

Chapter:

Code Optimization

Topics in this Chapter:

- Machine Dependent Optimization
- Machine Independent Optimization
 - Function Preserving Transformations
 - Loop Optimizations

12.1 What is Code Optimization?

Exam Questions:

Q) **Explain Code Optimization in Compilers.** (Dec '04 [IT], Dec '05 [IT], Dec '08 [IT] – 10M, May '05 [Comps], June '07 [IT] – 5M)

Q) **What is Code Optimization in Compilers? What is the need of Optimization? Explain types of Code Optimization techniques.** (May '05 [IT] – 10 M)

A piece of code is said to be *optimized*, if it consumes less processing time and/or less memory space.

Optimization attempts to improve structure of the code to get an optimized version; it must not affect the meaning of the code.

12.2 Types of Optimization:

Code Optimization can be divided into two broad categories:

- 1. Machine Dependent Optimization**
- 2. Machine Independent Optimization**

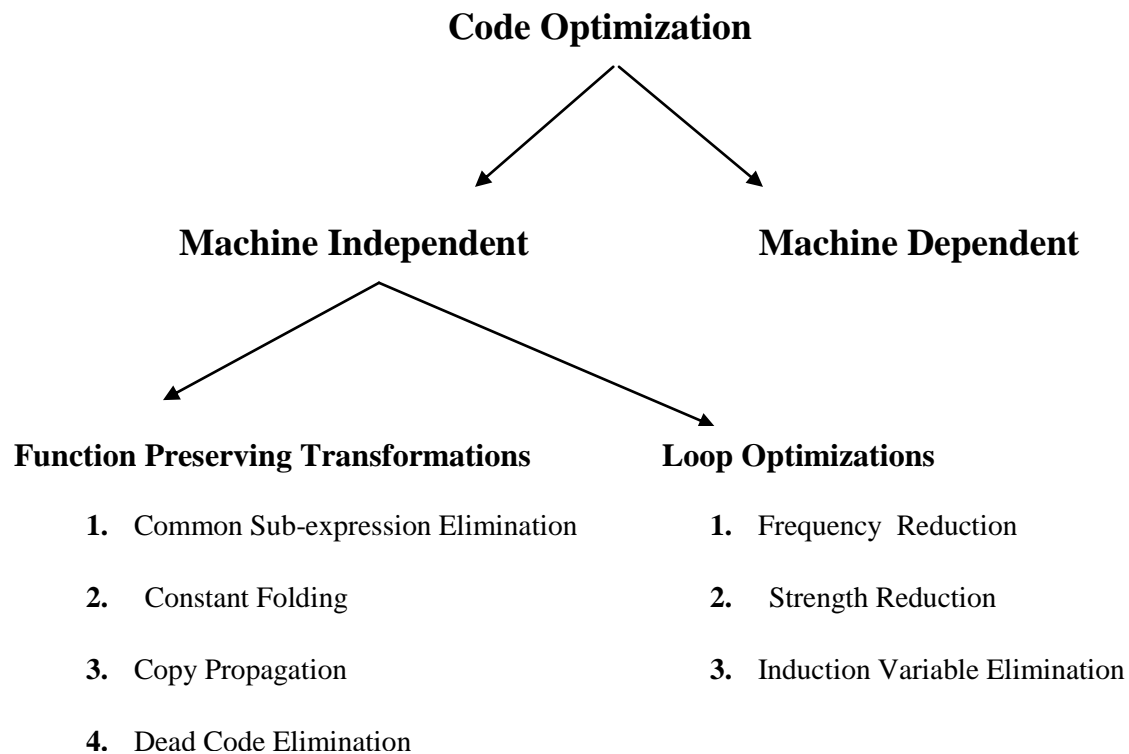


Fig: 12.1: Types of Optimizations

12.2.1 Machine Dependent Optimization

Machine Dependent Optimization focuses on how features of the native machine language and its architecture can be exploited to generate optimized code.

There are different ways of performing machine dependent optimization. Some of them are:

1. **Register operations are faster.** So if possible, instructions using memory-resident operands must be replaced by instructions using registers as operands.
2. **Special addressing modes or control instructions of the native machine language** can be used to optimize the code.
3. **Parallelism could be used** i.e. instructions addressing different functional units could be clubbed together for faster operation.

12.2.1.1 Sources of Machine Dependent Optimization

- a. Jumps to Jumps must be avoided:** Unconditional jump to another unconditional jump instruction must be avoided as it only wastes time and resources.

For example:

| | | |
|--------------------------|---------------------------|------------------|
| <i>jmp down1</i> | | <i>jmp down2</i> |
| | After Optimization | |
| <i>down1: jmp down 2</i> | → | |
| | | |
| <i>down2:</i> | | |
| | | |

- b. Jumps to calls must be avoided:** Unconditional jump to a procedure call instruction must be removed as it's unnecessary.

For example:

| | | |
|---------------------------|---------------------------|--------------------|
| <i>jmp down1</i> | | <i>call myproc</i> |
| | After Optimization | |
| <i>down1: call myproc</i> | → | |

- c. Call Recursion Elimination:** Every call to a procedure requires pushing current execution state onto stack and popping it back when procedure returns (i.e. house-keeping operations). Instead of calling a procedure from inside of another procedure, we can replace the call with a jump to address of the procedure. This saves the time and resources required for house-keeping operations.

For example:

| | | |
|---------------------------|---|---------------------|
| <i>jmp down1</i> | | <i>myproc:</i> |
| | | <i>call myproc2</i> |
| | | |
| <i>down1: call myproc</i> | ↗ | |

After Optimization:

jmp down1

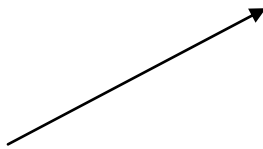
.....

down1: call myproc

myproc:

jmp [addr of myproc2]

.....



12.2.2 Machine Independent Optimization

There are two broad classes of Machine Independent Optimization (*principle sources of optimization*):

1. Function Preserving Transformations
2. Loop Optimizations

12.2.2.1 Function Preserving Transformations

Function Preserving Transformations focus on optimizing arithmetic expressions so that they can be evaluated more efficiently.

There are four types of Function Preserving Transformations:

1. **Common Sub-expression Elimination**
2. **Constant Folding**
3. **Copy Propagation**
4. **Dead Code Elimination**

1. Common Sub-expression Elimination – If an expression gets repeated more than once *and* values of its variables do not change between its occurrences, then the expression is called a common sub-expression. This common sub-expression can be computed once and its value can be used in all remaining occurrences.

For example,:

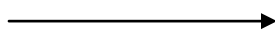
*t1=4*i;*

x=a[t1];

*t2=4*i;*

y=a[t2];

After Optimization



*t1=4*i;*

x=a[t1];

y=a[t1];

2. Constant Folding - If an expression involves all constants, it can be evaluated at compile-time itself; it reduces execution time of the program.

For example,:

| | | |
|------------------------|--------------------------------|----------------------|
| <code>A=3.12/2;</code> | After Optimization → | <code>A=1.56;</code> |
|------------------------|--------------------------------|----------------------|

3. Copy Propagation – When value of a variable is assigned to multiple variables, we can eliminate assignment operations that use variables (stored in main memory) with the ones using temporary variables (stored in CPU registers). The main idea here is that read/write operations on registers are much faster and cheaper than load/store operations on memory.

For example,:

| | | |
|--------------------|--------------------------------|--------------------|
| <code>x=t3;</code> | | <code>x=t3;</code> |
| <code>.....</code> | | <code>.....</code> |
| <code>y=x;</code> | After Optimization → | <code>y=t3;</code> |
| <code>.....</code> | | <code>.....</code> |
| <code>z=y;</code> | | <code>z=t3;</code> |

4. Dead Code Elimination – If a piece of code is not reachable via any program execution path, then it's called as *dead code*. Dead code means computations are performed even if they are of no use; so it must be removed.

For example,:

```
boolVar=false;
if (boolVar==true) then    //the condition will always return false
    print ("boolVar is true");    // so this is piece of code is dead as it's never reached
```

After Optimization: `boolVar=false;`
 `.....` *// dead code eliminated*

12.2.2.2 Loop Optimizations


Exam Questions:

Q) Explain types of Loop Optimization techniques with the help of suitable examples. (Dec '06 [Comps] – 10 M)

1. **Frequency Reduction (or Code Motion)**
2. **Strength Reduction**
3. **Induction Variable Elimination**

1. Frequency Reduction (or Code Motion) – If the result of an expression remains same during all the iterations of the loop, the repeating expression can be placed outside of the loop. This way, it gets computed only once and not many times over (i.e. *frequency of its computation* gets reduced).

For example,:

| | | |
|---|--|---------------------------------------|
| <pre>while (j>=0) i=2; // value of i doesn't change here</pre> | <p>After Optimization</p>  | <pre>i=2; while (j>=0)</pre> |
|---|--|---------------------------------------|

2. Strength Reduction – Operations requiring more CPU cycles can be replaced with the ones that require less CPU cycles.

For example, Multiplication by n is a costlier operation than adding the number n times. So, multiplication by n can be replaced by n -times addition.

| | | |
|--------------------|--|-------------------------------|
| <pre>t1=4*i;</pre> | <p>After Optimization</p>  | <pre>t1=i + i + i + i ;</pre> |
|--------------------|--|-------------------------------|

3. Induction Variable Elimination – If changes to a variable inside the loop form an arithmetic progression (i.e. it varies based on some formula inside the loop), then that variable is called an Induction Variable. If a loop involves 2 or more such induction variables (one being dependent on the other), one of them can be eliminated.

For example,:

| | | |
|-------------------------------|------------------------------|--------------------------------|
| <i>i</i> =1; | | <i>t</i> 1=0; |
| <i>while</i> (<i>i</i> <=20) | $\xrightarrow{\hspace{1cm}}$ | <i>while</i> (<i>t</i> 1<=76) |
| | After Optimization | |
| { <i>t</i> 1=4* <i>i</i> ; | | { <i>t</i> 1= <i>t</i> 1+4; |
| | | |
| <i>i</i> = <i>i</i> +1; } | | } |

12.3 Chapter Summary

- A piece of code is said to be optimized if it consumes less processing time and/or less memory space. Code Optimization must only attempt to improve the program structure for efficiency; it must not affect meaning of the program.
- Two basic types of optimizations: **machine-dependent** and **machine-independent**.
- Machine-dependent optimization focuses on how features of the native machine language and architecture can be exploited to generate efficient code. (For example, Jumps-to-Jumps, Jumps-to-calls, Call Recursion Elimination.)
- Machine-independent are further subdivided into two types:
 1. **Function preserving Transformations** – Optimize arithmetic expressions.
 2. **Loop Optimizations** – Optimize loops for efficiency.
- **Function preserving transformations** include following four sub types:
 1. **Common Sub-expression Elimination** – Evaluate commonly occurring sub expressions only once.
 2. **Constant Folding** – Evaluate “all-constant” expressions at compile-time itself.
 3. **Copy Propagation** – While copying one value to many variables, replace load/store operations from memory with register-based operations.

- 4. Dead Code Elimination** – Eliminate code that is never going to be reached.
- Loop Optimizations have following three sub types:
 - 1. Frequency Reduction** – If an expression inside loop evaluates to same value every time, move it and place it outside the loop.
 - 2. Strength Reduction** – Replace operations requiring more CPU cycles with the one requiring less CPU cycles
 - 3. Induction Variable Elimination** – Eliminate multiple inter-dependent variables that form arithmetic progression.

12.4 Expected Viva Questions

Q.1) What is optimized code? What is main aim of code optimization?

Q.2) Give examples of machine-dependent optimizations

Q.3) What is copy propagation? What is “dead” code?

Q.4) What is “strength” reduction?

Q.5) What are “induction” variables?