ECE 454
Computer Systems Programming

Compiler Optimizations

Ashvin Goel
ECE Dept, University of Toronto
http://www.eecg.toronto.edu/~ashvin
Content

- History and overview of compilers
- Basic compiler optimizations
- Program optimizations
- Advanced optimizations
  - Parallel unrolling
  - Profile-directed feedback
A Brief History of Compilation
Programmers wrote machine instructions
Then Came the Assembler

Programmer

Add r3,r3,r1
Cmp r3,r1
Bge 0x3340a
Mulu r3,r5,r2
Sub r1,r3,r4
...

Assembler

Machine Instructions

1010010010
0101101010
1010010100
1010010100
...

Processor

 programma wrote human-readable assembly
Then Came the Compiler

Programmer

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

Compiler

Assembly

```
Add r3,r3,r1
Cmp r3,r1
Bge 0x3340a
Mulu r3,r5,r2
Sub r1,r3,r4
...
```

Machine Instructions

```
1010010010
0101101010
1010010100
1010001010
...
```

Processor

Programmers wrote high-level language (HLL)
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)

HLL

Intermediate Representation
(similar to assembly)

IR

Low-level language
(IA64)

LLL

Front End

Code Generator
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
Inside an Optimizing Compiler

CSCD70/ ECE540 Optimizing Compilers

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

Front End

HLL

Optimizer

IR

(IR (Improved))

Code Generator

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
  • E.g., use special/new instructions

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

- Provide efficient mapping of program to machine
  - 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

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

- Evaluate expressions containing constants

```java
n = 5 + 3;
for (i = 0 ; i < n ; ++i) {
    
}
```

- 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

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

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

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

- Replace right-hand side 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

```java
debug = 0; // set to false
debug = 0;
if (debug) {
    a = f(b);
}  \Rightarrow
```

• 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

<table>
<thead>
<tr>
<th>a[0][0][0]</th>
<th>a[0][0][1]</th>
<th>...</th>
<th>a[0][0][K-1]</th>
<th>a[0][1][0]</th>
<th>...</th>
<th>a[I-1][J-1][0]</th>
<th>..</th>
<th>a[I-1][J-1][K-1]</th>
</tr>
</thead>
</table>

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

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

addr of a[i][j][k] = (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;
        }
    }
}

⇒

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)

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

\begin{verbatim}
for (i=0; i < I; ++i) {
    for (j=0; j < J; ++j) {
        for (k=0; k < K; ++k) {
            \texttt{a[i][j][k]} = \texttt{i*j*k;}
        }
    }
}
\end{verbatim}

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

• Inner loop will execute 1,000,000 times
  • Many of the computations in the inner loop are moved out
  • Improves performance dramatically
Function Inlining

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

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

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