



# Optimization

March 1<sup>st</sup>, 2017

Michael Liut  
[liutm@mcmaster.ca](mailto:liutm@mcmaster.ca)  
[michael.liut@utoronto.ca](mailto:michael.liut@utoronto.ca)



## TABLE OF CONTENTS

SLIDES 1-2	TITLE PAGE AND TABLE OF CONTENTS
SLIDE 3	WHAT IS OPTIMIZATION?
SLIDES 4 - 5	MACHINE INDEPENDENT/DEPENDENT
SLIDES 6 - 10	BASIC BLOCKS
SLIDE 11	LEVELS OF OPTIMIZATION
SLIDE 12	KEY OPTIMIZATION FACTS
SLIDES 13 - 16	CONTROL FLOW GRAPHS AND QA
SLIDES 17 - 21	PEEPHOLE OPTMIZATION AND BREAK
SLIDES 22 - 31	COMMON TECHNIQUES AND EXERCISES
SLIDES 32 - 33	REFERENCES, QA AND CLOSURE

## What is Optimization?

The process of analyzing and transforming code to make it more **efficient**\* without affecting the output or end-result.

Two Types:

1. Machine Independent

- e.g. reduction of repeat assignments, improving parse tree mapping to intermediary representation, etc...

2. Machine Dependent

- e.g. register allocation, direct memory addressing, etc...

**efficient**\*: referring to time or space.



## Machine Independent

This type of optimization is where the compiler receives some *Intermediary Representation* and transforms it. The portion of code transformed in this stage does not involve any CPU registers nor does it involve any memory allocation.

For example (loop optimization):

<pre>while (cnt &lt; 100) {     val = 10;     cnt = cnt + val; }</pre>	<pre>val = 10; while (cnt &lt; 100) {     cnt = cnt + val; }</pre>
Before Optimization	After Optimization

**Note:** the modification of the repeated assignment saves CPU cycles.

## Machine Dependent

This type of optimization happens when the target code is being transformed to the target machine architecture. It involves CPU registers and may have **absolute memory references\***. Machine-dependent optimization uses information about the limits and special features of the target machine to produce code which is shorter or which executes more quickly on the machine.

**absolute memory references\***: specifying the actual address of a memory location versus the distance from another address (relative memory references).

## Basic Blocks

A maximal sequence of consecutive three-address instructions with the following properties:

1. The flow of control can only enter the basic through the first instruction.
2. Control will leave the block without halting or branching, except possibly at the last instruction.

Basic blocks become the nodes of a flow graph, with edges indicating the order.

## Basic Blocks Example

```
1) i = 1
2) j = 1
3) t1 = 10 * i
4) t2 = t1 + j
5) t3 = 8 * t2
6) t4 = t3 - 88
7) a[t4] = 0.0
8) j = j + 1
9) if j <= 10 goto (3)
10) i = i + 1
11) if i <= 10 goto (2)
12) i = 1
13) t5 = i - 1
14) t6 = 88 * t5
15) a[t6] = 1.0
16) i = i + 1
17) if i <= 10 goto (13)
```

```
for i from 1 to 10 do
  for j from 1 to 10 do
    a[i,j]=0.0

for i from 1 to 10 do
  a[i,i]=0.0
```

## Identifying Basic Blocks

**Input:** a sequence of instructions  $instr(i)$

**Output:** a list of basic blocks


**Method:**

1. Identify **Leaders\***.
2. Iterate: add subsequent instructions to basic block until you reach another leader.

**Leaders\*** are:

1. The first instruction of a basic block.
2. Any instruction that is the target of a conditional/unconditional jump.
3. Any instruction that immediately follows a conditional/unconditional jump.





## Can you find the Leaders and Basic Blocks?

**Input:** a sequence of instructions *instr(i)*

**Output:** a list of basic blocks

**Method:**

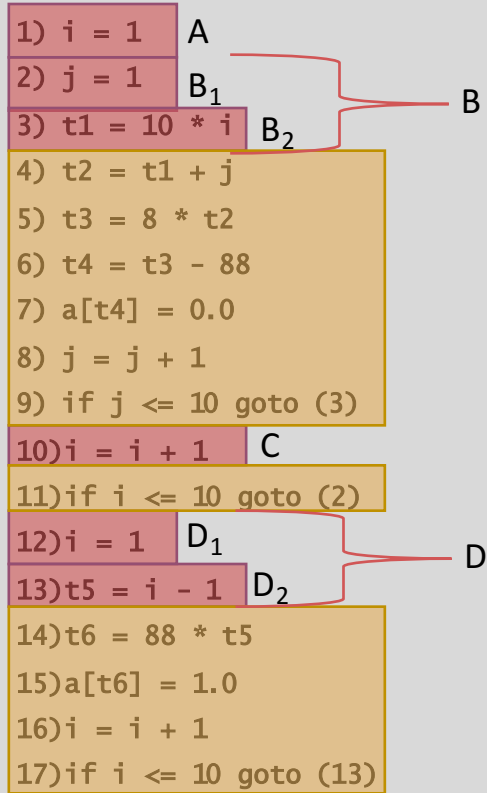
1. Identify **Leaders\***.
2. Iterate: add subsequent instructions to basic block until you reach another leader.

**Leaders\*** are:

1. The first instruction of a basic block.
2. Any instruction that is the target of a conditional/unconditional jump.
3. Any instruction that immediately follows a conditional/unconditional jump.

```
1) i = 1
2) j = 1
3) t1 = 10 * i
4) t2 = t1 + j
5) t3 = 8 * t2
6) t4 = t3 - 88
7) a[t4] = 0.0
8) j = j + 1
9) if j <= 10 goto (3)
10) i = i + 1
11) if i <= 10 goto (2)
12) i = 1
13) t5 = i - 1
14) t6 = 88 * t5
15) a[t6] = 1.0
16) i = i + 1
17) if i <= 10 goto (13)
```

## Basic Blocks Example



Leaders

Basic Blocks

Code

## Levels of Optimization

<ul style="list-style-type: none"><li>• inside a basic block</li></ul>	<ul style="list-style-type: none"><li>• across basic blocks</li><li>• whole procedure analysis</li></ul>	<ul style="list-style-type: none"><li>• across procedures</li><li>• whole program analysis</li></ul>
<b>Local</b>	<b>Global</b> <i>(Intra-Procedural)</i>	<b>Inter-Procedural</b>

## Key Optimization Facts

1. 80% of a program's execution time is spent executing 20% of the code.
  - Known as the 80/20 rule
  - Performance-hungry programs change this rule to 90/10
  - Spend time targeting that 10%/20% and optimize it
2. "Premature optimization is the root of all evil" – Donald Knuth.
  - Optimization can introduce new, subtle bugs
  - Optimization usually makes code harder to understand and maintain
3. Ensure your code is fully functioning prior to optimizing.
  - Optimize the common case, even at the cost of making the uncommon case slower
  - Document optimization carefully
  - Keep the non-optimized code

## Control-Flow Graphs

**Node:** an instruction or sequence of instructions (a **Basic Block**).

- Two instructions (i and j) in the same basic block iff execution of i guarantees execution of j. Essentially  $i \rightarrow j$  (“j” is functionally dependent on “i”).

**Directed Edge:** **potential** flow of control.

**Distinguished Start Node:** distinguished start node (**entry and exit**).

- First and last instruction in the program.

## Control-Flow Edges

**Basic Blocks** = nodes

**Edges:** Add directed edge between  $B_1$  and  $B_2$  if:

- the branch from the last statement of  $B_1$  to the first statement of  $B_2$  ( $B_2$  is a leader);
- $B_2$  immediately follows  $B_1$  in program order and  $B_1$  does not end with an unconditional branch (goto).

**Note:**  $B_1$  is a predecessor of  $B_2$ ,  
while  $B_2$  is a successor of  $B_1$ .

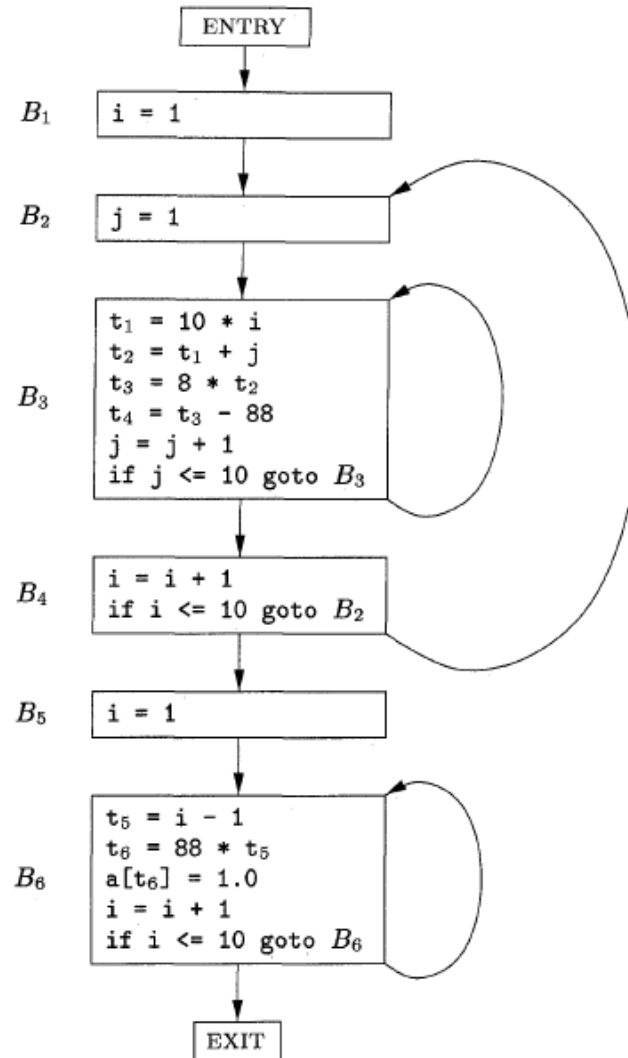
**Input:** block(i), sequence of basic blocks

**Output:** CFG where nodes are basic blocks

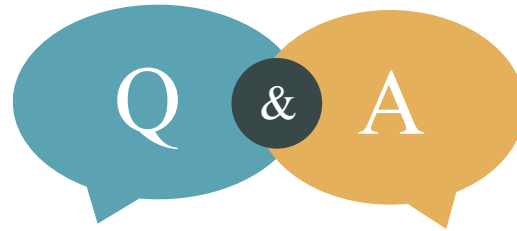
```
for i = 1 to the number of blocks
  x = last instruction of block(i)
  if instr(x) is a branch
    for each target y of instr(x),
      create edge (i -> y)
  if instr(x) is not unconditional branch,
    create edge (i -> i+1)
```

**Control-Flow Edge Algorithm**

## CFG Example



Questions?



DOES ANYONE HAVE ANY QUESTIONS?





## Peephole Optimization

Machine code instructions that could be examined and potentially replaced by less or more efficient instructions. Peephole, as in a “window” size of code to examine.

For example, let's look at the following **Java bytecode\***:

...	...
aload 1	aload 1
aload 1	dup
mul	mul
...	...
Before Optimization	After Optimization

**Note:** it is assumed that the dup operation is more efficient than the aload x operation.

**Java bytecode\*:** the instruction set of the *Java Virtual Machine*.

An aerial photograph of a city skyline, likely New York City, with a semi-transparent white circle in the center. The circle contains the text "Removing Redundant Code Exercise". The background is a blue-tinted aerial view of the city, showing numerous skyscrapers and the Hudson River. A decorative bar at the bottom of the slide consists of five colored segments: blue, red, black, orange, and green.

Removing  
Redundant Code  
Exercise

## Redundant Code Removal

Given the following snippet of code, eliminate the redundant load stores.

<pre>a = b + c; d = a + e;</pre>	<pre>MOV  b, R0  # Copy b to the register ADD  c, R0  # Add c to the register MOV  R0, a   # Copy the register to a MOV  a, R0  # Copy a to the register ADD  e, R0  # Add e to the register MOV  R0, d   # Copy the register to d</pre>
High-Level	Instructions

## Redundant Code Removal

Given the following snippet of code, eliminate the redundant load stores.

```
MOV  b, R0  # Copy b to the register
ADD  c, R0  # Add c to the register
MOV  R0, a   # Copy the register to a
MOV  a, R0  # Copy a to the register
ADD  e, R0  # Add e to the register
MOV  R0, d   # Copy the register to d
```

Before Optimization

```
MOV  b, R0  # Copy b to the register
ADD  c, R0  # Add c to the register
MOV  R0, a   # Copy the register to a
ADD  e, R0  # Add e to the register
MOV  R0, d   # Copy the register to d
```

After Optimization



## INTERMISSION

TAKE 5-10 MINUTES



## Common Techniques

1. **Constant Folding:** evaluate common sub expressions in advance.
2. **Constant Propagation:** substituting values of known constants and expressions.
3. **Strength Reduction:** replacing slow operations with faster equivalents.
4. **Null Sequences:** deleting useless operations.
5. **Combine Operations:** replace several operations with one equivalent.
6. **Algebraic Laws:** use algebraic laws to simplify or re-order instructions.
7. **Special Case Instructions:** use instructions for special operand cases.
8. **Address Mode Operations:** use address modes to simplify code.

## Constant Folding

Example 1:

```
int foo (void)
{
    return (128 * 32);
}
```

Before Optimization

```
int foo (void)
{
    return (4096);
}
```

After Optimization

Example 2\*:

```
int foo (int x)
{
    return (x * 0);
}
```

Before Optimization

```
int foo (void)
{
    return (0);
}
```

After Optimization

\***NOTE:** we do not need to know what x is, as it will always evaluate to 0.

## Constant Propagation

1. Propagate x:

```
...  
int x = 14;  
int y = 7 - x / 2;  
return y * (28 / x + 2);  
...
```

Before Optimization

```
...  
int x = 14;  
int y = 7 - 14 / 2;  
return y * (28 / 14 + 2);  
...
```

After Optimization

2. Continued Propagation  
(& Dead Code Elimination\*):

```
...  
int x = 14;  
int y = 7 - x / 2;  
return y * (28 / x + 2);  
...
```


Before Optimization

```
...  
int x = 14;  
int y = 0;  
return 0;  
...
```

After Optimization

\*Dead Code Elimination (DCE) is a technique whereby code that does affect the program's results are removed.



An aerial photograph of a city skyline, likely New York City, with a circular white overlay in the center. The overlay contains the text "Constant Folding and Propagation Exercise". The background is a warm, golden-brown color. At the bottom of the image, there is a horizontal bar with five colored segments: blue, red, dark blue, orange, and green.

Constant Folding  
and Propagation  
Exercise

## Exercise

Reduce the following snippet of code, use constant folding and propagation.

```
int a = 30;
int b = 9 - (a / 5);
int c;

c = b * 4;

if (c > 10) {
    c = c - 10;
}

return c * (60 / a);
```

Before Optimization

1. Apply constant propagation
2. Apply constant folding
3. Repeat 1. and 2. twice
4. Once a and b are simplified (i.e. become constant values), apply DCE.

HINTS

## Exercise

Reduce the following snippet of code, use constant folding and propagation.

```
int a = 30;
int b = 9 - (a / 5);
int c;

c = b * 4;

if (c > 10) {
    c = c - 10;
}

return c * (60 / a);
```

Before Optimization

```
int c;

if (true) {
    c = 2;
}

return c * 2;
```

```
return 4;
```

After Optimization

## Strength Reduction

- The compiler is interested in loop invariants (values not changing in a loop) and induction variables (values that are being iterated each time in a loop).

Let's look at an example for expressions involving a loop invariant  $c$  and an induction variable  $i$ :

```
c = 7;
```

```
for (int i = 0; i < N; i++)  
{  
    y[i] = c * i;  
}
```

Before Optimization

```
c = 7;  
k = 0;
```

```
for (int i = 0; i < N; i++)  
{  
    y[i] = k;  
    k = k + c;  
}
```

After Optimization

**NOTE:** the optimized result replaces multiplication with successive weaker additions.



Strength Reduction  
Exercise



## Strength Reduction

Given the following lines of code, utilize strength reduction to optimize them.

```
1. a = b * 2;
```

```
2. a = b / 2;
```

Before Optimization

```
1. a = b + b;
```

or  
a = b << 1;

```
2. a = b >> 1;
```

After Optimization

## Algebraic Sequences

It is beneficial to recognize single instructions with a constant operand, for example:

1.  $a + 0 = a;$

2.  $a * 0 = 0;$

3.  $a * 1 = a;$

4.  $a / 1 = a;$

**Null Sequences**

1.  $a * 2 = a + a;$

2.  $a / 2 = a * 0.5;$

**Strength  
Reduction**

1.  $3.14 * 2 = 6.28;$

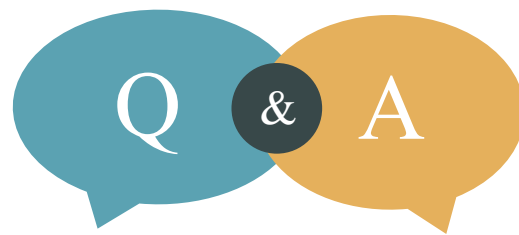
**Constant Folding**

## References

1. Advanced Compiler Design & Implementation, Steven S. Muchnick. University of Kansas.
2. Compiler Design – Code Optimization, Tutorialspoint. <[https://www.tutorialspoint.com/compiler\\_design/compiler\\_design\\_code\\_optimization.htm](https://www.tutorialspoint.com/compiler_design/compiler_design_code_optimization.htm)>
3. Optimisation, Mick O'Donnell. Autonomous University of Madrid. <[http://arantxa.ii.uam.es/~modonnel/Compilers/08\\_1\\_OptimisationI.pdf](http://arantxa.ii.uam.es/~modonnel/Compilers/08_1_OptimisationI.pdf)>
4. Introduction to Optimization, Yao Guo. School of EECS, Peking University.  
<[https://www.google.ca/url?sa=t&rct=j&q=&escr=s&source=web&cd=1&cad=rja&uact=8&ved=0ahUKEwiA-fe4\\_qTSAhUF5oMKHQK6BAIQFggqMAA&url=http%3A%2F%2Fsei.pku.edu.cn%2F~yaoguo%2FACT11%2Fslides%2Flect2-opt.ppt&usg=AFQjCNEtcCtJk4D4T4ThjZZCC\\_a12nUlXw&sig2=VCibINeR0I\\_p8FzKOzitow](https://www.google.ca/url?sa=t&rct=j&q=&escr=s&source=web&cd=1&cad=rja&uact=8&ved=0ahUKEwiA-fe4_qTSAhUF5oMKHQK6BAIQFggqMAA&url=http%3A%2F%2Fsei.pku.edu.cn%2F~yaoguo%2FACT11%2Fslides%2Flect2-opt.ppt&usg=AFQjCNEtcCtJk4D4T4ThjZZCC_a12nUlXw&sig2=VCibINeR0I_p8FzKOzitow)>
5. Peephole Optimization, Wikipedia. <[https://en.wikipedia.org/wiki/Peephole\\_optimization](https://en.wikipedia.org/wiki/Peephole_optimization)>
6. Dead Code Elimination, Wikipedia <[https://en.wikipedia.org/wiki/Dead\\_code\\_elimination](https://en.wikipedia.org/wiki/Dead_code_elimination)>
7. Constant Folding, Wikipedia <[https://en.wikipedia.org/wiki/Constant\\_folding](https://en.wikipedia.org/wiki/Constant_folding)>
8. Strength Reduction, Wikipedia <[https://en.wikipedia.org/wiki/Strength\\_reduction](https://en.wikipedia.org/wiki/Strength_reduction)>
9. Compiler Writing, What-When-How <<http://what-when-how.com/compiler-writing/machine-dependent-optimization-compiler-writing-part-1/>>



## Q & A SESSION



THANKS FOR LISTENING  
I'LL BE ANSWERING QUESTIONS NOW

