Compiler Optimization: Increasing Research Impact

Chris Lattner
LLVM Founder & Architect
CGO 2012 - April 2, 2012
A Disclaimer

• This talk:
  – is highly biased by personal experiences and opinions
  – plays to broad stereotypes :-)

• Uses code optimization as an example
  – focused on traditional static compilation
  – focused on open source compilers

• Not intended to be new or novel
  – hopefully some tasty food for thought
Roadmap

• Industry vs Academia
• Heroic Optimizations
• Open Problems in LLVM
• Suggestions for the Community
Academia vs Industry
Perspectives on Code Optimization

• How do we speed something up?
• What do we control?
Perspectives - Compiler Academia

• Can’t change the benchmark
  – Results on well known benchmarks ⇒ credibility

• Easy (and desirable!) to change the compiler
  – Preferably in novel / publishable ways
  – Quality threshold: enough to run benchmarks
Perspectives - Industry

• “Hard” to change the compiler
  – Compiler engineers are specialized
  – Many competing demands
  – Compiler needs to be ~100% reliable

• Easy to change the application
  – Code changing and evolving rapidly
  – Performance tools are a necessity!
Tradeoffs

• Improving the compiler:
  – Benefits lots of code
  – Expensive

• Improving the application:
  – Only helps one application
  – Cheap

http://llvm.org/
Motivation, Goals, Results, and Impact: Academia
Motivation and Goals

• Motivation:
  – Contribution to the field
  – Graduate degree, Tenure, ...

• Goal:
  – Paper publication
  – Novel research contribution
  – Build towards large research goals
  – Future citations
Result and Impact

• Result:
  – “Our optimization speeds up SPECINT2000 by a geometric mean of 10% compared to our baseline”
  – New ideas and algorithms
  – Basis of future work

• Impact:
  – Achieved goal
  – Unclear impact on real-world code
  – Optimization never ships in production compiler
Motivation, Goals, Results, and Impact: Industry
Motivation and Goals

• Motivation:
  – “Video playback on widget X is stuttering!”

• Goal:
  – Video decoder runs 25% faster
Result and Impact

• Result:
  – 2%: improved modeling of subregister kill flags
  – 3%: form FMAs more aggressively with -ffast-math
  – 0%: add builtin for “sum of absolute differences”
  – 20%: source changes to video decoder

• Impact:
  – Achieved goal
  – Better video decoder code base
  – Modest compiler improvements:
    – Broad code benefits
    – Composes with future changes
  – Product ships on time
Heroic Optimizations
A random example

“[Our work] improves the performance of many programs from 5% to 20%, improves analyzer and llu-bench by roughly 2X, and ft and chomp more than 10X.”

"Automatic Pool Allocation: Improving Performance by Controlling Data Structure Layout in the Heap"
Chris Lattner and Vikram Adve
PLDI 2005
What is a heroic optimization?

- Success leads to a dramatic performance effect
  - Failure implies no performance change or a loss

- Anything relying on heroic analysis:
  - Shape analysis transformations
  - Restructuring optimizations for cache
  - Auto-parallelization
  - Many SPEC hacks :-)

http://llvm.org/
What’s the problem?

• App performance swings wildly as code changes
  – Small changes to an app can “break” optimization
  – e.g. one new alias introduced
  – How does a developer predict or control this?

• Often unrealistic assumptions:
  – e.g. requires the “whole program”

• Difficult to justify in production setting
  – Hard to qualify correctness
  – “Kicks in” in limited situations
When can it make sense?

• Code you can’t (or don’t want to) change
  – Benchmark hacks (aka Marketing :-)
  – Legacy code - e.g. dusty deck Fortran

• Optimizing common idioms
  – Pattern matching loops to memset/memcpy

• Overcoming source language limitations:
  – e.g. 1D arrays in Java

http://llvm.org/
Solving the Transparency Problem

• Look to auto-vectorizers for inspiration:
  – Report what optimizations happened
  – Report why an expected optimization failed

• Hard problems:
  – Optimizations happen on IR, not source code
  – Some concepts are very abstract!
Better Programming Languages?

• Abstract away details, not algorithmic issues:
  – Good: register allocation, inst selection, scheduling
  – Bad: cache behavior, vectors vs scalars, parallelism

• Provide abstractions for architecture portability:
  – Allow reasoning about memory hierarchy
  – Allow expressing intentions of how code is run

• Need to verify that the “right thing” happens
What about Performance Tools?

• Explain how to restructure code for performance
  – Instead of automatically fixing it during compilation

• Benefits:
  – Communicate to the developer in terms of source code
  – Compile time doesn’t matter for off-line tools
  – Less fragile as code evolves

• Challenges:
  – Need to reimplement most analyses
  – Source level is more complex than IR
Other Challenges

• Many people don't want to look at assembly
  – higher level way to reason about code execution?
  – still need to see what happens after optimization

• Distributed performance problems
  – Small slowdown, spread across the entire app
  – No hot spot, no obvious way to find the culprit
  – Extremely common in C++ and OO apps
Some Open Problems in LLVM
Instruction Selection for Vector Shuffles

- Shuffles critical for OpenGL / CL and vectorization
  - Produce vector from two inputs
  - Allows “don’t care” elements in the result

shufflevector %V1, %V2, <i32 3, i32 7, i32 7, i32 4>
**Instruction Selection for Vector Shuffles**

![Diagram showing vector shuffles and constants.]

**PowerPC Altivec Example**

```asm
li r8, lo16(LCPI0_0)
lis r7, ha16(LCPI0_0)
lvx v2, r7, r8
vperm v2, v4, v3, v2
```

Better: Three Shuffles

```asm
vmrglw v3, v3, v2
vmrglw v3, v3, v3
vsldoi v2, v3, v2, 4
```

```
shufflevector %V1, %V2, <i32 3, i32 7, i32 7, i32 4>
```
“Perfect” Shuffle

- Precomputed table of shuffles
  - 4 Elements: $9 \times 9 \times 9 \times 9 = 6561$ entries * 4 bytes = 26K
  - Code generator indexes into table to emit code

```c
%result = vsldoi %tmp1, %V2, 4
%result = shuffle %V1, %V2, <3,7,7,4>
```
"Perfect" Shuffle

- Precomputed table of shuffles
  - 4 Elements: 9*9*9*9=6561 entries * 4 bytes = 26K
  - Code generator indexes into table to emit code

```
%result = vsldoi %tmp1, %V2, 4
%tmp1 = vmrglw %tmp2, %tmp2
%result = vsldoi %tmp1, %V2, 4
%result = shuffle %V1, %V2, <3,7,7,4>
```
“Perfect” Shuffle

• Precomputed table of shuffles
  – 4 Elements: $9 \times 9 \times 9 \times 9 = 6561$ entries * 4 bytes = 26K
  – Code generator indexes into table to emit code

  ... 
  1256575800U, // <2,6,3,6>: Cost 2 vmrglw V1, <6,6,6,6>
  135056694U, // <2,6,3,7>: Cost 1 vmrglw V1, V2
  135056695U, // <2,6,3,u>: Cost 1 vmrglw V1, V2
  ... 
  3371648548U, // <3,3,7,6>: Cost 4 vmrglw <2,6,3,7>, <2,1,3,6>
  1224165306U, // <3,3,7,7>: Cost 2 vmrglw <2,6,3,7>, <2,6,3,7>
  1224165306U, // <3,3,7,u>: Cost 2 vmrglw <2,6,3,7>, <2,6,3,7>
  ... 
  2297907567U, // <3,7,7,3>: Cost 3 vmrglw <2,6,3,7>, <3,2,7,3>
  2637729078U, // <3,7,7,4>: Cost 3 vsldoi,4 <3,3,7,7>, V2
  3371649312U, // <3,7,7,5>: Cost 4 vmrglw <2,6,3,7>, <3,1,7,5>
  ... 

  %tmp2 = vmrglw %V1, %V2
  %tmp1 = vmrglw %tmp2, %tmp2
  %result = vsldoi %tmp1, %V2, 4

  %result = shuffle %V1, %V2, <3,7,7,4>
Perfect Shuffle Problems

• More than 4 elements:
  – 8 Elements: $9^8$ table entries = 172MB
  – 16 Elements: $9^{16}$ table entries = $1.8e15$ entries

• X86 Code Generation:
  – One table per SSE level prohibitive
  – Some operations can fold memory loads

• End result:
  – A pile of heuristics and hacks
Interprocedural Alias/ModRef Analysis

• Obviously, a very well explored area
• Remains very difficult to use in practice
LLVM AA Challenges

• Don't want to recompute it for every optimization
  – Compute once early on, and update it
  – ... Optimizations must update Alias Analysis

• Want flexibility for different AA implementations
  – Allow easy experimentation
  – Willing to limit to flow-insensitivity
LLVM Alias Analysis Needs

• Really need an “Alias Analysis IR API”
  – Abstraction between clients and implementations
  – Must be efficient (no parallel data structures)

• Should support full generality:
  – Alias queries
  – Mod/Ref queries
  – Pointer capture analysis
  – Type-Based alias analysis
Some Suggestions & Comments
Reproducibility/Believability of Results

- Results vary widely with:
  - Target Architecture & Source Language
  - Compiler Infrastructure
  - Quality of Implementation
  - Benchmark set
Reproducibility of Results: Wishlist

- Use a specific version of a well known compiler:
  - “LLVM 3.0”, “GCC 4.7”, “GHC 7.4.1” ...
  - Avoids measuring artifacts of an immature foundation

- Measure, measure, measure:
  - Dynamic performance, static metrics
  - Code size, compile time

- Publish implementation and dataset:
  - github link in your paper!
  - No need for it to be production quality
Empirical Meta-Comparison Studies

• Solutions to important problems have:
  – multiple different algorithms
  – many implementation refinements
  – different tradeoffs (e.g. analysis time vs quality)

• Need more studies to *fairly* compare these:
  – ideally by third parties
  – code and dataset made available for scrutiny
  – as apples-to-apples as possible

• Obvious “citation bait”!
Use LLVM!

• LLVM advantages:
  – Well known, mature, and robust
  – Widely used in both industry and academia
  – Modular code base with modern design
  – Spans the entire toolchain:
    – assembler to compiler, runtime, and debugger

• Other advantages:
  – Great basis to measure and share reproducible results
  – Blog is a great platform to advertise your work
  – LLVM experience is very useful in the job market
Wrap up

• Industry and Academia work differently
  – Different goals lead to different results

• Helping humans write better code is useful
  – Just as much as doing it automatically in a compiler

• LLVM is a fantastic foundation for research:
  – Mature, well known, widely used
  – Easy to work with and change
  – Lots of hard problems left!

http://llvm.org/