ECE 454
Computer Systems Programming

Compiler Optimizations

Ashvin Goel
ECE Dept, University of Toronto
http://www.eecg.toronto.edu/~ashvin
<table>
<thead>
<tr>
<th>Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>• History and overview of compilers</td>
</tr>
<tr>
<td>• Basic compiler optimizations</td>
</tr>
<tr>
<td>• Program optimizations</td>
</tr>
<tr>
<td>• Advanced optimizations</td>
</tr>
<tr>
<td>• Parallel unrolling</td>
</tr>
<tr>
<td>• Profile-directed feedback</td>
</tr>
</tbody>
</table>
A Brief History of Compilation
In the Beginning…

Programmers wrote machine instructions
Then Came the Assembler

Programmers wrote human-readable assembly

add r3,r3,r1
cmp r3,r1
bge 0x3340a
mulu r3,r5,r2
sub r1,r3,r4
...

Assembler

1010010010
0101101010
1010010100
1010001010
...

Processor
Then Came the Compiler

Programmers wrote high-level language (HLL)

```
int Foo(int x){
    return x+5;
}
...
```

```
add r3,r3,r1
cmp r3,r1
bge 0x3340a
mulu r3,r5,r2
sub r1,r3,r4
...
```

1010010010
0101101010
1010010100
1010001010
...

Processor
Overview of Compilers
Goals of a Compiler

• Correct program executes correctly
• Provide support for debugging incorrect programs
• Program executes fast
• Compilation is fast?
• Small code size?
• More energy efficient program?
Inside a Basic Compiler

CSC488 Compilers and Interpreters

High-level language
(C, C++, Java)

Front End

Code Generator

Low-level language
(IA64)

Intermediate Representation
(similar to assembly)

HLL
IR
LLL
Control Flow Graph:
(how a compiler sees your program)

Example IR:
```
add ...
L1: add ...
    add ...
    branch L2
    add ...
L2: add ...
    branch L1
    return ...
```

Basic Blocks:
```
add ...
L1: add ...
    add ...
    branch L2
    add ...
L2: add ...
    branch L1
    return ...
```

Basic Block: a group of consecutive instructions with a single entry point and a single exit point
Data Flow Analysis

• Many compiler optimizations (discussed later) use a technique called data flow analysis

• Basic idea
  • Analyse and summarize the effects of instructions in a basic block
  • Use CFG to propagate these effects to succeeding basic blocks

• E.g., reaching definition data flow analysis
  • Calculates for each program point the set of definitions (program points) that may potentially reach this program point

```c
1: if b==4 then      // BB1
2:   a = 5;         // BB2
3: else              
4:   a = 3;         // BB3
5: endif             
6: if a < 4 then ... // BB4
```
Inside an Optimizing Compiler

CSCD70/ ECE540 Optimizing Compilers

High-level language (C, C++, Java)

Front End → Optimizer → Code Generator

HLL → IR (Improved) → LLL

Low-level language (IA64)
Performance Optimization: 3 Requirements

- **Preserve correctness**
  - The speed of an incorrect program is irrelevant

- **Improve performance of average case**
  - Optimized program may be worse than original if unlucky

- **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?

• Recall

\[ \text{Execution\_time} = \text{num\_instructions} \times \text{CPI} \times \text{time/cycle} \]

• Fewer instructions
  • Use optimized sequence of instructions
  • Use new instructions

• Fewer cycles per instruction
  • Schedule instructions to avoid hazards
  • Improve cache/memory behavior
    • E.g., prefetching, code and data locality
Role of Optimizing Compilers

- Provide efficient mapping of program to machine instructions
  - Eliminate minor inefficiencies
  - Register allocation
  - Instruction selection
  - Instruction scheduling

- 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 always be conservative
Role of the Programmer

- How should I write my programs, given that I have a good, optimizing compiler?
- Don’t: smash code into oblivion
  - Hard to read, maintain, assure correctness
Role of the Programmer

• How should I write my programs, given that I have a good, optimizing compiler?

• Do:
  • Select best algorithm
  • Write code that’s readable and maintainable
    • Procedures, recursion
    • Even though these may slow down code
  • Focus on inner loops
    • Do detailed optimizations where code will be executed repeatedly
    • Will get most performance gain here
  • Eliminate optimization blockers
    • Allows compiler to do its job!
Basic Compiler Optimizations
Compiler Optimizations

- **Machine independent (apply equally well to most CPUs)**
  - Constant propagation
  - Constant folding
  - Copy propagation
  - Common subexpression elimination
  - Dead code elimination
  - Loop invariant code motion
  - Function inlining
Compiler Optimizations

- Machine dependent (apply differently to different CPUs)
  - Instruction selection and scheduling
  - Loop unrolling
  - Parallel unrolling
- Possible to do all these optimizations manually, but much better if compiler does them
  - Many optimizations make code less readable/maintainable
Constant Propagation (CP)

- Replace variables with constants when possible

```c
a = 5;
b = 3;

n = a + b;
for (i = 0; i < n; ++i) {
    ...
}
```

```
22
n = 5 + 3
```
Constant Folding (CF)

- Evaluate expressions containing constants

\[
\begin{align*}
\text{n} & = 5 + 3; \\
\text{for} \ (i = 0; i < \text{n}; ++i) \{ \\
\text{\quad} \text{:\text{}} \\
\text{\} }
\end{align*}
\]

- Can lead to further optimization
  - E.g., another round of constant propagation
Common Sub-Expression Elimination (CSE)

- Try to only compute a given expression once

\[
\begin{align*}
  a &= c \times d; \\
  \vdots & \quad \Rightarrow \\
  \vdots & \\
  d &= (c \times d + t) \times u
\end{align*}
\]

\[
\begin{align*}
  a &= c \times d; \\
  \vdots & \\
  \vdots & \\
  d &= (a + t) \times u
\end{align*}
\]

- Need to ensure the variables have not been modified
Copy Propagation

- Replace target of assignment with corresponding value
  
  ```
  y = x;
  z = 3 + y;
  ```

- Often used after common sub-expression elimination and other optimizations
Dead Code Elimination (DCE)

- Compiler can determine if certain code will never execute

```
debug = 0;  // set to false
default {
    
    a = f(b);  
```

- Compiler will remove that code
  - You don’t have to worry about such code impacting performance
  - Makes it easier to have readable/debugable programs
Loop Invariant Code Motion (LICM)

• Loop invariant: value does not change across iterations
• LICM: moves invariant code out of the loop
• Leads to significant performance win
Loop Invariant Code Motion (LICM)

- Consider this triply nested loop

```c
for (i=0; i < I; ++i) {
    for (j=0; j < J; ++j) {
        for (k=0; k < K; ++k) {
            a[i][j][k] = i*j*k;
        }
    }
}
```

- In C, a multi-dimensional array is stored in row-major order

```
char a[I][J][K];

addr of a[i][j][k] = (addr of a) + (i * J * K) + (j * K) + (k)
```
Loop Invariant Code Motion (LICM)

\[
\text{addr of } a[i][j][k] = (\text{addr of } a) + (i \times J \times K) + (j \times K) + (k)
\]

for (i=0; i < I; ++i) {
    for (j=0; j < J; ++j) {
        for (k=0; k < K; ++k) {
            \[ a[i][j][k] = i \times j \times k; \]
        }
    }
}

\[ \Rightarrow \]

for (i = 0; i < I; ++i) {
    t1 = a + i \times J \times K; // t1=a[i];
    for (j = 0; j < J; ++j) {
        t2 = t1 + j \times K; // t2=t1[j];
        for (k = 0; k < K; ++k) {
            t2[k] = i \times j \times k;
        }
    }
}
Loop Invariant Code Motion (LICM)

- When I=J=K=100, inner loop will execute 1,000,000 times
  - Many of the computations in the inner loop are moved out
  - Improves performance dramatically

```c
for (i=0; i < I; ++i) {
    for (j=0; j < J; ++j) {
        for (k=0 ; k < K; ++k) {
            a[i][j][k] = i*j*k;
        }
    }
}

for (i = 0; i < I; ++i) {
    t1 = a + i * J * K; // t1=a[i];
    for (j = 0; j < J; ++j) {
        t2 = t1 + j * K; // t2=t1[j];
        tmp = i * j;
        for (k = 0 ; k < K; ++k) {
            t2[k] = tmp * k;
        }
    }
}
```

\[
\text{addr of } a[i][j][k] = (\text{addr of } a) + (i \times J \times K) + (j \times K) + (k)
\]
Function Inlining

- A function call site is replaced with the body of the function.

```c
main()
{
    ...
    x = foo(x);
    ...
}

foo(int z) {
    int m = 5;
    return z + m;
}

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

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

• **Performance**
  - Eliminates call/return overhead
  - Can expose potential optimizations
  - Can be hard on instruction-cache if many copies made
    • Code size can increase if large procedure body and many calls

• **As a programmer**
  - A good compiler should inline for best performance
  - Feel free to use procedure calls to make your code readable!
Loop Unrolling

- Reduces loop overhead, why?
  - Fewer adds to update j
  - Fewer loop condition tests
  - Reduces branch penalties

- Enables more aggressive instruction scheduling
  - I.e., more instructions in loop basic block for scheduler to move around

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

```c
j = 0;
while (j < 99){
    a[j] = b[j+1];
    a[j+1] = b[j+2];
    j += 2;
}
```
Summary: 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, inline functions called once
- **-O2**: Take longer optimizing, more aggressive scheduling
  - E.g., inline small functions
- **-O3**: Make space/speed trade-offs
  - Can increase code size, loop unrolling, more inlining
- **-Os**: Optimize program size