Evolutionary code optimisation

CERN tCSC, 2017

Nikola Hardi
nhardi@cern.ch

June 9, 2017

Previoius and related work

Fortran Automatic Coding System

The purpose of section 4 is to prepare for section 5 a table of predecessors (PRED table) which enumerates the basic blocks and lists for every basic block each of the basic blocks which can be its immediate predecessor in flow, together with the absolute frequency of each such basic block link. **This table is obtained by an actual "execution" of the program in Monte-Carlo fashion**, *in which the outcome of conditional transfers arising out of IF-type statements and computed GO TO'S is determined by a random number generator suitably weighted according to whatever FREQUENCY statements have been provided.*

"Fortran Automatic Coding System", J. Backus, 1957

Superoptimiser

> **Given an instruction set, the superoptimizer finds the shortest program to compute a function.** *Startling programs have been generated, many of them engaging in convoluted bit-fiddling bearing little resemblance to the source programs which defined the functions.*

"Superoptimizer - A Look at the Smallest Program", H. Massalin, 1987

LLVM and lifelong code optimisation

> *While some applications have small hot spots, others spread their execution time evenly throughout the application. In order to maximize the efficiency of all of these programs, we believe that* **program analysis and transformation must be performed throughout the lifetime of a program**. *Such lifelong code optimization techniques encompass interprocedural optimizations performed at link-time (to preserve the benefits of separate compilation), machine-dependent optimizations at install time on each system, dynamic optimization at run- time, and profile-guided optimization between runs (idle time) using profile information collected from the end-user.*

"LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation", C. Lattner, V. Adve, 2004

MILEPOST GCC and cTuning

MILEPOST GCC is free community-driven open-source adaptive self-tuning compiler that combines stable production-quality GCC, Interactive Compilation Interface and machine learning plugins to **adapt to any given architecture and program automatically** *and predict profitable optimizations to improve program execution time, code size and compilation time.*

"Milepost gcc: Machine learning enabled self-tuning compiler International journal of parallel programming", G. Fursin, 2011

The LLVM effect

Standard compiler features

- Many frontends
- Stable IR code
- Strict optimisation passes
- Many backends

Young programming languages

- Rust
- Julia
- (Haskell)

Iterative compilation

Why do we care?

Developers viewpoint

- Even few percents are important (consumer electronics)
- Computers are cheaper than paying a human

Compiler builders viewpoint

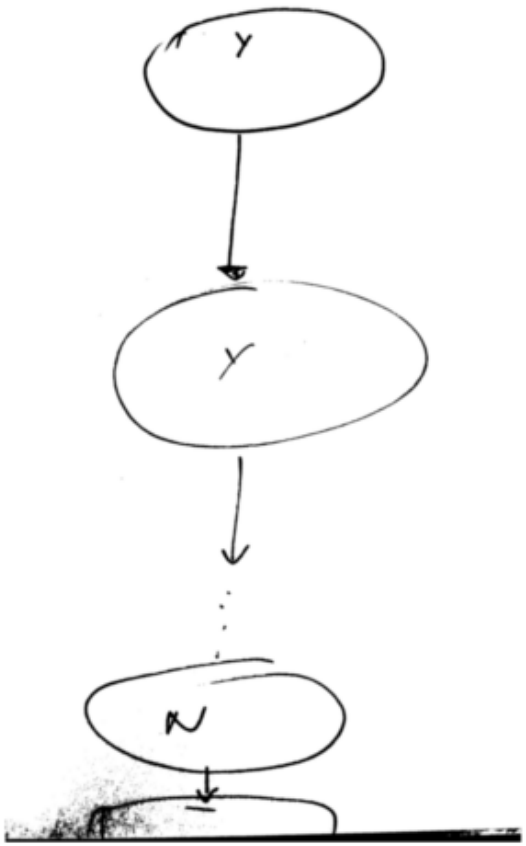
- Heuristic based decision are uncertain
- Usually impossible to generalize
- Architecture/system dependant details

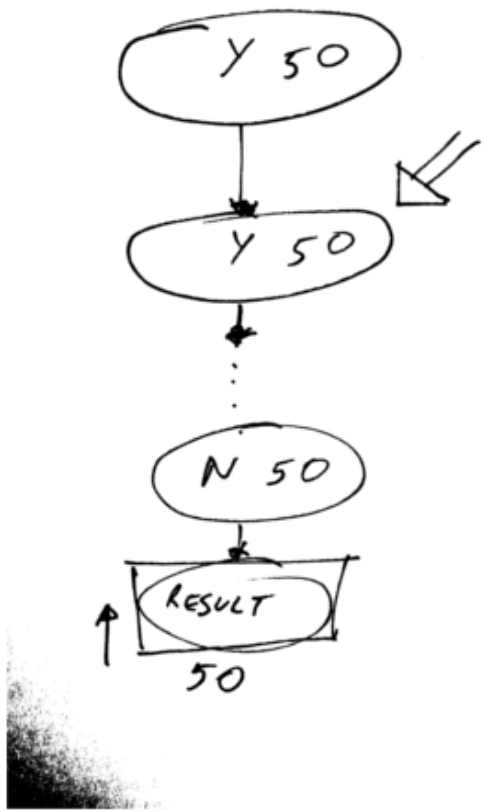
A proposed solution

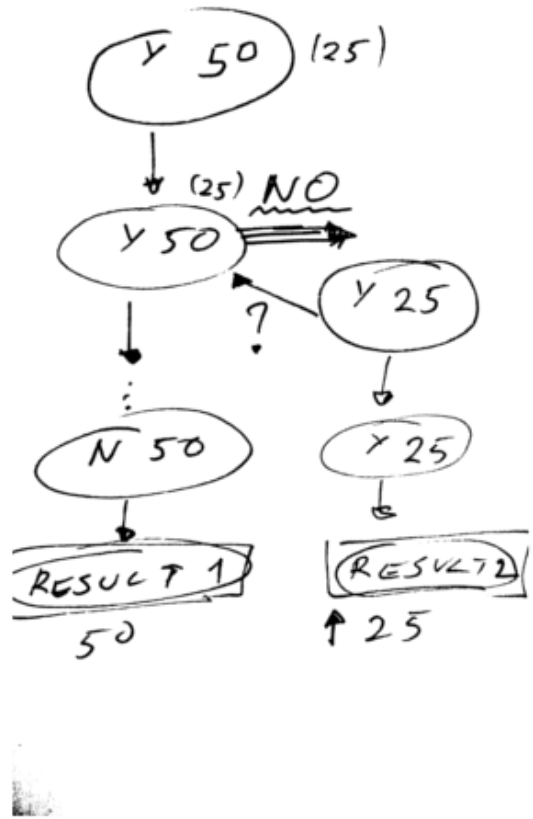
- Tweak it a bit
- Compile
- Evaluate
- Repeat

Implementation

- Form a tree
- Each call to heuristic is a node
- Each leaf is a compiled program
- Right child is alternative path
- A path to the leaf is one iteration
- Evaluated results propagate upwards, better wins
- You can flip only one decision per iteration
- Repeat


Y
Y
N


Y 50
Y 50
N 50
RESULT
50


Y 50 (25)
(25) NO
Y 50
Y 25
?
N 50
Y 25
RESULT 1
RESULT 2
50
25

Exploring the optimisation space

- Too big to explore completely
- Standard approaches: random walk, hill climber...
- Lots of redundant paths
- Non-convex, non-linear

Optimising for performance

Decomposing a program

- Single entry point for program
- Single entry point for function
- Single path through a basic block
- Basic unit of measure: instructions

Optimising for performance

Decomposing the problem:

- Good block frequency prediction
- Good CPU model

Simplified CPU model:

> *Executioin time is equal to the total number of instructions to be executed.*

Test setup and results

Test setup

- MIPS architecture, ImaginationTechnologies CI20
- Monitoring the PMU
- Using LLVM single source regression tests
- Evaluated for all optimisation levels (O0-O3 + Os)

Results

- Optimise for size: improvements 5 percents for LLVM-IC
- Performance prediction by static analysis: 85+ percent with LLVM BFA and M1

Future work

- Optimise just part of the code
- Optimise several parts in parallel
- Improve block frequency analysis
- Develop advanced CPU models (pipeline, superscalars...)