COMPUTER ORGANIZATION AND SOFTWARE SYSTEMS

COMPILER OPTIMIZATION

BITS Pilani
Pilani Campus

Presented By: Darshitha

Introduction

Compiler Optimization

- algorithms which take a program and transform it to an equivalent output program that uses fewer resources.
- minimizing program executing time
- minimizing memory use
- minimizing the power consumed by a program.

Introduction

The default is optimization off. This results in the fastest compile times, but compiler makes absolutely no attempt to optimize, and the generated programs are considerably larger and slower than when optimization is enabled.

Without any optimization option, the compiler's goal is to reduce the cost of compilation and to make debugging produce the expected results. Statements are independent:

Turning on optimization makes the compiler attempt to improve the performance and/or code size at the expense of compilation time and possibly the ability to debug the program.

Compiler Optimization Levels

4 numeric levels

- On where n ranges from 0 (no optimization) to 3.
  - O0 - No optimization (the default), generates un optimized code but has the fastest compilation time.
  - Many compilers do substantial optimization even if no optimization is specified. O0 means (almost) no optimization.

-O1 Optimization

- Minimal impact on compilation time as compared to O0 compile.
- Only optimizations applied to straight line code (basic blocks) like instruction scheduling.

-O2 Optimization

- Default when no optimization arguments given.
- Optimizations that always increase performance.
- Can significantly increase compilation time.
- O2 optimization examples:
  - Loop nest optimization.
  - Global optimization within a function scope.
  - 2 passes of instruction scheduling.
  - Dead code elimination.
  - Global register allocation.

-O3 Optimization

- More extensive optimizations that may in some cases slow down performance.
- Optimizes loop nests rather than just inner loops, i.e. inverts indices, etc.
- "Safe" optimizations – produces answers identical with those produced by – O0.

Compiler Optimizations

Machine independent (apply equally well to most CPUs)

- Constant propagation
- Constant folding
- Common Sub expression Elimination
- Dead Code Elimination
- Loop Invariant Code Motion
- Function In lining

Machine dependent (apply differently to different CPUs)

- Instruction Scheduling
- Loop unrolling
- Parallel unrolling

Could do these manually, better if compiler does them

- Many optimizations make code less readable/maintainable

Constant Propagation (CP)

```
a = 5;
b = 3;
   :
   :
n = a + b;
for (i = 0 ; i < n ; ++i) {
        :
        :
}
```

Replace variables with constants when possible

Constant Folding (CF)

:
:
:
:

~~n = 5 + 3;~~ $\rightarrow$ n = 8

for (i = 0 ; i < n ; ++i) {

:

}

$\searrow$ 8

- Evaluate expressions containing constants
- Can lead to further optimization
    - Eg., another round of constant propagation

Dead Code Elimination (DCE)

```
debug = 0;  // set to False
     :
if (debug) {
     :
     :
}
a = f(b);
```

→

```
debug = 0;
     :
     :
a = f(b);
```

- Compiler can determine if certain code will never execute:
  - Compiler will remove that code
  - You don't have to worry about such code impacting performance
    - i.e., you are more free to have readable/debugable programs!

Common Sub-expression Elimination (CSE)

```
a = c * d;
      :
      :
d = (c * d + t) * u
```

→

```
a = c * d;
      :
d = (a + t) * u
```

Try to only compute a given expression once

(assuming the variables have not been modified)

Loop Invariant Code Motion (LICM)

```
for (i=0; i < 100 ;  ++i) {
          a =5;
  for (j=0; j < 100 ; ++j) {
          b = 6;
    for (k=0 ; k < 100 ; ++k) {
      x= i+j+k;
    }
  }
}
```

$\Rightarrow$

```
a = 5;
b = 6;
for (i = 0; i < 100 ; ++i) {
  for (j = 0; j < 100 ; ++j) {
   for (k = 0 ; k < 100 ; ++k) {
         x= i+j+k;
  }
 }
}
```

- Loop invariant: value does not change across iterations
- LICM: move invariant code out of the loop
- Big performance wins:
  - Inner loop will execute 1,000,000 times
  - Moving code out of inner loop results in big savings

Function Inlining

```
foo(int z){
  int m = 5;
  return z + m;
}
main(){
   ...
   x = foo(x);
   ...
 }
```

```
main(){
 ...
 {
  int foo_z = x;
  int m = 5;
  int foo_return = foo_z + m;
  x = foo_return;
 }
 ...
}
```

```
main(){
  ...
  x = x + 5;
  ...
}
```

Code size
- can decrease if small procedure body and few calls
- can increase if big procedure body and many calls

Performance
- eliminates call/return overhead
- can expose potential optimizations
- can be hard on instruction-cache if many copies made

As a Programmer:
- a good compiler should inline for best performance
- feel free to use procedure calls to make your code readable!

Loop Unrolling

```
j = 0;
while (j < 100){
    a[j] = b[j+1];
    j += 1;
}
```

```
j = 0;
while (j < 99){
    a[j] = b[j+1];
    a[j+1] = b[j+2];
    j += 2;
}
```

reduces loop overhead

- Fewer adds to update j
- Fewer loop condition tests

enables more aggressive instruction scheduling

- more instructions for scheduler to move around

Example Compiler: gcc Optimization Levels

-g:
- Include debug information, no optimization

-O0:
- Default, no optimization

-O1:
- Do optimizations that don't take too long
- CP, CF, CSE, DCE, LICM, inlining small functions

-O2:
- Take longer optimizing, more aggressive scheduling

-O3:
- Make space/speed trade-offs: loop unrolling, more inlining

-Os:
- Optimize program size

Performance Optimization: Requirements

1) Preserve correctness
   - the speed of an incorrect program is irrelevant
2) On average improve performance
   - Optimized may be worse than original if unlucky
3) Be "worth the effort"
   - Is this example worth it?
     - 1 person-year of work to implement compiler optimization
     - 2x increase in compilation time
     - 0.1% improvement in speed

How do optimizations improve performance?

Execution_time = num_instructions * CPI

- Fewer cycles per instruction
  - Schedule instructions to avoid dependencies
  - Improve cache/memory behavior
    - Eg., locality
- Fewer instructions
  - Eg: Target special/new instructions

Role of Optimizing Compilers

Provide efficient mapping of program to machine

- eliminating minor inefficiencies
- code selection and ordering
- register allocation

Don't (usually) improve asymptotic efficiency

- up to programmer to select best overall algorithm
- big-O savings are (often) more important than constant factors
    - but constant factors also matter

Limitations of Optimizing Compilers

- Operate Under Fundamental Constraints
  - Must not cause any change in program behavior under any possible condition
- Most analysis is performed only within procedures
  - inter-procedural analysis is too expensive in most cases
- Most analysis is based only on *static* information
  - compiler has difficulty anticipating run-time inputs
- When in doubt, the compiler must be conservative

Role of the Programmer

How should a programmer write programs, given that there is a good, optimizing compiler?

Don't: Smash Code into Oblivion

- Hard to read, maintain, & assure correctness

Do:

- Select best algorithm
- Write code that's readable & maintainable
  - Procedures, recursion
  - Even though these factors can slow down code
- Eliminate optimization blockers
  - Allows compiler to do its job

Focus on Inner Loops

- Do detailed optimizations where code will be executed repeatedly
- Will get most performance gain here

Questions ?

BITS Pilani
Pilani Campus

Thank you.