Compiler Construction: Introduction

Lecture 9
Source Code Program Optimizations

Eugene Zouev
Fall Semester 2018
Innopolis University

Before we start…

The schedule

Date	№	The Lecture Topic(s)	Lab Topics	Home Assignments
8 Oct	8	**MIDTERM**	Presentation of project tasks Organizing project teams	
15 Oct	9	Semantic analysis	Consultations on projects: E. Zouev & TAs	**Projects start**
22 Oct	10	Source-level optimizations		Working on projects
29 Oct	11	Virtual Machines		
5 Nov	12	**Backend essentials 1** D. Botcharnikov (Samsung Research)	**Backend essentials 2** D. Botcharnikov (Samsung Research)	
12 Nov	13	Evolution of compiler architecture	Consultations on projects: E. Zouev & TAs	
5-7 Dec	14	FINAL PROJECT PRESENTATIONS	---	---

NB: Lectures on Nov 5 are moved to Nov 3 (the same time)

Program Optimization

Some general points

- Optimization can be performed on each stage of the program lifecycle: not only while compilation but while design, development and maintenance.
- Do we **really need** optimization?
  The best way to optimize a program – is **to design it correctly** (then perhaps we do not need to optimize it ☺)
- "Optimization-in-the-small" vs "optimization-in-the-large" …

Program Optimization

- Finding places in programs which could be optimized (by some criteria) is very much empirical job; in the best case, there is just a set of techniques taken from experience.
- At the same time, there is a number of formal and/or constructive approaches for some kind of optimizations.

Today, we will be discussing what to optimize, but not how to do this…

Program Optimization

- While source code processing (lexical & syntax analysis)

  Big spectrum of optimization techniques.

- While semantic analysis (AST processing).

  Sequential AST traversing.
  Optimizations depend on the language semantics heavily.

- While target code generation (machine-dependent optimizations)

  Depend on the target architecture & on the instruction set.

- While linking: ***global*** code optimizations.

  Example: – C++ code bloat removing.

Elimination of repeated calculations (1)

```
double a = x*(1-sin(y));
double b = x + y/z;
double c = y/z + 1 - sin(y);
```

⇓

```
double tmp1 = 1-sin(y);
double tmp2 = z/y;

double a = x*tmp1;
double b = x + tmp2;
double c = tmp2 + tmp1;
```

The place:
While AST analysis.

Limitations:

1. Factorized functions cannot issue side effects.
2. Operands of factorized expressions cannot modify their values.

Elimination of repeated calculations (2)

```
static int x = 0;
double F(double y)
{
    x++;
    return <expression>;
}
```

Side effect

```
double a = x*(1-F(y));
double b = x + y/z;
...
z = <expression>;
...
double c = z/y + 1 - F(y);
```

Modifying value

Elimination of repeated calculations (3)

An expression may look different but still calculate the same value as some other expression => it can also get optimized.

```
double a = b*c - d;

double e = b;

b = b + 1 - b*c;

double f = b*c + c*e;
```

b*c: the second calculation of the same value.

The second calculation of b*c results in a different value.

The value of c*e is the same as b*c.

Replacing slow instructions for faster ones (1)

Actions: comparative performance

Multiplication/division on a power of two
Addition/subtraction
Multiplication
Division
Calculation of an integer power
Calculation of an arbitrary power

⇒ Replacing slower operations for faster ones (where possible)

For some target architectures it's **mandatory**: e.g., some RISCs just do not support multiplication!

Replacing slow instructions for faster ones (2)

Multiplication		Addition
$X*K$	$\Rightarrow$	$\underbrace{X+X+...+X}_{K}$

K is a small integer constant

In general case it's impossible...

- At least one operand must be an integer constant.
- The constant should be relatively small; otherwise rounding errors will accumulate.

Replacing slow instructions for faster ones (3)

Division

X/K

Multiplication

X*(1/K)

K - constant

```
double x = c/b;
double y = (e+f)/b + d;
double z = b;
b = b+1;
...
z = sin(x)/z + e/b;
```

```
double tmp = (double)1/b;
double x = c*tmp;
double y = (e+f)*tmp + d;
double z = b;
b = b+1;
...
z = sin(x)*tmp + e/b;
```

Excluding redundant calculations

```
double a;
...
a = (x+y)*sin(z);
...
a = x/y;
```

```
double a;
...
a = x/y;
```

If the value of a does not change between two assignments, then the first assignment can be removed.

Limitation: the action being removed cannot make side effects.

Constant propagation (1)

If the value of a variable is known
then the variable reference could be
replaced for the value itself.

```
double a = 2.0;
double b = 3.5;
...
double c = a*b;
...
double t = (b+c)*a+x;
```

⟹

```
double a = 2.0;
double b = 3.5;
...
double c = 7.0; // a*b
...
double t = 10.5 + x; // (b+c)*a
```

Constant propagation (2)

If the value in a loop condition is known in advance then the loop could be simplified.

```
while ( <Expression> )
{
    <Statements>
}
```

⇒

```
Loop:
  if ( !<Expression> ) goto Exit;
  <Statements>
  goto Loop;
Exit:
  ;
```

```
while ( 1 )
{
    <Statements>
}
```

⇒

```
Loop:
  if ( !<Expression> ) goto Exit;
  <Statements>
  goto Loop;
Exit:
  ;
```

Type conversion optimizations

Type conversion is a potentially costly operation; therefore it's a good candidate for optimizations.

```
double a, b;
long i, j;
...
double c = a + i + b - j;
```

```
... a + (double)i + b - (double)j ...
```

```
double a, b;
long i, j;
...
double c = (a+b) + (i-j);
```

```
... a+b + (double)(i-j) ...
```

Reducing calls & indexings

Access to array elements and function calls are also good candidates for optimizations...

```
for i:integer range 1..100 loop
    x(i) = y(i)+1/y(i);
    z(i) = y(i)**2;
end loop;
```

Address of y(i) gets calculated **300 times**

```
for i:integer range 1..100 loop
    declare
        tmp : real := y(i);
    begin
        x(i) = tmp+1/tmp;
        z(i) = tmp**2;
    end;
end loop;
```

Address of y(i) gets calculated **100 times**

Merging loops (1)

Loops are main consumers of CPU time!

```
for i:integer range 1..100 loop
    x(i) = 0;
end loop;
```

```
for i:integer range 1..100 loop
    z(i) = y(i)**2;
end loop;
```

```
for i:integer range 1..100 loop
    x(i) = 0;
    z(i) = y(i)**2;
end loop;
```

Costs for loop organization are reduced

Merging loops (2)

```
for i:integer range 1..100 loop
    x(i) = 0;
end loop;
```

```
for i:integer range 1..200 loop
    z(i) = y(i)**2;
end loop;
```

(More general case)

Overall amount of iterations: **300**

$\Downarrow$

```
for i:integer range 1..100 loop
    x(i) = 0;
    z(i) = y(i)**2;
end loop;
```

```
for i:integer range 101..200 loop
    z(i) = y(i)**2;
end loop;
```

Overall amount of iterations: **200**

Unrolling loops (1)

```
for (int i=0; i<100; i++)
{
    x[i] = y[i]*z[i];
}
```

Loop step = **1**
Overall amount
of iterations: **100**

```
for (int i=0; i<100; i+=2)
{
    x[i] = y[i]*z[i];
    x[i+1] = y[i+1]*z[i+1];
}
```

Loop step = **2**
Overall amount
of iterations: **50**

Unrolling loops (2)

```
-- Skip past blanks, loop is opened up for speed
while Source (Scan_Ptr) = ' ' loop
    if Source (Scan_Ptr + 1) /= ' ' then
        Scan_Ptr := Scan_Ptr + 1; exit;
    end if;
    if Source (Scan_Ptr + 2) /= ' ' then
        Scan_Ptr := Scan_Ptr + 2; exit;
    end if;
    if Source (Scan_Ptr + 3) /= ' ' then
        Scan_Ptr := Scan_Ptr + 3;  exit;
    end if;
    if Source (Scan_Ptr + 4) /= ' ' then
        Scan_Ptr := Scan_Ptr + 4; exit;
    end if;
    if Source (Scan_Ptr + 5) /= ' ' then
        Scan_Ptr := Scan_Ptr + 5; exit;
    end if;
    if Source (Scan_Ptr + 6) /= ' ' then
        Scan_Ptr := Scan_Ptr + 6; exit;
    end if;
    if Source (Scan_Ptr + 7) /= ' ' then
        Scan_Ptr := Scan_Ptr + 7; exit;
    end if;
    Scan_Ptr := Scan_Ptr + 8;
end loop;
```

A real example:
The scanner of the
Ada GNAT compiler

Stack optimizations (1)

```
class Program
{
  int F(int a,int b)
  {
    int c = 7;
    int x = (a-b)*(a+c);
    return x;
  }
}
```

```
...
IL_0001:  ldc.i4.7
IL_0002:  stloc.0
IL_0003:  ldarg.1
IL_0004:  ldarg.2
IL_0005:  sub
IL_0006:  ldarg.1
IL_0007:  ldloc.0      <==  ldc.i4.7
IL_0008:  add
IL_0009:  mul
IL_000a:  stloc.1
...
```

Even such a simple code **is not optimal**;
it could be improved.

The first improvement:
constant propagation

Stack optimizations (2)

```
class Program
{
  int F(int a,int b)
  {
    int c = 7;
    int x = (a-b)*(a+c);
    return x;
  }
}
```

```
...
IL_0001:  ldc.i4.7
IL_0002:  stloc.0
IL_0003:  ldarg.1
IL_0004:  ldarg.2
IL_0005:  sub
IL_0006:  ldarg.1
IL_0007:  ldloc.0
IL_0008:  add
IL_0009:  mul
IL_000a:  stloc.1
...
```

The value of a gets pushed **two times**

```
...
IL_0003:  ldarg.1
          dup
IL_0004:  ldarg.2
IL_0005:  sub
          swap
IL_0006:  ldarg.1
IL_0007:  ldloc.0
IL_0008:  add
IL_0009:  mul
IL_000a:  stloc.1
...
```

The second improvement:
Replace the second push instruction for faster instruction(s).

dup Copies the value on the top of the stack

swap Swaps positions of two stack topmost operands (exists in JVM, but **doesn't** in MSIL)

VM: stack- or register-based?

```
VAR
    ch: CHAR;
    lint: LONGINT;
BEGIN
    lint := LONG(ORD(ch));
    ...
```

```
MOVZBD ch, lint
   // move, zero-extending
   // byte to double-word
```

National Semiconductor 32000 family

```
LOAD1 ch     // move single_byte CHAR value onto stack
ORD          // zero-extend, result is an INTEGER
LONG         // sign-extend, result is a LONGINT
STORE4 lint  // move four_byte LONGINT value back to memory
```

An abstract stack computer

Michael Franz

Code-Generation On-the-Fly: A Key to Portable Software

Dissertation ETH No. 10947, 1994

Tail Recursion (1)

```
void f(int x)
{
    if ( x == 0 ) return;
    ...Some actions...
    f(x-1);
}
```

```
void f(int x)
{
    if ( x == 0 ) return;
    ... Some actions...
    if ( Some_condition )
        f(x-1);
    else
        f(x-2);
}
```

The idea:
If the recursive call is the very last operation in the function body then it could be replaced for the direct jump to the beginning of the body – perhaps with argument (re)initialization.

See more details in:
http://en.wikipedia.org/wiki/Tail_call

Tail Recursion (2)

```
long factorial(long n)
{
   if (n == 0) return 1;
   else return n*factorial(n-1);
}
```

This is **not** tail recursion. **Why**?

```
int fac_times(int n, int acc)
{
    if (n == 0) return acc;
    else return fac_times(n-1,acc*n);
}

int factorial(int n)
{
    return fac_times(n,1);
}
```

Equivalent program **with** tail recursion.