CS3005D Compiler Design

Winter 2024

Lecture #29

Introduction to Code Optimization

Saleena N

CSED NIT Calicut

March 2024

Code Improving Transformations

- Machine Independent Optimizations
  - Code Improving Transformations (done in the intermediate code) to generate better machine code (that runs faster / or takes less space)
- Machine Dependent Optimizations - transformations done in the target code
- Transformation should be *semantics-preserving* - preserve the meaning of the program

Code Improving Transformations

- Analysis of the code to identify possible improvements
- Transform the code

Local Vs. Global Optimization

- Local Optimizations - done in portion of code known as basic blocks - a sequence of instructions such that flow of control always enters the first instruction and exits only after executing the last instruction
- Global Optimization - require program analysis to retrieve the required information

Exercise

Write the 3-address code generated for $x = (a + b) * (a + b)$ and identify possible code improvements.

Array References: 3-address instructions

Indexed copy instructions of the form:

$x = y[i]$ sets x to the value in the location i memory units beyond location y

$x[i] = y$ copies the contents of y to the location i units beyond x

Array References: 3-address code

Assuming one element requires 4 bytes, the expression $a[i]$ is translated to:

$t_1 = i * 4$
$t_2 = a[t_1]$

Write the 3-address code generated for $b = a[i]$

Array References: 3-address code

$b = a[i]$ translated to:

$t_1 = i * 4$
$t_2 = a[t_1]$
$b = t_2$

Write the 3-address code generated for $x = a[i] + b[i]$

Array References: 3-address code

$x = a[i] + b[i]$ translated to:

$t_1 = i * 4$
$t_2 = a[t_1]$
$t_3 = i * 4$
$t_4 = b[t_3]$
$t_5 = t_2 + t_4$
$x = t_5$

Redundant Computation?

Redundant Expression: example

$x = a[i] + b[i]$ translated to:

$t_1 = i * 4$
$t_2 = a[t_1]$
$t_3 = i * 4$
$t_4 = b[t_3]$
$t_5 = t_2 + t_4$
$x = t_5$

Redundant Expression: computation of $i * 4$ in the third instruction is redundant. Value is already computed in the first instruction. Redundancy introduced by the compiler.

Eliminate the redundancy? How?

Common Subexpression Elimination (CSE)

Generated Code	Transformed Code
$t_1 = i * 4$	$t_1 = i * 4$
$t_2 = a[t_1]$	$t_2 = a[t_1]$
$t_3 = i * 4$	$t_3 = t_1$
$t_4 = b[t_3]$	$t_4 = b[t_3]$
$t_5 = t_2 + t_4$	$t_5 = t_2 + t_4$
$x = t_5$	$x = t_5$

Note: Instance of *Local CSE*.
Global CSE to be discussed later.

Further code Improvements?

$t_1 = i * 4$
$t_2 = a[t_1]$
$t_3 = t_1$
$t_4 = b[t_3]$
$t_5 = t_2 + t_4$
$x = t_5$

After the third instruction, t_3 has the same value as t_1
Replace uses of t_3 by t_1?

Copy Propagation

3-address code	Transformed Code
$t_1 = i * 4$	$t_1 = i * 4$
$t_2 = a[t_1]$	$t_2 = a[t_1]$
$t_3 = t_1$	$t_3 = t_1$
$t_4 = b[t_3]$	$t_4 = b[t_1]$
$t_5 = t_2 + t_4$	$t_5 = t_2 + t_4$
$x = t_5$	$x = t_5$

Further Improvements?

3-address code	After Copy Propagation
$t_1 = i * 4$	$t_1 = i * 4$
$t_2 = a[t_1]$	$t_2 = a[t_1]$
$t_3 = t_1$	$t_3 = t_1$
$t_4 = b[t_3]$	$t_4 = b[t_1]$
$t_5 = t_2 + t_4$	$t_5 = t_2 + t_4$
$x = t_5$	$x = t_5$

Further Improvements: If value of t_3 is not used again, remove the statement $t_3 = t_1$

Dead Code Elimination

3-address code	After Copy Propagation
$t_1 = i * 4$	$t_1 = i * 4$
$t_2 = a[t_1]$	$t_2 = a[t_1]$
$t_3 = t_1$	$t_4 = b[t_1]$
$t_4 = b[t_3]$	$t_5 = t_2 + t_4$
$t_5 = t_2 + t_4$	$x = t_5$
$x = t_5$	

If value of t_3 is not used again, the statement $t_3 = t_1$ is dead code which can be eliminated.

Simple Transformations

- Use of algebraic identities

  replace $x + 0$ by x

  replace $x * 1$ by x

- Strength Reduction: Replacing an operation by a less expensive one

  replacing multiplication by shift operation

Constant Propagation

Source code: $i = 5; \quad x = sum + i * 10;$

```
i = 5
t1 = i * 10
t2 = sum + t1
x = t2
```

replace uses of *i* by 5 - eliminates a memory load

```
i = 5
t1 = 5 * 10
t2 = sum + t1
x = t2
```

Constant Folding

Replace an expression that is known to evaluate to a constant[1], by its value

```
i = 5
t1 = 5 * 10
t2 = sum + t1
x = t2
```

Replace `5 * 10` by its value 50

```
i = 5
t1 = 50
t2 = sum + t1
x = t2
```

[1]the same constant value irrespective of the target machine

Dead Code Elimination

```
i = 5
t1 = 50
t2 = sum + t1
x = t2
```

if i is not used again, remove $i = 5$ - reduces code size

```
t1 = 50
t2 = sum + t1
x = t2
```

References

References:

- Aho A.V., Lam M.S., Sethi R., and Ullman J.D. Compilers: Principles, Techniques, and Tools (ALSU). Pearson Education, 2007.

Further reading:

- ALSU Chapter 9