



NTNU – Trondheim
Norwegian University of
Science and Technology

Introduction to optimizations

Transformations to improve program performance

- This topic is scattered around a few different subchapters in the book
 - Some are most easily applied to high-level IR
 - Others are simpler at low-level
- I'm collecting them under a single heading to give a context for the analysis methods we're about to cover
 - Many optimizations require combinations of different analysis results
 - If you can keep them at the back of your mind, it's easier to see what the analyses are for

A number of possible tricks

- Function inlining
- Function cloning
- Constant folding
- Constant propagation
- Unreachable/dead code elimination
- Loop-invariant code motion
- Common sub-expression elimination
- Strength reduction
- Loop unrolling



Function inlining

- A function like

```
int sumsq ( x, y ) { return (x*x)+(y*y); }
```

makes the call

```
z = sumsq ( a, b );
```

equivalent to

```
z = (a*a)+(b*b);
```

- This saves a function call
 - Altered control flow + memory interactions for stack frame
- Generated code size grows with the number of inlined function instances
 - Repeated generation of same instruction sequence



(As an aside)

- Both C and C++ have an `inline` keyword for functions, in support of this transformation
 - In slightly different ways, these work as programmer-provided suggestions that the compiler should consider a function for inlining
 - Whether or not they *are* inlined becomes subject to a performance estimate at the compiler's discretion
 - This is great, except for when it needs to behave predictably across different compilers
- Inlining can be *forced* with a macro definition

```
#define SUMSQ(x,y) ((x)*(x)+(y)*(y))
```

(at the cost of some type safety, and the benefit of the compiler's analysis)
 - The exercises may have revealed that I'm a habitual macro abuser
 - For better or worse, my reason for that is the predictability thing
 - Consider it a work-related injury if you will, excessive preprocessor use is not pretty software engineering

Function cloning

- If we can establish that the arguments frequently have the value 1, the same function

```
int sumsq ( x, y ) { return (x*x)+(y*y); }
```

could be generated in multiple versions

```
int sumsq_x_eq_1 ( y ) { return (y*y)+1; }
```

```
int sumsq_y_eq_1 ( x ) { return (x*x)+1; }
```

```
int sumsq ( x, y ) {
```

```
    if ( x==1 ) return sumsq_x_eq_1 ( y );
```

```
    else if ( y==1 ) return sumsq_y_eq_1 ( x );
```

```
    else return (x*x)+(y*y);
```

```
}
```

- When the work saved in the appropriate clone outweighs the overhead of the inserted code to select it at run-time, this is an optimization



Function cloning in action

- Without having to predict values, one use of this you may spot in the wild is
 - Generate a variety of implementations which target various specific CPU instruction set extensions (vector operations, fused multiply-accumulate instructions, ...)
 - Inject run-time code to identify the specific CPU model in use
 - Branch to the appropriate version of the function
- This creates portable code by default, and is usually complemented with the option to generate code for one specific instruction set (saving the overhead)
 - In case you're sure that your program will only ever run on, say, AVX2-capable processors

Constant propagation

- If the value of a variable is known to be constant, its uses can be replaced by the constant value

```
n = 10
```

```
c = 2
```

```
for ( i = 0; i < n; i++ ) { s = s + i * c; }
```

becomes

```
for ( i=0; i<10; i++ ) { s = s + i * 2; }
```

- Named constants can appear for readability reasons, maintaining a single place to modify a constant used in many places, *etc.*



Constant folding

- We did some of this when simplifying VSL trees:

```
x = 1.1 * 2;
```

becomes

```
x = 2.2;
```

Constant expressions appear for several reasons:

- “n_elements * sizeof(element_t)” reads more easily than “22*12”
- “2*PI” is clearer than “2 * 3.1415928...” is clearer than “6.283185...”
- Translations and optimizations can create them

```
int x = a[2] → t1 = 2*4  
              t2 = a + t1  
              x = *t2;
```

Fancier constant folding

- Algebra can be simplified in a number of obvious ways:

$$x * 1 = x \quad x * 0 = 0$$

$$x / 1 = x \quad x + 0 = x$$

$$x \parallel \text{false} = x \quad x \&\& \text{true} = x$$

etc. etc.

- Repeated application can simplify expressions away

$$a = 1; b=0; h = 1;$$

$$(a*x + b*y) / (h*h)$$

$$\rightarrow (1*x + 0*y) / (1*1)$$

$$\rightarrow (x + 0) / 1$$

$$\rightarrow x$$

(NB: this can be risky business with floating point numbers)



Copy propagation

- After $x=y$, y can be used instead of x until x is assigned differently

```
x = y;
```

```
if ( x > 1 ) { s = x * f ( x-1 ); }
```

becomes

```
x = y;
```

```
if ( y > 1 ) { s = y * f ( y-1 ); }
```

- Repeated application gives further benefit
 - If there was a “ $y = z$ ” before, z could be replaced instead
 - Fewer variables reduce pressure on the use of a limited number of registers



Common subexpression elimination

- If a program computes the same intermediate value several times, the value can be re-used:

```
a = (b+c) * d
```

```
c = b + c
```

can be re-written as

```
temp = b+c
```

```
a = temp * d
```

```
c = temp
```

- Common subexpressions can occur as side-effects of translation

```
a[i] = b[i] + 1
```

is liable to generate the same offset-calculation for “[i]” twice, if a and b are same type

Unreachable code elimination

- It can be useful to insert code that never runs under particular compile-time conditions:

```
#define DEBUG false
```

```
...
```

```
s = 1;
```

```
if ( DEBUG )
```

```
    printf ( "s = %d", s );
```

translates to "s=1;" when you don't care for the output

(Unreachable code can be hard to detect in low-IR, where control flow is reduced to jumps and labels)



Dead vs. unreachable

- Statements can also be eliminated if their effects are never seen

$x = y + 1$

$y = 1$

$x = 2 * z$

becomes

$y = 1$

$x = 2 * z$

because the $y + 1$ value of x is never used (it's "dead")

- Dead code may appear as a side-effect of translation, and/or other optimizations



Loop-invariant code motion

- Code that repeats the same computation inside a loop can be moved out of the iteration:

```
for ( i=0; i<360; i++ )  
    angle_rad = i * ( PI / 180.0 )
```

becomes

```
temp = PI / 180.0  
for ( i=0; i<360; i++ )  
    angle_rad = i * temp
```

- Invariant code can only be moved if it has no visible side-effect
 - Moving a print statement won't do, even if its values are the same every iteration



Strength reduction

- Replace expensive operations with cheaper ones

```
for ( i=0; i<n; i++ ) {  
    v = 8 * i;  
    sum += v;  
}
```

can be written

```
v = -8;  
for ( i=0; i<n; i++ ) {  
    v += 8;  
    sum += v;  
}
```

to replace multiplication by addition



Strength reduction

- If you take it one step further, the induction variable i can be removed altogether:

```
v = -8;
for ( i=0; i<n; i++ ) {
    v += 8;
    sum += v;
}
```

can be written

```
v = -8;
for ( ; v < (n-1)*8; ) {
    v += 8;
    sum += v;
}
```



Strength reduction

- There are a bunch of equivalences for various frequently used operation/value combinations

$$x * 2 = x+x$$

$$x * 2 = (x \ll 1) \quad (\text{for integers})$$

$$x * 2^c = (x \ll c) \quad \dots$$

$$x / 2^c = (x \gg c) \quad \dots$$

- Whether a particular replacement actually saves any time is architecture-dependent, and merits measurement



Loop unrolling

- Run loop body multiple times per iteration:
for (i=0; i<n; i++) { S; }
unrolled 4 times becomes
for (i=0; i<n; i+=4) { S₀; S₁; S₂; S₃; }
(with substitutions of 'i+1', 'i+2', 'i+3' for i in copies 1-3)
- Pro: Computation workload is the same, but $\frac{3}{4}$ fewer conditional branch instructions
- Con: Loop body code grows bigger
 - ...and needs care when n is not a multiple of 4...

The importance of loops

- Program hotspots are often loops
 - Most execution time is spent doing repetitive tasks
- Loop optimizations multiply any gain of the optimization by the iteration count

The safety of optimizing

- It's best when you can rely on the compiler to implement these maneuvers
 - They make a mess of tidy source programs
- The compiler has to be conservative when applying optimizations
 - *E.g.*, it can not take a value to be constant unless the language semantics absolutely guarantee it
 - The programmer knows what the program is meant to do, but may overlook potential interpretations that ruin automatic tuning
- Part of the value of studying compilers is to notice it when they can't help you do what you had in mind
 - When it's possible, you can rework the program so that the compiler sees what you want
 - When it's not, you can transform the program yourself (trading readability for speed only where it counts)



Going forward

- There are many ways to boost the efficiency of a program
- The whole is greater than the sum of parts
 - Optimizations interact
 - Optimizations can be applied several times
 - Optimizations can work at different levels of abstraction
- Problem:
When can we automatically detect that they are safe?
- That's the backdrop for the last chunk of our syllabus



An elephant in the room

- The transformations we look at trade operations and control flow constructs for each other
- I've alluded a few times to the observation that data movement is at least equally important for program performance
- Automatic recognition of data movement tuning is an open research topic
 - We don't cover it much because contemporary compilers are frankly not very good at it
 - That's well worth being aware of, we'll return to it in the end

