COMPILER OPTIMIZATIONS

- Optimizing compilers typically
  - Try to implement language abstractions efficiently
  - Try to utilize hardware resources efficiently
- Different targets may be generated for different compilation goals
  - Improved code speed
  - Reduced code size
  - Improved programmer productivity
    - Feedback to the users; debugging support
  - Shortened compilation time
  - ...

- Feedback to the users; debugging support
MODERN COMPILERS

- Applications keep changing
  - Interactive
  - real-time
  - mobile

- Desired target properties keep changing
  - Code size
  - running time
  - power consumption
  - security

COMPILED STRUCTURE

- Front-end: source program to intermediate code
  - Analyze syntax and semantics of the input program

- Back-end: intermediate code to target program
  - Transform and generate the output program
TRANSLATING LANGUAGE ABSTRACTIONS

- Aggregated data structure
  - Arrays, structures
  - Symbol table
  - Address calculation for references
- Control flows
  - If-then-else, for-loop, while-loop, switch-case
  - Labels and simple branches
- Procedures
  - Activation records
  - Run-time stack
- Objects
  - Object representation
  - Method dispatching

CODE GENERATION

- Intermediate representation $\rightarrow$ target program
- Important issues
  - Instruction selection
  - Instruction scheduling
  - Register allocation
- Often involves machine-dependent optimizations
CODE OPTIMIZATIONS

- **Definition**
  - An *optimization* is a transformation that is expected to improve program in some way
    - Program running time, memory assumption, power consumption, etc.

- **Basic components**
  - Code analysis: analyze the code to derive knowledge about run-time behavior
  - Code rewriting: use collected knowledge to improve the code

WHY STUDY COMPILERS OPTIMIZATION?

- An opportunity to explore compiler techniques in both breadth and depth
  - Parallelization? Functional?
  - Optimizations with more details

- Compiler optimizations rely on both program analysis and transformation, which are useful in many related areas
  - Software engineering: program understanding / reverse engineering / debugging
  - Run-time support and improvement

- Open problems
  - Engineering effort: limits and issues
  - Motivate research topics
NO MAGIC BULLET

- Fully optimizing compiler (FOC)
  - For any input program P, FOC guarantees to generate the fastest/smallest translation with the same behavior
  - Impossible to construct
    - Otherwise, we solve the halting problem!

WORD “OPTIMIZATION”

A clear misnomer!!

OPTIMIZATIONS

“Optimization for scalar machines is a problem that was solved ten years ago”
David Kuck - 1990  Rice University

- Changes in architecture changes in compilers
  - new features present new problems
  - changing costs lead to different concerns
  - must re-engineer well-known solutions

- Compiler optimizations can significantly improve performance
CODE OPTIMIZATION?

Compiler optimization is essential for modern computer architectures.

- Without optimization, most applications would perform very poorly on modern architectures
- Even with optimization, most applications do not get a high fraction of peak performance
- Optimization techniques are also basis for exploiting SIMD components, vector units, hyperthreading, other forms of multithreading

What is it that compilers can exploit?  What does the compiler (writer) need to know about architectures?

CODE OPTIMIZATIONS

Modern optimizers are structured as a series of passes

- Optimizers
  - Work on some form of intermediate representations
  - Multiple passes are typical
    - Each with a particular optimization task
EVALUATION OF CODE OPTIMIZATIONS

- Safety
  - Need to be sure the transformed code preserves the meaning of the original code
    - Generate the same output given the same input
- Profitability
  - Need to improve program in some way
    - Enabling optimizations: interaction with other transformations
- Opportunity
  - Need to know where/when to apply
    - Average case or special programs/inputs only
- Overhead
  - Must be practically computable
  - Should be justified by the profit

TYPES OF CODE OPTIMIZATIONS

- Machine-independent optimizations
  - Eliminate redundant computation
  - Eliminate dead code
  - Perform computation at compile time if possible
  - Execute code less frequently

- Machine-dependent optimizations
  - Hide latency
  - Parallelize computation
  - Replace expensive computations with cheaper ones
  - Improve memory performance
Scopes of Code Optimizations

- **Peephole optimizations**
  - Consider a small window of instructions

- **Local optimizations**
  - Consider instruction sequences within a basic block

- **Intra-procedural (global) optimizations**
  - Consider multiple basic blocks within a procedure
  - Need support from control flow analysis
    - Branches, loops, merging of flows

- **Inter-procedural optimizations**
  - Consider the whole program w/ multiple procedures
  - Need to analyze calls/returns

Peephole optimizations

- **Constant Folding**
  - \( x := 32 \) becomes \( x := 64 \)
  - \( x := x + 32 \)

- **Unreachable Code**
  - \( \text{goto L2} \)
  - \( x := x + 1 \) \( \Leftarrow \) unneeded

- **Flow of control optimizations**
  - \( \text{goto L1} \) becomes \( \text{goto L2} \)
  - ...
  - \( L1: \text{goto L2} \)
PEEPHOLE OPTIMIZATIONS

- Algebraic Simplification
  \[ x := x + 0 \leftarrow \text{unneeded} \]

- Dead code
  \[ x := 32 \leftarrow \text{where } x \text{ not used after statement} \]
  \[ y := x + y \quad \rightarrow y := y + 32 \]

- Reduction in strength
  \[ x := x \times 2 \quad \rightarrow x := x + x \]

PEEPHOLE OPTIMIZATIONS

- Local in nature
- Pattern driven
- Limited by the size of the window
Basic Block Level

- Common Subexpression elimination
- Constant Propagation
- Dead code elimination
- Plus many others such as copy propagation, value numbering, partial redundancy elimination, ...

Sample Optimizations

- Redundant loads and stores elimination
  - MOV R0, a  ⇒  MOV R0, a
  - MOV a, R0
- Unreachable code elimination
  - GOTO L2
    - x := x + 1  ⇐ unreachable
- Algebraic identities
  - x := x + 0  ⇐ can eliminate
  - x := x * 1
- Reduction in strength
  - x := x * 2  ⇒  x := x + x
- Constant folding
  - p = 2 * 3.14  ⇒  p = 6.28
- Constant propagation
  - p = 6.28  p=6.28
  - x = x * p  ⇒  x = x * 6.28
SAMPLE OPTIMIZATIONS

- Common sub-expression elimination
  - Local
    \[
    m = 2 \times y \times z \\
    o = 2 \times y - z \\
    \Rightarrow \\
    t = 2 \times y \\
    m = t \times z \\
    o = t - z
    \]
  - Global
    \[
    a := d + e \\
    b := d + e \\
    c := d + e \\
    \Rightarrow \\
    t := d + e \\
    a := t \\
    b := t \\
    c := t
    \]
  - Global partial
    \[
    a := d + e \\
    b := d + e \\
    c := d + e \\
    \Rightarrow \\
    t := d + e \\
    a := t \\
    b := t \\
    c := t
    \]

SAMPLE OPTIMIZATIONS

- Loop optimizations
  - Code motion
    \[
    \text{while (} i \leq \text{limit - 2) } \{ \ldots \} \quad \Rightarrow \\
    t = \text{limit - 2} \\
    \text{while (} i \leq t) \{ \ldots \}
    \]
  - Loop unrolling
    \[
    \text{do } i=1 \text{ to } n \text{ by 1} \\
    a(i) = a(i)+b(i) \\
    \Rightarrow \\
    \text{do } i=1 \text{ to } n \text{ by 4} \\
    a(i) = a(i)+b(i) \\
    a(i+1) = a(i+1)+b(i+1) \\
    a(i+2) = a(i+2)+b(i+2) \\
    a(i+3) = a(i+3)+b(i+3) \\
    \text{end} \\
    \ldots //\text{process tail part}
**Redundant Expressions**

An expression \( x \ op \ y \) is redundant at a point \( p \) if it has already been computed at some point(s) and no intervening operations redefine \( x \) or \( y \).

\[
\begin{align*}
m &= 2y z & t0 &= 2y & t0 &= 2y \\
n &= 3y z & t1 &= 3y & t1 &= 3y \\
o &= 2y z & t2 &= 2y & o &= t0 z
\end{align*}
\]

redundant

**Copy Propagation**

\[
\begin{align*}
b &= a \\
c &= 4b \\
c &> b \\
e &= a + b \\
d &= b + 2
\end{align*}
\]
**Simple example: a[i+1] = b[i+1]**

- $t_1 = i + 1$
- $t_2 = b[t_1]$
- $t_3 = i + 1$
- $a[t_3] = t_2$

- $t_1 = i + 1$
- $t_2 = b[t_1]$
- $t_3 = i + 1$  \( \leftarrow \text{no longer live} \)
- $a[t_1] = t_2$

Common expression can be eliminated

Now, suppose $i$ is a constant:

- $i = 4$
- $t_1 = i + 1$
- $t_2 = b[t_1]$
- $a[t_1] = t_2$

- $i = 4$
- $t_1 = 5$
- $t_2 = b[t_1]$
- $a[t_1] = t_2$

Final Code:

- $i = 4$
- $t_2 = b[5]$
- $a[5] = t_2$
Simple Loop Optimizations: Strength Reduction

- **Induction Variables** control loop iterations

Diagram: (Induction Variable expressions)

Simple Loop Optimizations

- Loop transformations are often used to expose other optimization opportunities:
  - Normalization
  - Loop Interchange
  - Loop Fusion
  - Loop Reversal
  - ...
**Consider Matrix Multiplication**

```
for i = 1 to n do
    for j = 1 to n do
        for k = 1 to n do
        end
    end
end
```

**Memory Usage**

- **For A:** Elements are accessed across rows, spatial locality is exploited for cache (assuming row major storage)
- **For B:** Elements are accessed along columns, unless cache can hold all of B, cache will have problems.
- **For C:** Single element computed per loop – use register to hold

```
C
```

```
A
```

```
B
```
**Matrix Multiplication Version 2**

for $i = 1$ to $n$ do
  for $k = 1$ to $n$ do
    for $j = 1$ to $n$ do
    end
  end
end

**Memory Usage**

- **For A:** Single element loaded for loop body
- **For B:** Elements are accessed along rows to exploit spatial locality.
- **For C:** Extra loading/storing, but across rows
**Simple Loop Optimizations**

- **How to determine safety?**
  - Does the new multiply give the same answer?
  - Can be reversed??
    
    ```
    for (I=1 to N) a[I] = a[I+1] – can this loop be safely reversed?
    ```

**Types of Code Optimizations**

- **Machine-independent optimizations**
  - Eliminate redundant computation
  - Eliminate dead code
  - Perform computation at compile time if possible
  - Execute code less frequently

- **Machine-dependent optimizations**
  - Hide latency
  - Parallelize computation
  - Replace expensive computations with cheaper ones
  - Improve memory performance