermediate results than others.

2.2 Basic Blocks and Flow Graphs

Basic Blocks

- A basic block is a sequence of consecutive statements in which flow of control enters at the beginning and leaves at the end without any halt or possibility of branching except at the end.

Basic Block Construction Algorithm

Algorithm : Partition into basic blocks
Input : A sequence of three-address statements
Output : A list of basic blocks with each three-address statement in exactly one block
Steps :

1. First determine the set of leaders, the first statements of basic blocks.

The rules for determining the Leaders are,

a. The **first statement** is a leader.
b. Any statement that is the **target of a conditional or unconditional goto** is a leader.
c. Any statement that **immediately follows a conditional or unconditional goto** statement is a leader.

2. For each leader, its basic block consists of the leader and all statements up to but not including the next leader or the end of the program.

Flow Graphs

- Flow graph is a directed graph containing the flow-of-control information for the set of basic blocks making up a program.

Properties of Flow-graph

- The nodes of the flow graph are basic blocks and the first block is designated as a Initial block
- There is an directed edge from the basic block B1 to B2, iff B1 is a predecessor of B2.

Rules for determining the predecessor node

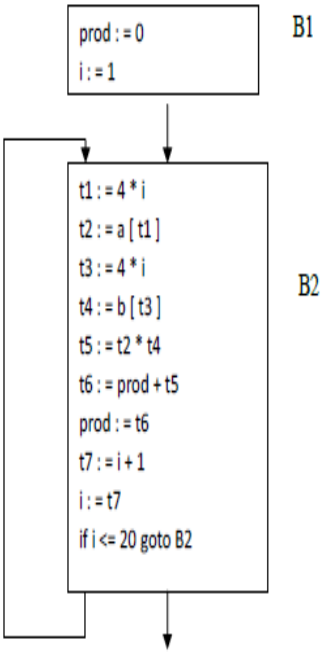
The block B1 is said to be predecessor to Block B2,

1. If B2 can immediately follows B1 in execution sequence or
2. If B2 immediately follows B1 in the execution sequence and B1 does not end with an unconditional goto statement.

Example:

- For the following segment of the Code,

```
begin
        prod :=0;
        i:=1;
        do begin
                prod :=prod+ a[i] * b[i];
                i :=i+1;
        while (i <= 20)
end
```



- The Three address code Instructions are,

(1) prod := 0
(2) i := 1
(3) t1 := 4* i
(4) t2 := a[t1]
(5) t3 := 4* i
(6) t4 := b[t3]
(7) t5 := t2*t4

(8) t6 := prod+t5
(9) prod := t6
(10) t7 := i+1
(11) i := t7
(12) if i<=20 goto (3)

- Leaders are statement No: (1) and (3). Hence the Basic block B1: Statement (1) to (2) & Basic block B2: Statement (3) to (12).

2.3 Next-Use Information

- If the name in a register is no longer needed, then we remove the name from the register and the register can be used to store some other names.
- Next use information is used in code generation and register allocation.

The use of a name in a three-address statement is defined as

Let the TAC statement "i" assigns a value to the element x,

- *If TAC statement "j" has x as an operand and control can flow from statement "i" to "j" along the path there is no intervening assignments to x, then the statement "j" uses the value of x computed at statement "i".*
- The Next use Computation algorithm assumes that on exit from the basic block,
  1. All temporaries are considered non-live
  2. All variables defined in the source program (and non-temps) are live
  3. Each procedure/function call is assumed to start a basic block

Algorithm to determine Liveliness and Next-use information

Input : Basic block B of three-address statements

Output : At each statement i: x= y op z, we attach to i the liveliness and next-uses of x, y and z.

Method : We start at the last statement of B and scan backwards.

1. Attach to statement i the information currently found in the symbol table regarding the next-use and liveliness of x, y and z.
2. In the symbol table, set x to "not live" and "no next use".
3. In the symbol table, set y and z to "live", and next-uses of y and z to i.

Example: Consider the following TAC sequence in a Basic Block

[NNU – No Next Use; NU – Next Use, NLive – Not Live]

(1)	prod := 0	**{prod- [Live, NNU] }**
(2)	i := 1	**{ i – [Live, NU – (3)] }**
(3)	t1 := 4* i	**{ t1 – [NLive, NU – (4), i-[Live, NNU] }**
(4)	t2 := a[t1]	**{ t2 – [NLive, NNU], t1 – [NLive, NNU], a – [Live, NNU]}**

2.4 The DAG Representation for Basic Blocks

- A DAG for a basic block is a directed acyclic graph with the following labels on nodes
  1. Leaves are labeled by unique identifiers, either variable names or constants.
  2. Interior nodes are labeled by an operator symbol.
  3. Nodes are also optionally given a sequence of identifiers for labels to store the computed values.
- DAGs are useful data structures for implementing transformations on basic blocks.
- It gives a picture of how the value computed by a statement is used in subsequent statements.
- It provides a good way of determining common sub - expressions.

Algorithm of Construction of DAG

Input : A basic block

Output : A DAG for the basic block containing the following information:

1. A label for each node.
   For leaves, the label is an identifier.
   For interior nodes, an operator symbol.
2. For each node, a list of attached identifiers to hold the computed values.
   Case (i) x : = y OP z
   Case (ii) x : = OP y
   Case (iii) x : = y

Method:

Step 1: If y is undefined then create node(y). If z is undefined, create node (z) for case(i).

Step 2:

For the Case (i):

Create a node (OP) whose left child is node(y) and right child is node (z). (Checking for common sub expression. Let n be this node)

For Case (ii):

Determine whether there is node (OP) with one child node(y). If not create such a node.

1. $t_1 := 4* i$
2. $t_2 := a[t_1]$
3. $t_3 := 4* i$
4. $t_4 := b[t_3]$
5. $t_5 := t_2*t_4$
6. $t_6 := prod+t_5$
7. $prod := t_6$
8. $t_7 := i+1$
9. $i := t_7$
10. if i<=20 goto (1)

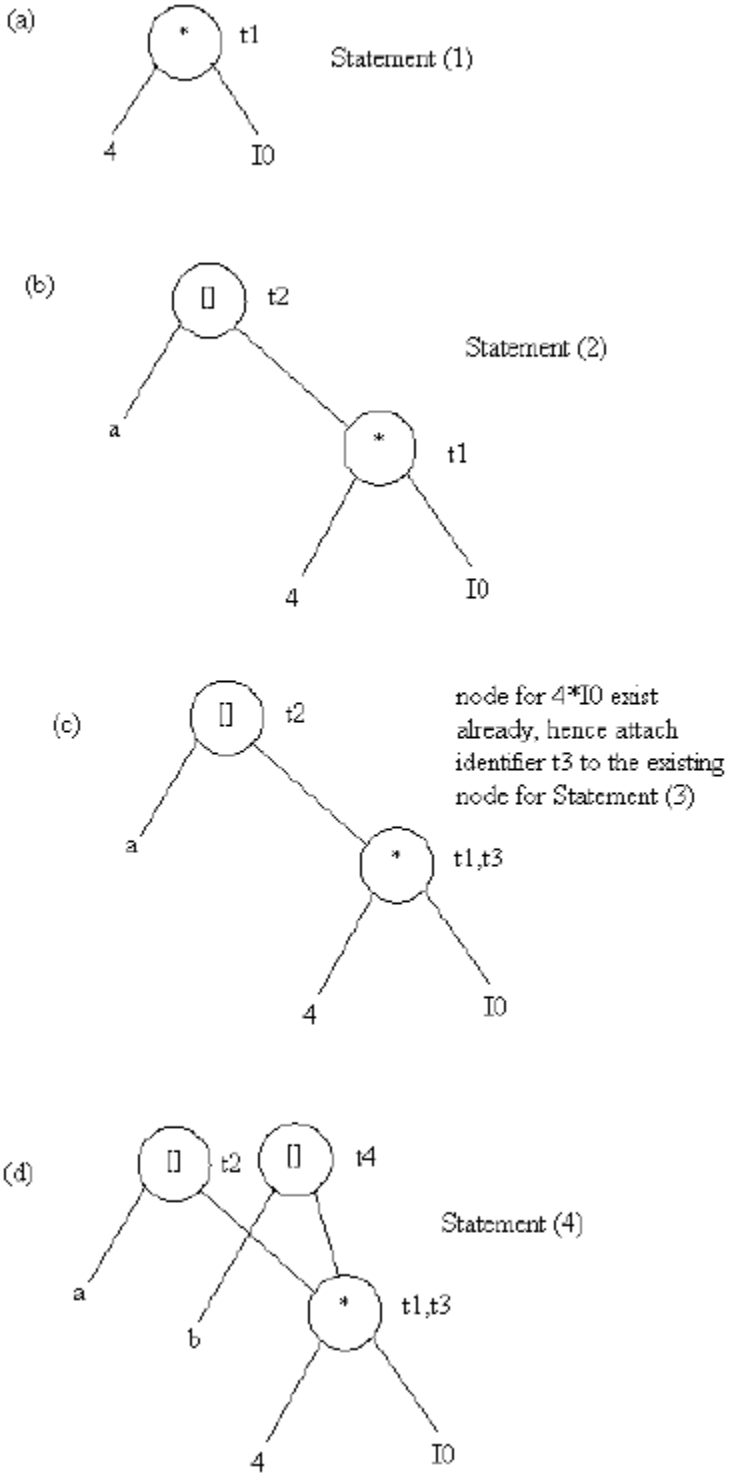
For Case (iii):

Node n will be node(y).

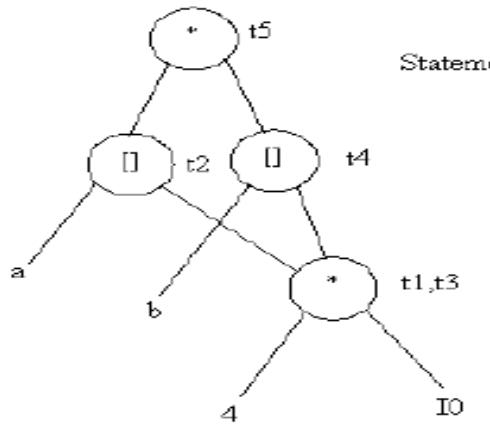
Step 3: Delete x from the list of identifiers for node(x). Append x to the list of attached identifiers for the node n found in step 2 and set node(x) to n.

Example: Construct the DAG representation for the following basic block.

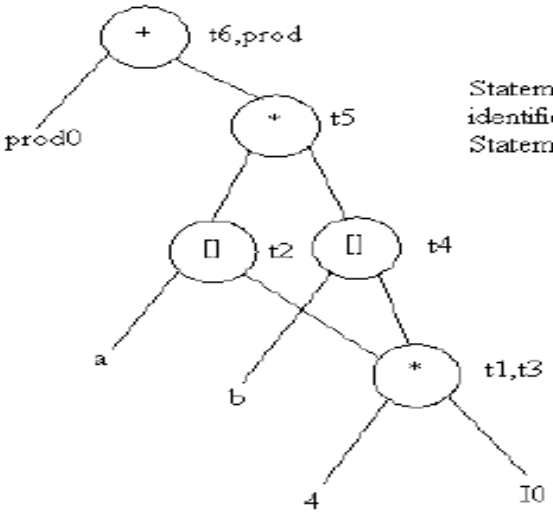
Stages in DAG Construction



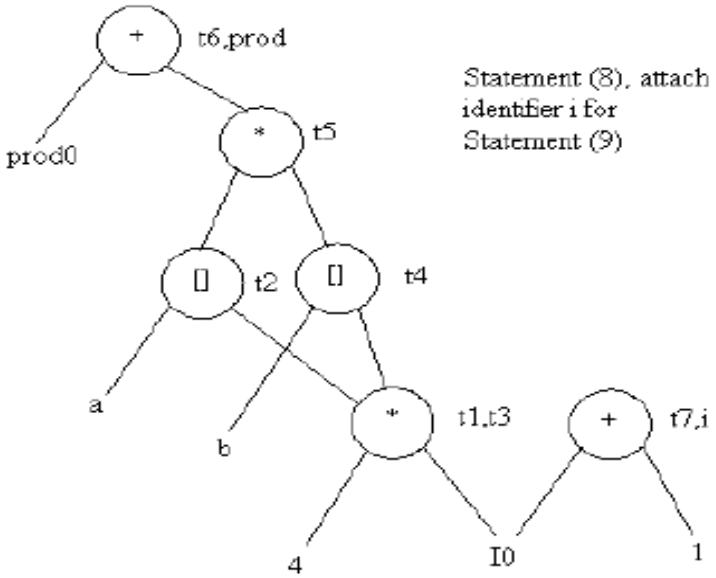
(e)

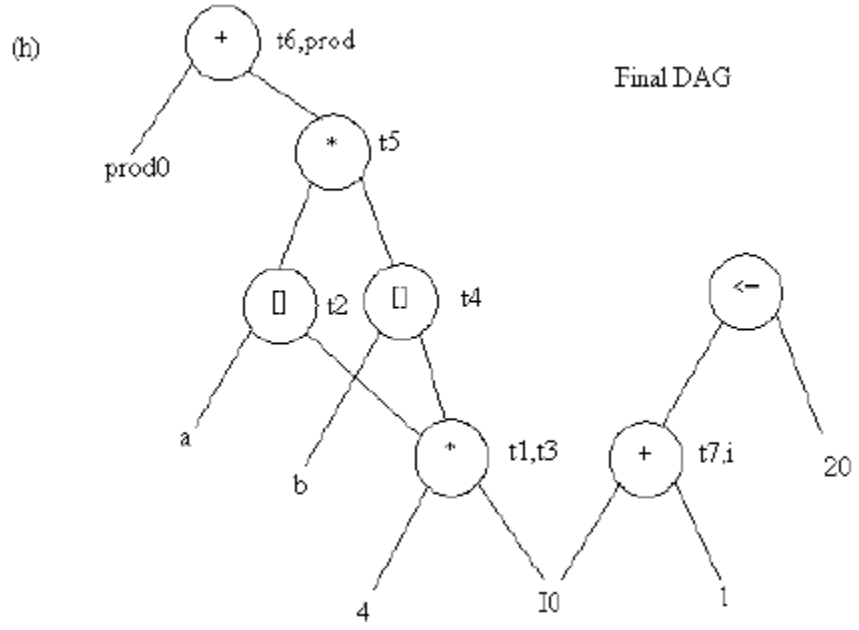

*
t5
Statement (5)
[]
t2
[]
t4
a
b
*
t1,t3
4
I0

(f)


+
t6,prod
prod0
*
t5
Statement (6), attach
identifier prod for
Statement (7)
[]
t2
[]
t4
a
b
*
t1,t3
4
I0

(g)


+
t6,prod
prod0
*
t5
Statement (8), attach
identifier i for
Statement (9)
[]
t2
[]
t4
a
b
*
t1,t3
+
t7,i
4
I0
1



Application of DAG

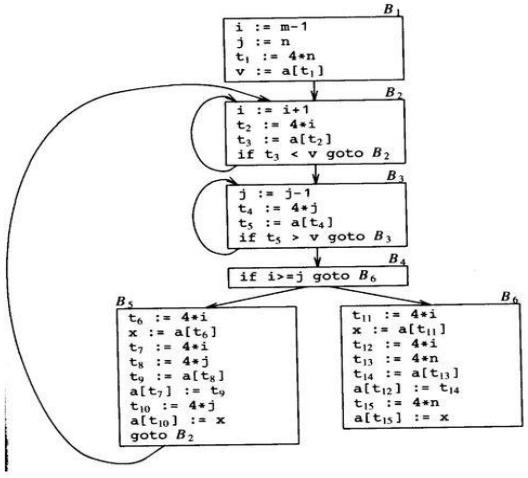
1. We can automatically detect common sub expressions.
2. We can determine which identifiers have their values used in the block.
3. We can determine which statements compute values that could be used outside the block.

Optimization Example:

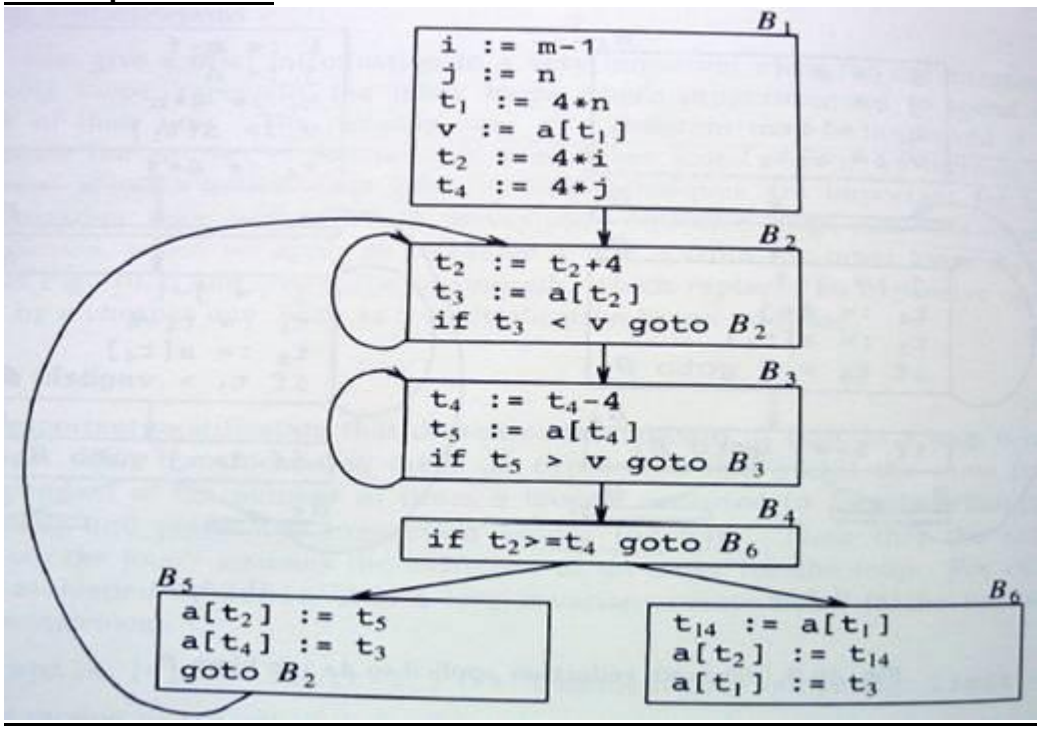
Basic Block

```
(1)   i := m-1
(2)   j := n
(3)   t1 := 4*n
(4)   v := a[t1]
(5)   i := i+1
(6)   t2 := 4*i
(7)   t3 := a[t2]
(8)   if t3 < v goto (5)
(9)   j := j-1
(10)  t4 := 4*j
(11)  t5 := a[t4]
(12)  if t5 > v goto (9)
(13)  if i >= j goto (23)
(14)  t6 := 4*i
(15)  x := a[t6]
```

```
(16)     t7 := 4*i
(17)     t8 := 4*j
(18)     t9 := a[t8]
(19)  a[t7] := t9
(20)     t10 := 4*j
(21)  a[t10] := x
(22)     goto (5)
(23)     t11 := 4*i
(24)     x := a[t11]
(25)     t12 := 4*i
(26)     t13 := 4*n
(27)     t14 := a[t13]
(28)  a[t12] := t14
(29)     t15 := 4*n
(30)  a[t15] := x
```



After Optimization



2.5 Peephole Optimization

- A statement-by-statement code-generations strategy often produce target code that contains redundant instructions and suboptimal constructs .The quality of such target code can be improved by applying "optimizing" transformations to the target program.
- A simple but effective *technique for improving the target code is peephole optimization, a method for trying to improve the performance of the target program by examining a short sequence of target instructions (called the peephole) and replacing these instructions by a shorter or faster sequence, whenever possible.*
- The *peephole is a small, moving window on the target program.* The code in the peephole need not contiguous, although some implementations do require this. It is characteristic of peephole optimization that each improvement may spawn opportunities for additional improvements.

- Transformations of peephole optimizations
  1. **Redundant-instructions elimination**
  2. **Flow-of-control optimizations**
  3. **Algebraic simplifications**
  4. **Use of machine idioms**
  5. **Unreachable Code**

1. Redundant Loads and Stores

If we see the instructions sequence

(1) MOV R0, a

(2) MOV a, R0

We can delete instructions (2) because whenever (2) is executed. (1) will ensure that the value of a is already in register R0. If (2) had a label we could not be sure that (1) was always executed immediately before (2) and so we could not remove (2).

2. Unreachable Code

- Another opportunity for peephole optimizations is the removal of unreachable instructions.
- An unlabeled instruction immediately following an unconditional jump may be removed.
- This operation can be repeated to eliminate a sequence of instructions.
- For example, for debugging purposes, a large program may have within it certain segments that are executed only if a variable debug is 1. In C, the source code might look like:

  #define debug 0

  ….

  If (debug) {

  Print debugging information

  }

  In the intermediate representations the if-statement may be translated as:

  If debug =1 goto L2

  goto L2

  L1: print debugging information

  L2: ……(a)

- One obvious **peephole optimization is to eliminate jumps over jumps** .Thus no matter what the value of debug; (a) can be replaced by:

  If debug ≠1 goto L2

  Print debugging information

  L2: ……(b)

- As the argument of the statement of (b) evaluates to a constant true it can be replaced by

  If debug ≠0 goto L2

  Print debugging information

  L2: ……(c)

- As the argument of the first statement of (c) evaluates to a constant true, it can be replaced by goto L2. Then all the statement that print debugging aids are manifestly unreachable and can be eliminated one at a time.

3. Flows of Control Optimizations

- The unnecessary jumps can be eliminated in either the intermediate code or the target code by the following types of peephole optimizations. We can replace the jump sequence,

goto L1		**goto L2**
….	by the sequence	**….**
L1: gotoL2		**L1: gotoL2**

- If there are now no jumps to L1, then it may be possible to eliminate the statement L1: goto L2 provided it is preceded by an unconditional jump.

- Similarly, the sequence

if a < b goto L1		**If a < b goto L2**
….	**can be replaced by**	**….**
L1: goto L2		**L1: goto L2**

- Finally, suppose there is only one jump to L1 and L1 is preceded by an unconditional goto. Then the sequence

goto L1		**If a < b goto L2**
……..	May be replaced by	**goto L3**
L1: if a < b goto L2		**….**
L3: ……..(1)		**L3: ……(2)**

- While the number of instructions in (1) and (2) is the same, we sometimes skip the unconditional jump in (2), but never in (1).Thus (2) is superior to (1) in execution time

4. Algebraic Simplification

- There is no end to the amount of algebraic simplification that can be attempted through peephole optimization.
- Only a few algebraic identities occur frequently enough that it is worth considering implementing them .For example, statements such as

  x := x + 0 or x := x * 1
- Are often produced by straightforward intermediate code-generation algorithms, and they can be eliminated easily through peephole optimization.

5. Reduction in Strength

- Reduction in strength replaces expensive operations by equivalent cheaper ones on the target machine. Certain machine instructions are considerably cheaper than others and can often be used as special cases of more expensive operators.
- For example, x^2 is invariably cheaper to implement as x*x than as a call to an exponentiation routine.

6. Use of Machine Idioms

- The target machine may have hardware instructions to implement certain specific operations efficiently. For example, some machines have auto-increment and auto-decrement addressing modes.

2.6 Generating Code from DAGs

- The advantage of generating code for a basic block from its DAG representation is that, from a DAG we can easily see how to rearrange the order of the final computation sequence then we can start from a linear sequence of three-address statements or quadruples.

Rearranging the order

- The order in which computations are done can affect the cost of resulting object code.

For example, consider the following basic block,

```
t1 : = a + b
t2 : = c + d
t3 : = e – t2
t4 : = t1 – t3
```

Generated code sequence for basic block,

```
MOV a , R0
ADD b , R0
MOV c , R1
ADD d , R1
MOV R0 , t1
MOV e , R0
SUB R1 , R0
MOV t1 , R1
SUB R0 , R1
MOV R1 , t4
```

Rearranged basic block,
Now t1 occurs immediately before t4.

```
t2 : = c + d
t3 : = e – t2
t1 : = a + b
t4 : = t1 – t3
```

Revised code sequence,

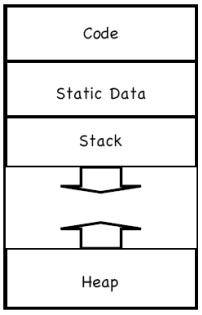
```
MOV c , R0
ADD d , R0
MOV a , R0
SUB R0 , R1
MOV a , R0
ADD b , R0
SUB R1 , R0
MOV R0 , t4
```

In this order, two instructions MOV R0, t1 and MOV t1, R1 have been saved.

4. Run Time Memory Management

4.1 Run Time storage

- The Compiler demands for a block of memory to the operating system, for executing the compiled program. This block of memory is called Run time storage.
- The Run time storage is subdivided into code area, data area, stack and heap.



(i). Code

The size of the target code is fixed. Hence the target code occupies the determined area of the memory.

(ii). Data objects

The amount of memory required for the data objects is known at compile time and can be placed in the determined area of memory. It includes memory for Global variables/constants, variables declared within procedures/blocks (size known at compile time).

(iii). Stack

The Stack is used to keep track of procedure activations.

(iv). Heap

The Heap is used to hold the variables created dynamically at runtime.

4.2 Activation Record

- Any information needed for a single activation of a procedure /sub-function is stored in the block of memory called an ACTIVATION RECORD (sometimes called as the STACK FRAME).

Structure of an Activation Record

Return Value
Actual Parameters
Control Link
Access Link
Saved Machine Status
Local variables
Temporaries

1. Temporaries

The temporary values generated by the compiler during the evaluation of expressions are stored in the temporary section of the activation recode.

2. Local Variables - The local names belonging to the procedure are stored in this field.

3. Saved machine Status

The information about the state of the machine just before the call to the procedure is stored in this field.

It includes return address, the contents of the registers that were used by the calling procedure and this must be restored when the return occurs.

4. Access Link - It refers to the non-local names in other activation record.

5. Control Link - Pointer to the activation record of the caller.

6. Actual parameters

This section is used by the calling procedure to keep the information about the actual parameters that are passed to the called procedure.

7. Return Values

This section is used by the called procedure to store the result of a function call.

STORAGE ALLOCATION STRATEGIES

The techniques to generate and manage the activation records at run time are,

1. **Static Allocation**
2. **Stack Allocation**
3. **Heap Allocation**

1. Static Allocation

- Statically allocated names are bound to storage at compile time. Storage bindings of statically allocated names never change, so even if a name is local to a procedure, its name is always bound to the same storage.
- The binding of name with the amount of storage allocated do not change at runtime. Hence the name of this allocation is static allocation.
- In static allocation the compiler can determine the amount of storage required by each data object. And therefore it becomes easy for a compiler to find the addresses of these data in the activation record

Limitations of Static Allocation strategy

1. The size of the activation record is required & must be known at compile time.
2. Recursive procedures cannot be implemented as all locals are statically allocated.
3. No data structure can be created dynamically as all data is static

2. Stack allocation

- Stack allocation strategy is a strategy in which the storage is organized as stack. This stack is also called **control stack.**
- As activation begins the activation records are pushed onto the stack completion of this activation the corresponding activation records can be popped.
- Local names and parameters are contained in the activation records for the call. This means locals are bound to fresh storage on every call.

Address generation in stack allocation

- The position of the activation record on the stack cannot be determined statically. Therefore the compiler must generate addresses RELATIVE to the activation record.

Limitation:

- The Stack allocation is not suitable if
  - The value of any names wants to be retained when activation ends (**Dangling reference** – occurs when there is a reference to the storage that has been de-allocated)
  - A called activation outlives the caller.

3. Heap allocation

- The heap allocation allocates the continuous block of memory when required for storage of activation records or other data object.
- This allocated memory de-allocated when activation ends. This de-allocated (free) space can be reused by heap manager.
- The efficient heap management can be done by:
  - Creating a linked list for the free blocks and when any memory de-allocated that block of memory is appended in the linked list.
  - Allocate the most suitable block of memory from the linked list.

REGISTER ALLOCATION AND ASSIGNMENT

Register allocation and assignment are important steps in **compiler design** to improve the efficiency of code by managing the usage of the CPU registers. Here's a detailed explanation:

1. REGISTER ALLOCATION:

Register allocation is the process of deciding **which variables** (or intermediate values) will reside in CPU registers during program execution. Since registers are faster than memory, efficient use of them is crucial for performance.

- **Goal**: To minimize the number of variables that need to be loaded from or stored in memory.
- **Challenge**: The number of registers is limited, so not all variables can be kept in registers all the time.

Types of Register Allocation:

- **Global Register Allocation**: Tries to allocate registers over entire functions or even across multiple functions.
- **Local Register Allocation**: Deals with allocating registers within a single basic block (a straight-line sequence of code without branches).

Local Register Allocation and Assignment

Allocation just inside a basic block is called Local Reg. Allocation. Two approaches for local reg. allocation: Top-down approach and bottom-up approach.
Top-Down Approach is a simple approach based on 'Frequency Count'. Identify the values which should be kept in registers and which should be kept in memory.
Algorithm:

1. Compute a priority for each virtual register.
2. Sort the registers into priority order.
3. Assign registers in priority order.
4. Rewrite the code.

Moving beyond single Blocks:

- More complicated because the control flow enters the picture.
- Liveness and Live Ranges: Live ranges consist of a set of definitions and uses that are related to each other as they i.e. no single register can be common in a such couple of instruction/data.

Following is a way to find out Live ranges in a block. A live range is represented as an interval [i,j], where i is the definition and j is the last use.

Global Register Allocation and Assignment:

1. The main issue of a register allocator is minimizing the impact of spill code;
   - Execution time for spill code.
   - Code space for spill operation.
   - Data space for spilled values.
2. Global allocation can't guarantee an optimal solution for the execution time of spill code.
3. Prime differences between Local and Global Allocation:
   - The structure of a global live range is naturally more complex than the local one.
   - Within a global live range, distinct references may execute a different number of times. (When basic blocks form a loop)
4. To make the decision about allocation and assignments, the global allocator mostly uses graph coloring by building an interference graph.
5. Register allocator then attempts to construct a k-coloring for that graph where 'k' is the no. of physical registers.
   - In case, the compiler can't directly construct a k-coloring for that graph, it modifies the underlying code by spilling some values to memory and tries again.
   - Spilling actually simplifies that graph which ensures that the algorithm will halt.
6. Global Allocator uses several approaches, however, we'll see top-down and bottom-up allocations strategies. Subproblems associated with the above approaches.
   - Discovering Global live ranges.
   - Estimating Spilling Costs.
   - Building an Interference graph.

2. REGISTER ASSIGNMENT:

After determining **which variables** should be stored in registers (through allocation), register assignment specifies **which physical register** each variable will use.

- **Goal**: To map virtual registers or variables to physical registers available on the CPU.
- **Challenge**: Ensuring that there are no conflicts (i.e., two variables trying to use the same register at the same time).

Techniques for Register Allocation and Assignment:

1. **Graph Coloring**:
    - The most common approach, where an interference graph is created. Each node represents a variable, and an edge exists between two nodes if they are **live at the same time**.
    - The task is to color this graph with the number of colors equal to the number of available registers. Each color represents a different register.
    - If the graph can't be colored with the available number of registers, **spilling** occurs, where some variables are moved to memory.

      Steps

      1. Identify the live range of each variable
      2. Build an interference graph that represents conflicts between live ranges (two nodes are connected if the variables they represent are live at the same moment)
      3. Try to assign as many colours to the nodes of the graph as there are registers so that two neighbours have different colours.

2. **Linear Scan Allocation**:
    - A simpler method often used in **Just-in-Time (JIT) compilers**.
    - It processes the code in a linear order, assigning registers as needed and spilling when registers are full.
3. **Priority-Based Allocation**:
    - Assigns priority to variables based on factors like how often they are used and allocates registers to higher-priority variables.

Spilling:
When the number of live variables exceeds the available registers, some variables are stored in memory temporarily (spilled) and reloaded when needed. This can slow down the program, so minimizing spills is crucial.

Importance of Register Allocation:

- **Performance**: Efficient register allocation reduces memory accesses, leading to faster code execution.
- **Resource Management**: Allocating registers wisely allows better utilization of limited CPU resources.