Code Optimization

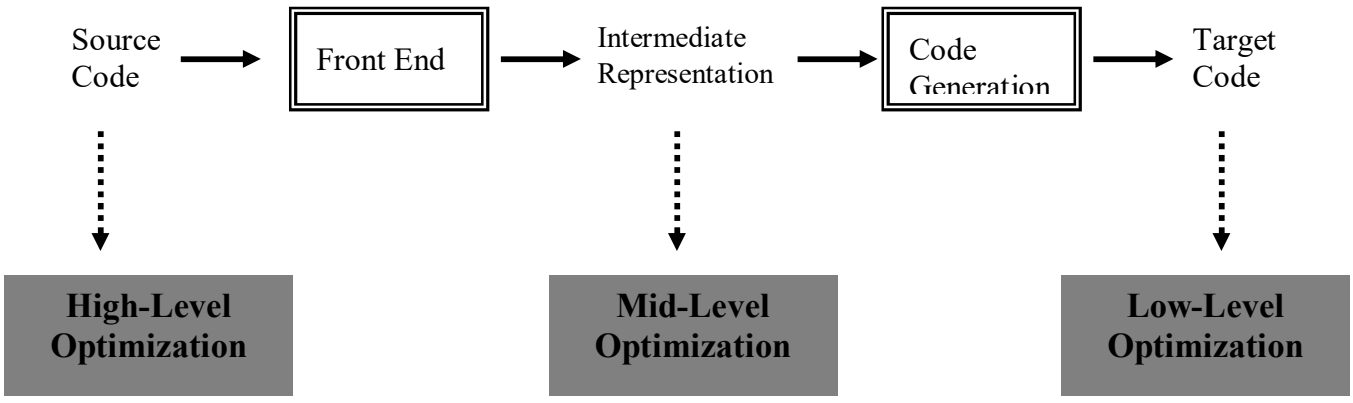
Compilers should produce target code that is as good as can be written by hand. This goal is achieved by program transformations that are called " Optimization " . Compilers that apply code improving transformations are called " Optimizing Compilers ".

Code optimization attempts to increase program efficiency by restructuring code to simplify instruction sequences and take advantage of machine specific features:-

- Run Faster , or
- Less Space , or
- Both (Run Faster & Less Space).

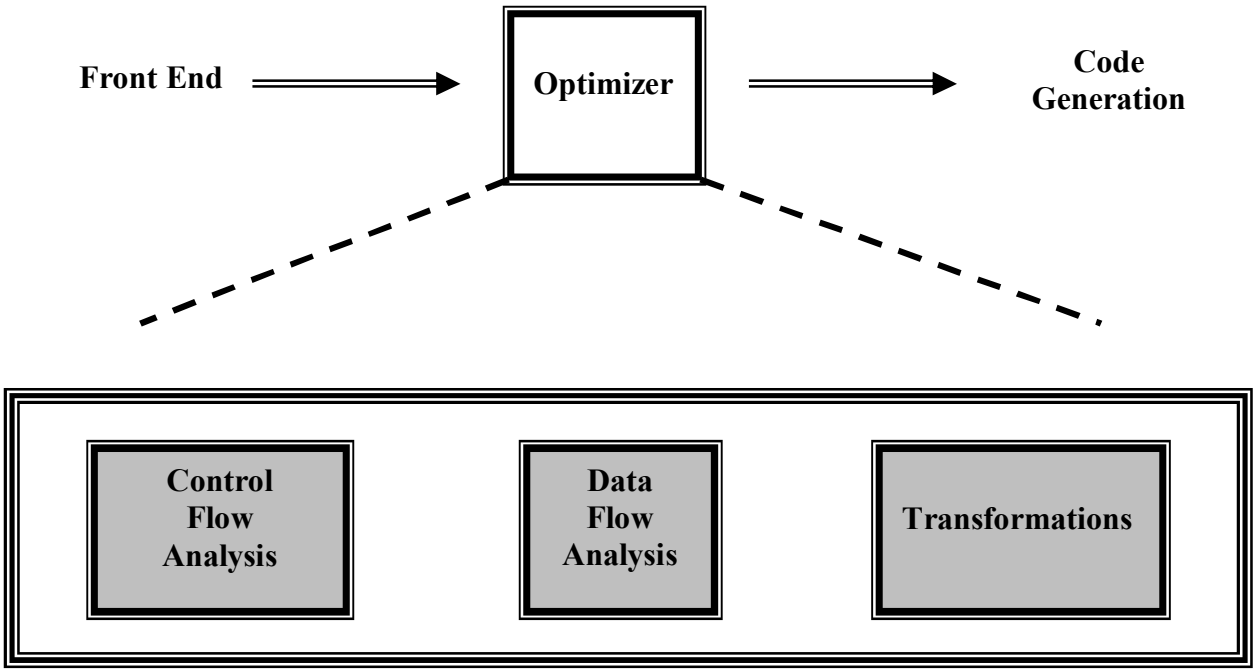
The transformations that are provided by an optimizing compiler should have several properties:-

1. A transformation must preserve the meaning of program. That is , an optimizer must not change the output produce by program for an given input, such as **division by zero.**
2. A transformation must speed up programs by a measurable amount.



Places for Optimization

This lecture concentrates on the transformation of intermediate code (Mid-Optimization or Independent Optimization),this optimization using the following organization:-



Organization of the Optimizer

This organization has the following advantages :-

1. The operations needed to implement high-level constructs are made explicit in the intermediate code.
2. The intermediate code can be independent of the target machine, so the optimizer does not have to change much if the code generator is replaced by one for different machine.

Basic Blocks:-

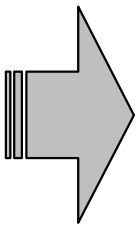
The code is typically devided into a sequence of "Basic Blocks". A Basic Block is a sequence of straight-line code,with no branches " In " or " Out " except a branch "In" at the top of block and a branch "Out" at the bottom of block.

- **Set of Basic Block :** The following steps are used to set the Basic Block:
  1. **Determine the Block beginning:**
     - **i- The First instruction**
     - **ii- Target of conditional & unconditional Jumps.**
     - **iii- Instruction follow Jumps.**

2. Determine the Basic Blocks:
i-There is Basic Block for each Block beginning.
ii-The Basic Block consist of the Block beginning and runs until the next Block beginning or program end.

Example

1) i=0
2) t=0
3) t=t+1
4) i=i+1
5) if I < 10 then goto 3
6) x=t

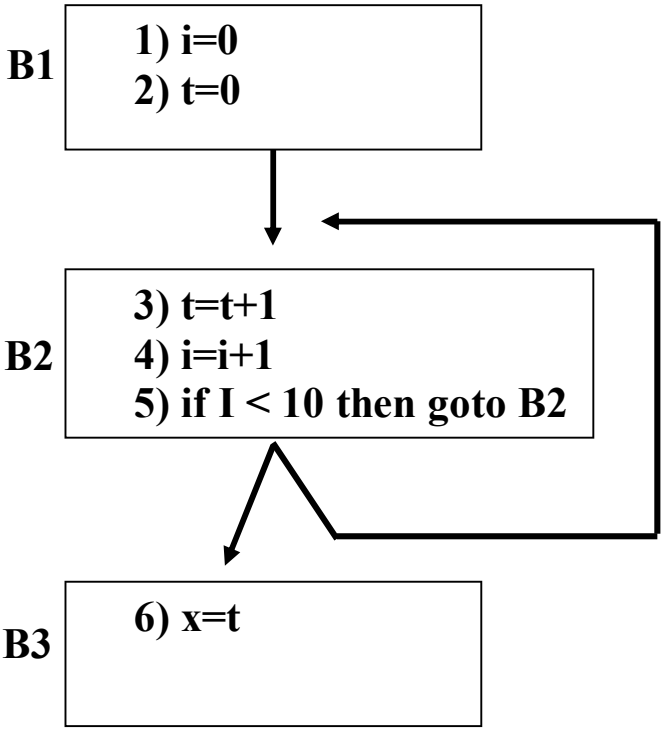


B1
1) i=0
2) t=0

B2
3) t=t+1
4) i=i+1
5) if I < 10 then goto 3

B3
6) x=t

Basic Blocks



Control Flow

Data – Flow Analysis (DFA)

In order to do code optimization a compiler needs to collect information about program as a whole and to distribute this information to each block in the flow graph. DFA provides information about how the execution of a program may manipulate its data , and it provides information for *global optimization* .

There are many DFA that can provide useful information for optimizing transformations. One data-flow analysis determines how definitions and uses are related to each other, another estimates what value variables might have at a given point, and so on. Most of these DFAs can be described by data flow equations derived from nodes in the flow graph.

Reaching Definitions Analysis: All definitions of that variable, which reach the beginning of the block, as follow:

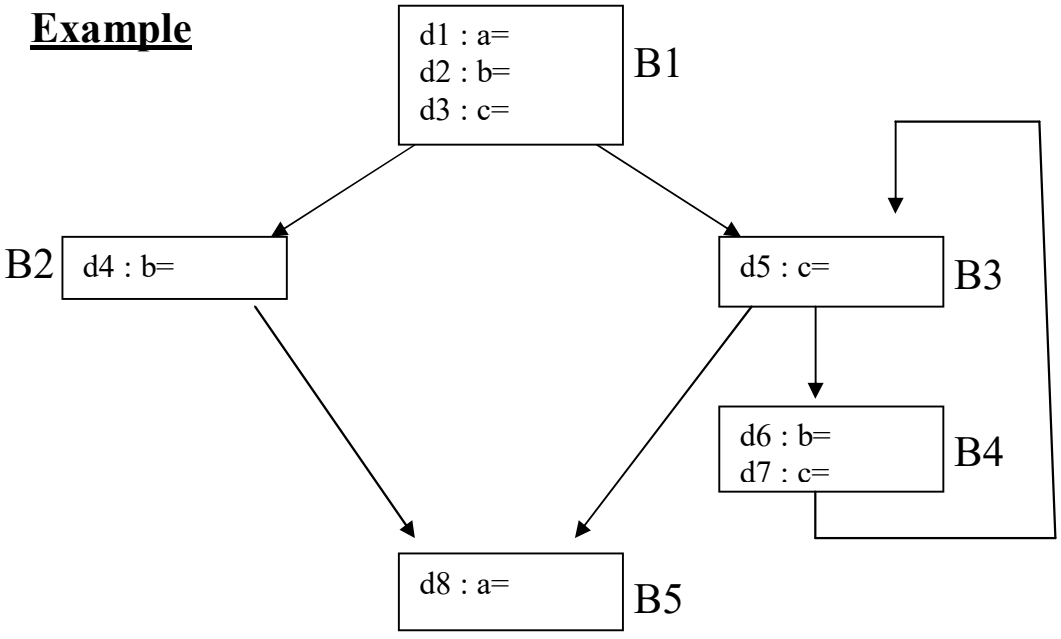
1. **Gen**[B] : contains all definitions *d:v=e* , in block B that *v* is not defined after *d* in B.
2. **Kill**[B] : if *v* is assigned in B , then Kill[B] contains all definitions *d:v=e*,in block different from B.
3. **In**[B] : the set of definitions reaching the beginning of B.

$$\mathbf{In[B]} = \cup\ \mathbf{Out[H]} \quad \text{where } H \in Pred[B]$$

4. **Out[B]** : the set of definitions reaching the end of B.

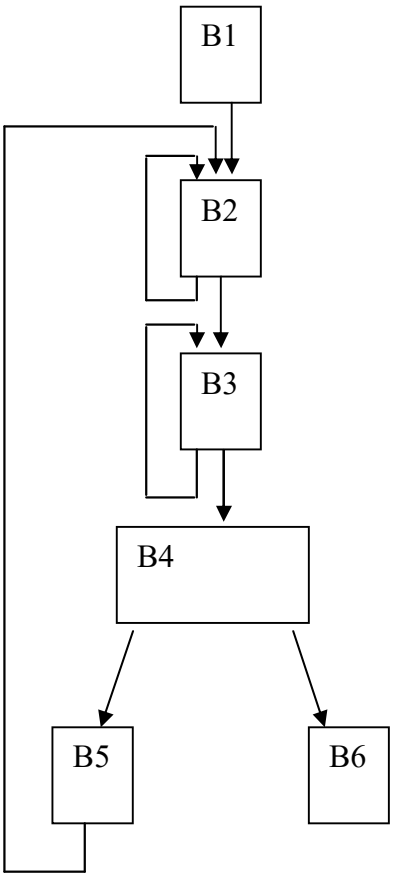
$$\mathbf{Out[B]} = \mathbf{Gen[B]} \cup (\mathbf{In[B]} - \mathbf{Kill[B]})$$

Example



Block	Gen	Kill	In	Out
B1	d1d2d3	d4d5d6d7d8	Ø	d1d2d3
B2	d4	d2d6	d1d2d3	d1d3d4
B3	d5	d3d7	d1d2d3d6d7	d1d2d5d6
B4	d6d7	d2d3d4d5	d1d2d5d6	d1d6d7
B5	d8	d1	d1d2d3d4d5d6	d2d3d4d5d6d8

Loop Information: The simple iterative loop which causes the repetitive execution of one or more *basic blocks* becomes the prime area in which optimization will be considered.Here we determine all the loops in program and limit *headers* & *preheaders* for every loop, for example:



Flow Graph

Loop No.	Header	Preheader	Blocks
1	B2	B1	2-3-4-5-2
2	B2	B1	2-2
3	B3	B2	3-3

Loop Information

Code Optimization Methods

A transformation of program is called " *Local* " if it can performed by looking only at the statements in a *Basic Block*, otherwise, it is called " *Global* " .

Local Transformations:

1. Structure-Preserving Transformations:-
   - Common Subexpression Elimination
   - Dead Code Elimination

2. Algebraic Transformations:-This transformations uses to change the set of expressions ,computed by a basic block, with an algebraically equivalent set. The useful ones are those that simplify expressions or replace expensive operations by cheaper one, such as:

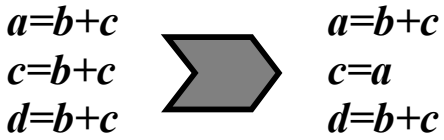
$$\left.\begin{array}{l} x:=x+0 \\ x:=x*1 \\ x:=x/1 \end{array}\right\} \quad \textit{Eliminated}$$

$$x:= y\text{^}2 \quad \Longrightarrow \quad x:=y*y$$

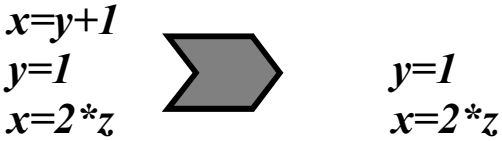
Another class of algebraic transformations is ***Constant Folding*** ,that is, we can evaluate constant expressions at compiler time and replace the constant expressions by their values, for example, the expression 2*3.14 would be replaced by 6.28.

Global Transformations:

1. Common Subexpression Elimination



2. Dead Code Elimination**:** Variable is *dead* if never used



3. Copy Propagation

Origin	**Copy Propagation**	**Dead Code**
x=t3	***x=t3***	
a[t2]=t5	***a[t2]=t5***	***a[t2]=t5***
a[4]=x	***a[4]=t3***	***a[4]=t3***
Goto B2	***Goto B2***	***Goto B2***

4. Constant Propagation

Origin	**Copy Propagation**	**Dead Code**
x=3	***x=3***	
a[t2]=t5	***a[t2]=t5***	***a[t2]=t5***
a[4]=x	***a[4]=3***	***a[4]=3***
Goto B2	***Goto B2***	***Goto B2***

5. Loop Optimization

- **Code Motion:** An important modification that decreases the amount of code in a loop is *Code Motion*. If result of expression does not change during loop(*Invariant Computation*),can hoist its computation out of the loop.

```
For(i=0;i<n;i++)
    A[i]=a[i]+( x*x )/( y*y );
```

```
c=( x*x )/( y*y );
 For(i=0;i<n;i++)
    A[i]=a[i]+c;
```

- **Strength Reduction:** Replaces expensive operat--ions (Multiplies, Divides)by cheap ones (Adds, Subs).For example, suppose the following expression**:**

For(i=1;i<n;i++){v=4*i;s=s+v;} ***i*** *is induction variable*

Then:

v=0;
For(i=1;i<n;i++){ v=v+4; s=s+v; }

Induction Variable: is a variable whose value changes by a constant amount on each loop iteration.

Code Generation

In computer science, code generation is the process by which a compiler's code generator converts some internal representation of source code into a form(e.g., machine code)that can be readily executed by a machine.

Issues in the Design of a Code Generator:-

1. **Input to the Code Generator :**The input to the code generator consists of the intermediate representation of the source program(Optimized IR),together with information in ST that is used to determine the Run Time Addresses of the data objects denoted by the names in IR. Finally, the code generation phase can therefore proceed on the assumption that its input is free of the errors.
2. **Target Programs :** The output of the code generator is the target program. The output code must be **Correct** and of **high Quality**, meaning that it should make effective use of the resources of the target machine. Like the IR ,this output may take on a variety of forms:
    a. **Absolute Machine Language //** Producing this form as output has the advantage that it can placed in a fixed location in memory and immediately executed. A small program can be compiled and executed quickly.
    b. **Relocatable Machine Language //** This form of the output allows subprograms to be compiled separately. A set of relocatable object modules can be linked together and loaded for execution by linking-loader.
3. **Memory Management :** Mapping names in the source program to addresses of data objects in run time memory. This process is done cooperatively by the Front-end & code generator.
4. **Major tasks in code generation :** In addition to the basic conversion from IR into a linear sequence of machine instructions, a typical code generator tries to optimize the generated code in some way. The generator may try to use