Copyright 2011, Keith D. Cooper & Linda Torczon, all rights reserved.
Students enrolled in Comp 512 at Rice University have explicit permission to make copies of these materials for their personal use.
Classic Compilers

Compiler design has been fixed since 1960

- Front End, Middle End, & Back End
- Series of filter-style passes
  (number of passes varies)
- Fixed order for passes
1957: The FORTRAN Automatic Coding System

- Six passes in a fixed order
- Generated good code
  - Assumed unlimited index registers
  - Code motion out of loops, with ifs and gotos
  - Did flow analysis & register allocation
1999: The SUIF Compiler System

- 3 front ends, 3 back ends
- 18 passes, configurable order
- Two-level IR (High SUIF, Low SUIF)
- Intended as research infrastructure

**Classic Compilers**

**Fortran 77**
**C & C++**
**Java**

**Middle End**

**C/Fortran**
**Alpha**
**x86**

**Front End**

**Back End**

**Academic research system (Stanford)**

- Data dependence analysis
- Redundancy elimination
- Reduction recognition
- Pointer analysis
- Affine loop transformations
- Blocking
- Capturing object definitions
- Virtual function call elimination
- Garbage collection
- SSA construction
- Dead code elimination
- Partial redundancy elimination
- Constant propagation
- Global value numbering
- Strength reduction
- Reassociation
- Instruction scheduling
- Register allocation
2000: The SGI Pro64 Compiler, now “Open 64”

Open source optimizing compiler for IA 64
- 3 front ends, 1 back end
- Five-level IR
- Gradual lowering of abstraction level
Even a 2000 JIT fits the mold, albeit with fewer passes

- Front end tasks are handled elsewhere
- Few (if any) optimizations
  - Avoid expensive analysis
  - Emphasis on generating native code
  - Compilation must be profitable
Most optimizing compilers fit this basic framework

What’s the difference between them?
  > More boxes, better boxes, different boxes
  > Picking the right boxes in the right order

To understand the issues
  > Must study compilers, for big picture issues
    
  > Must study boxes, for detail issues

Look at some of the great compilers of yesteryear

We will do both
Fortran H Enhanced (the “new” compiler)

Improved Optimization of Fortran Object Programs
R.G. Scarborough & H.G. Kolsky

Started with a good compiler — Fortran H Extended
• Fortran H - one of 1\textsuperscript{st} commercial compilers to perform systematic analysis (both control flow & data flow)
• Extended for System 370 features
• Subsequently served as model for parts of VS Fortran
  — not a great compiler

Authors had commercial concerns
• Compilation speed
• Bit-by-bit equality of results
• Numerical methods must remain fixed

Paper describes improvements in the -O2 optimization path
Fortran H Extended (the “old” compiler)

Some of its quality comes from choosing the right code shape

Translation to quads performs careful local optimization
• Replace integer multiply by $2^k$ with a shift
• Expand exponentiation by known integer constant
• Performs minor algebraic simplification on the fly
  > Handling multiple negations, local constant folding

Classic example of “code shape”
• Bill Wulf popularized the term (probably coined it)
• Refers to the choice of specific code sequences
• “Shape” often encodes heuristics to handle complex issues
### Code Shape

**My favorite example**

1. **What if** $x$ is 2 and $z$ is 3?
2. **What if** $y+z$ is evaluated earlier?

The “best” shape for the $x+y+z$ depends on contextual knowledge:

- There may be several conflicting options

---

Addition is commutative & associative for integers
Some of the improvement in Fortran H comes from choosing the right code shape for their target & their optimizations

- Shape simplifies the analysis & optimization
- Shape encodes heuristics to handle complex issues

The rest came from systematic application of a few optimizations

- Common subexpression elimination
- Code motion
- Strength reduction
- Register allocation
- Branch optimization

Not many optimizations, by modern standards …

(e.g., SUIF, OPEN 64)
Summary

- This compiler fits the classic model
- Focused on a single loop at a time for optimization
- Worked innermost loop to outermost loop
- Compiler was just 27,415 lines of Fortran + 16,721 lines of asm
This work began as a study of customer applications
- Found many loops that could be better
- Project aimed to produce hand-coded quality
- Project had clear, well-defined standards & goals
- Project had clear, well-defined stopping point

Fortran H Enhanced was already an effective compiler

<table>
<thead>
<tr>
<th>Instruction Type</th>
<th>Fortran G1</th>
<th>H Extended</th>
<th>H Enhanced</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>count</td>
<td>pct</td>
<td>count</td>
</tr>
<tr>
<td>Integer</td>
<td>70.216</td>
<td>83.5</td>
<td>7.120</td>
</tr>
<tr>
<td>Float</td>
<td>10.994</td>
<td>13.1</td>
<td>9.976</td>
</tr>
<tr>
<td>Control</td>
<td>1.456</td>
<td>1.7</td>
<td>1.435</td>
</tr>
<tr>
<td>Others</td>
<td>1.459</td>
<td>1.7</td>
<td>0.044</td>
</tr>
<tr>
<td>Totals</td>
<td>84.126</td>
<td>100.0</td>
<td>18.575</td>
</tr>
</tbody>
</table>

Aggregate operations for a plasma physics code, in millions

- Little decrease in useful ops
- Huge decrease in overhead ops

Another 35%

78% reduction
Fortran H Enhanced (new)

How did they improve it?

The work focused on four areas

• Reassociation of subscript expressions
• Rejuvenating strength reduction
• Improving register allocation
• Engineering issues

Note: this is not a long list!
Reassociation of Subscript Expressions

• Don’t generate the standard address polynomial

For those of you educated from EaC, a history lesson is needed

• Prior to this paper (& much later in the texts) the conventional wisdom was to generate the following code:

For a 2-d array \( A \) declared as \( A(\text{low}_1:\text{high}_1,\text{low}_2:\text{high}_2) \)

The reference \( A(i_1,i_2) \) generates the polynomial

\[
A_0 + ((i_2 - \text{low}_2) \times (\text{high}_1 - \text{low}_1 + 1) + i_1 - \text{low}_1) \times w
\]

• This form of the polynomial minimizes total ops
  > Good for operation count, bad for common subexpression elimination, strength reduction, instruction scheduling, ...
  > With \( A(i+1,j) \) and \( A(i+1,j+1) \) the difference is bound into the expression before the common piece can be exposed

Column-major order because the paper is about FORTRAN
Reassociation of Subscript Expressions

For a 2-d array $A$ declared as $A(low_1:high_1,low_2:high_2)$

The reference $A(i_1,i_2)$ generates the polynomial

$$A_0 + ((i_2 - low_2) \times (high_1 - low_1 + 1) + i_1 - low_1) \times w$$

- This form of the polynomial minimizes total ops
  - Good for operation count, bad for common subexpression elimination, strength reduction, instruction scheduling, ...
  - With $A(i+1,j)$ and $A(i+1,j+1)$ the difference is bound into the expression before the common piece can be exposed

- Now, imagine a typical “stencil” computation

  $$a(i,j) = (a(i-1,j) + a(i,j) + a(i+1,j) + a(i,j-1) + a(i,j+1))/5$$

Surrounding loops (on i, then j) move the stencil over the entire array, adjusting the value of the central element …

Typical stencils include 5, 7, 11 points
Reassociation of Subscript Expressions

For a 2-d array $A$ declared as $A(\text{low}_1: \text{high}_1, \text{low}_2: \text{high}_2)$

The reference $A(i_1, i_2)$ generates the polynomial

$$A_0 + ((i_2 - \text{low}_2) \times (\text{high}_1 - \text{low}_1 + 1) + i_1 - \text{low}_1) \times w$$

- This form of the polynomial minimizes total ops
  - Good for operation count, bad for common subexpression elimination, strength reduction, instruction scheduling, ...
  - With $A(i+1, j)$ and $A(i+1, j+1)$ the difference is bound into the expression before the common piece can be exposed

- Now, imagine a typical “stencil” computation

$$a(i, j) = (a(i-1, j) + a(i, j) + a(i+1, j) + a(i, j-1) + a(i, j+1))/5$$

And the subexpressions found (or hidden) inside it ...
Reassociation of Subscript Expressions

- Don’t generate the standard address polynomial
  - *Forget the classic address polynomial*
- Break polynomial into six parts
  - Separate the parts that fall naturally into outer loops
  - Compute everything possible at compile time
- Makes the tree for address expressions broad, not deep
- Group together operands that vary at the same loop level

The point

- Pick the *right* shape for the code *(expose the opportunity)*
- Let other optimizations do the work
- Sources of improvement
  - Fewer operations execute
  - Decreases sensitivity to number of dimensions

Tradeoff driven by cse versus strength reduction

Novel improvement

Read p 665ff carefully
**Reassociation of Subscript Expressions**

Distribution creates different expressions

\[ w + y \times (x + z) \Rightarrow w + y \times x + y \times z \]

More operations, but they may move to different places

Consider \( A(i,j) \), where \( A \) is declared \( A(0:n,0:m) \)

- Standard polynomial: \( \@A + (i \times m + j) \times w \)
- Alternative: \( \@A + i \times m \times w + j \times w \)

Does this help?
- \( i \) part and \( j \) part vary in different loops
- Standard polynomial pins \( j \) in the loop where \( i \) varies

Can produce **significant** reductions in operation count

General problem, however, is quite complex
Reduction of Strength

- Many cases had been disabled in maintenance
  - Almost all the subtraction cases turned off
- Fixed the bugs and re-enables the corresponding cases
- Caught “almost all” the eligible cases

Extensions

- Iterate the transformations
  - Avoid ordering problems
  - Catch secondary effects
- Capitalize on user-coded reductions
- Eliminate duplicate induction variables
Register Allocation

Original Allocator

- Divide register set into local & global pools
- Different mechanisms for each pool

Problems

- Bad interactions between local & global allocation
- Unused registers dedicated to the procedure linkage
- Unused registers dedicated to the global pool
- Extra (unneeded) initializations

*Remember the 360*

- Two-address machine
- Destructive operations
Register Allocation

New Allocator

• Remap to avoid local/global duplication
• Scavenge unused registers for local use
• Remove dead initializations
• Section-oriented branch optimizations

All symptoms arise from not having a global register allocator — such as a graph coloring allocator

Plus …

• Change in local spill heuristic
• Can allocate all four floating-point registers
• Bias register choice by selection in inner loops
• Better spill cost estimates
• Better branch-on-index selection
Engineering Issues

Increased the name space

- Was 127 slots (80 for variables & constants, 47 for compiler)
- Increased to 991 slots
- Constants no longer need slots
- “Very large” routines need < 700 slots  (remember inlining study?)

Common subexpression elimination (CSE)

- Removed limit on backward search for CSEs
- Taught CSE to avoid some substitutions that cause spills

Extended constant handling to negative values
Results

They stopped working on the optimizer.
Hand-coding no longer improved the inner loops.
\[\Rightarrow\] Produced significant change in ratio of flops to instructions

I consider this to be the classic Fortran optimizing compiler

<table>
<thead>
<tr>
<th>Instruction Type</th>
<th>Fortran</th>
<th>G1</th>
<th>H Extended</th>
<th>H Enhanced</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>count</td>
<td>pct</td>
<td>count</td>
<td>pct</td>
</tr>
<tr>
<td>Integer</td>
<td>70.216</td>
<td>83.5</td>
<td>7.120</td>
<td>38.3</td>
</tr>
<tr>
<td>Float</td>
<td>10.994</td>
<td>13.1</td>
<td>9.976</td>
<td>53.7</td>
</tr>
<tr>
<td>Control</td>
<td>1.456</td>
<td>1.7</td>
<td>1.435</td>
<td>7.7</td>
</tr>
<tr>
<td>Others</td>
<td>1.459</td>
<td>1.7</td>
<td>0.044</td>
<td>0.2</td>
</tr>
<tr>
<td>Totals</td>
<td>84.126</td>
<td>100.0</td>
<td>18.575</td>
<td>100.0</td>
</tr>
</tbody>
</table>

Aggregate operations for a plasma physics code, in millions
Results

Final points

• Performance numbers vary from model to model
• Compiler ran faster, too!

It relies on

• A handful of carefully targeted optimizations
• Generating the right IR in the first place \(\text{\textcopyright code shape}\)