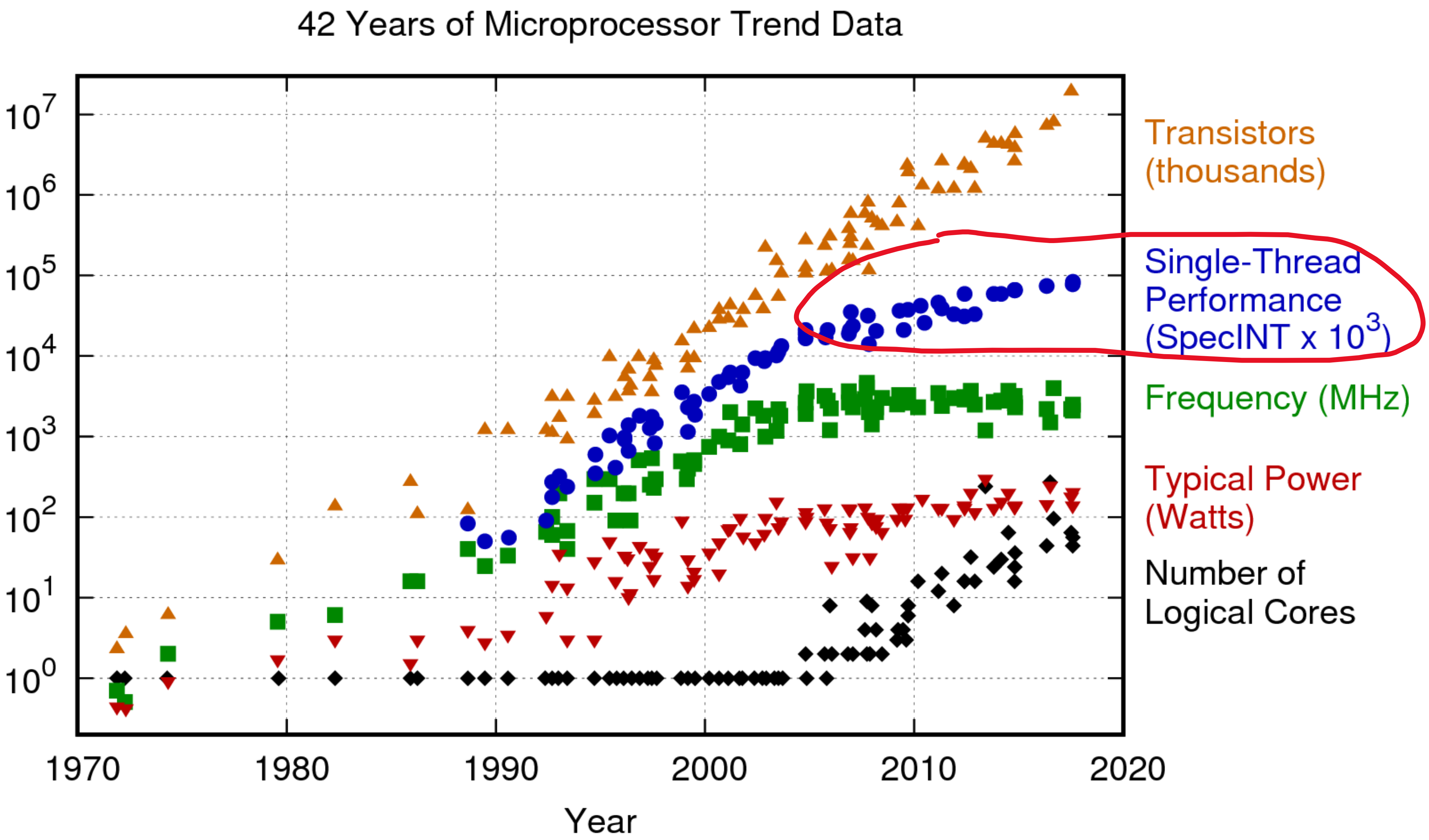
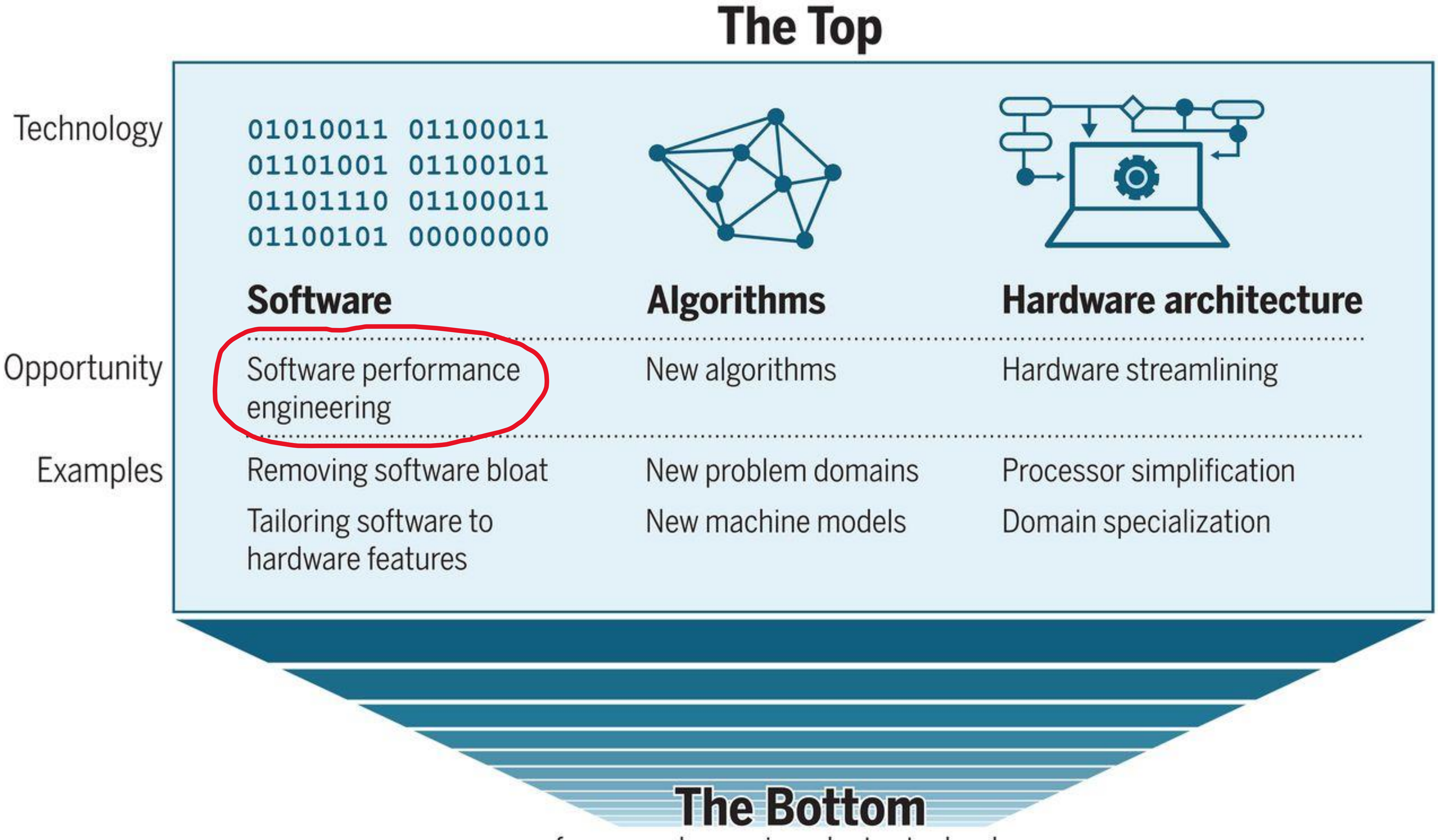
Performance Tuning: Future Compiler Improvements

Denis Bakhvalov

5th LLVM Performance Workshop at CGO
February 2021



Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2017 by K. Rupp



The Top

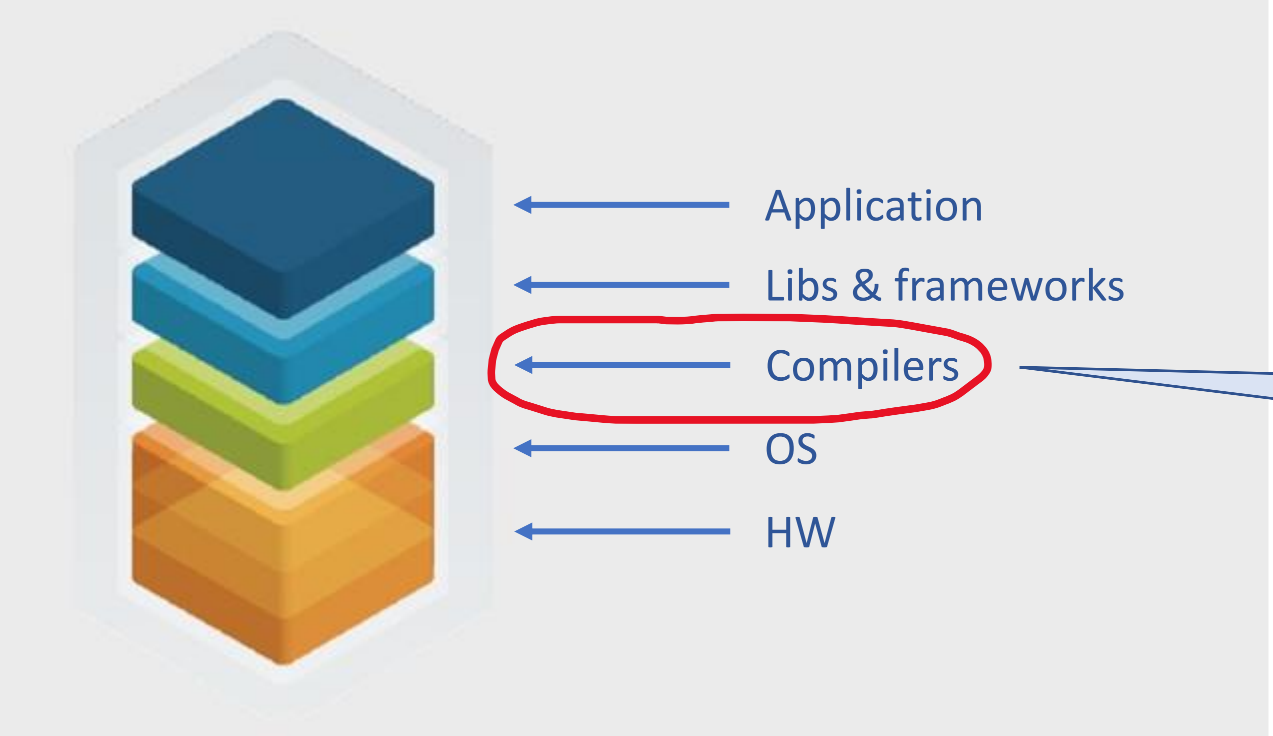
Technology	Software	Algorithms	Hardware architecture
Opportunity	Software performance engineering	New algorithms	Hardware streamlining
Examples	Removing software bloat	New problem domains	Processor simplification
	Tailoring software to hardware features	New machine models	Domain specialization

The Bottom

for example, semiconductor technology

from: “There is Plenty of Room at the Top”, Leiserson et. al

We are on the spot



Application

Libs & frameworks

Compilers

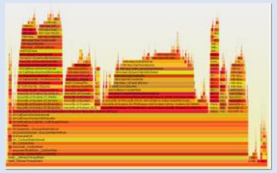
OS

HW

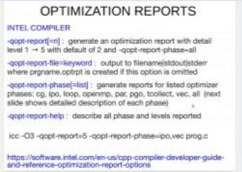
That’s us!

My book for SW devs

Part 1. Analysis



Flamegraphs



OPTIMIZATION REPORTS

INTEL COMPILER

-qopt-report[=n] : generate an optimization report with detail level 1 → 5 with default of 2 and -qopt-report-phase=all

-qopt-report-file=keyword : output to filename|stdout|stderr where prgname.optrpt is created if this option is omitted

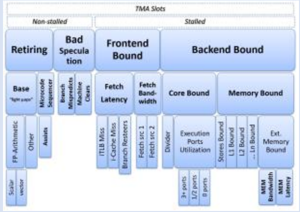
-qopt-report-phase[=list] : generate reports for listed optimizer phases: cg, ipo, loop, openmp, par, pgo, tcollect, vec, all (next slide shows detailed description of each phase)

-qopt-report-help : describe all phase and levels reported

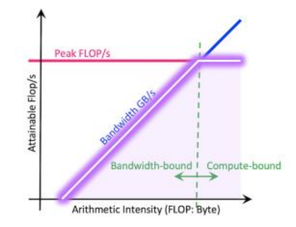
```
icc -O3 -qopt-report=5 -qopt-report-phase=ipo,vec prog.c
```

https://software.intel.com/en-us/cpp-compiler-developer-guide-and-reference-optimization-report-options

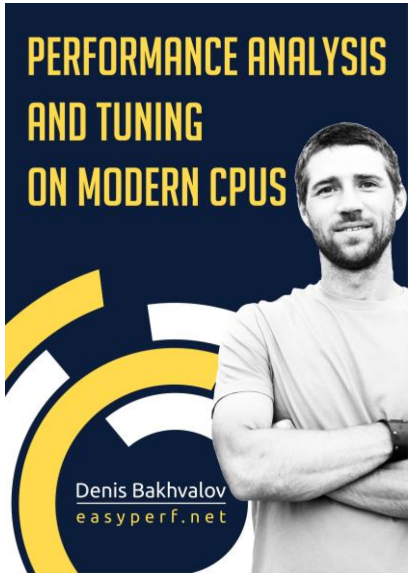
Compiler opt remarks



HW-specific



Roofline



book.easyperf.net/perf_book

(PDF version available for free)

Part 2. Tuning

PGO

Optimize memory accesses

Vectorization

Optimize code layout

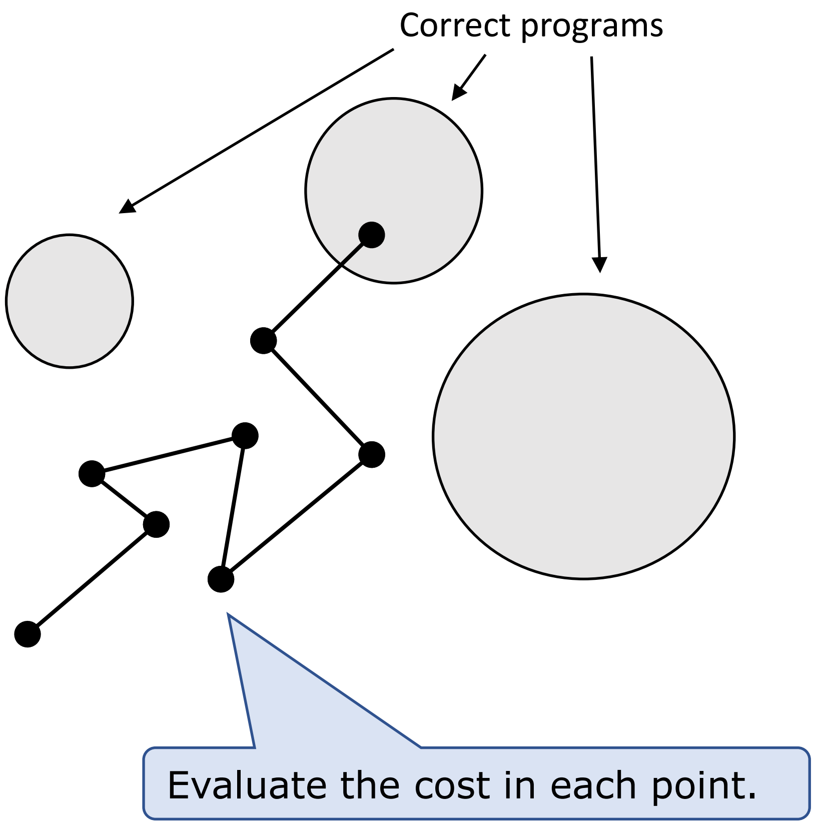
Eliminate branch mispredictions

Function inlining

aka "beat the compiler"

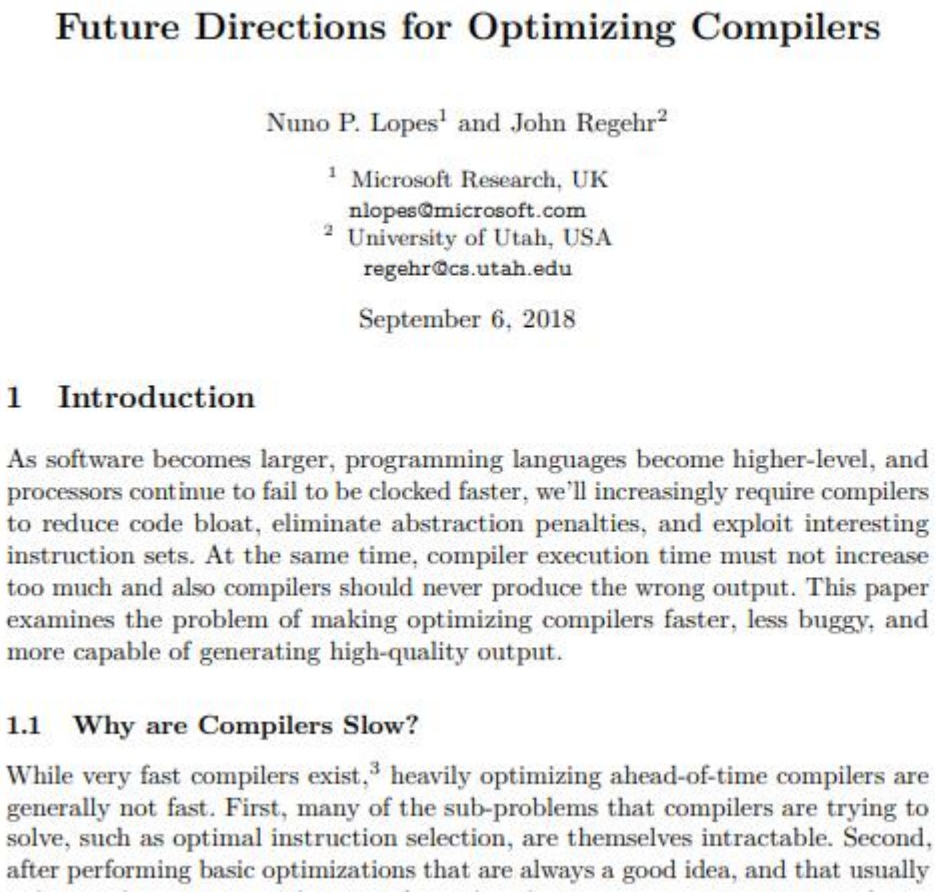
How can we further improve the performance of the code that we generate?

Synthesizing Superoptimizers



Evaluate the cost in each point.

[1]: STOKE: https://arxiv.org/abs/1211.0557
[2]: Souper: https://arxiv.org/abs/1711.04422



Future Directions for Optimizing Compilers

Nuno P. Lopes[1] and John Regehr[2]

[1] Microsoft Research, UK
nlopes@microsoft.com
[2] University of Utah, USA
regehr@cs.utah.edu

September 6, 2018

1 Introduction

As software becomes larger, programming languages become higher-level, and processors continue to fail to be clocked faster, we'll increasingly require compilers to reduce code bloat, eliminate abstraction penalties, and exploit interesting instruction sets. At the same time, compiler execution time must not increase too much and also compilers should never produce the wrong output. This paper examines the problem of making optimizing compilers faster, less buggy, and more capable of generating high-quality output.

1.1 Why are Compilers Slow?

While very fast compilers exist,[3] heavily optimizing ahead-of-time compilers are generally not fast. First, many of the sub-problems that compilers are trying to solve, such as optimal instruction selection, are themselves intractable. Second, after performing basic optimizations that are always a good idea, and that usually

Decoupling Transform Passes

a.ll ⇨ b.ll

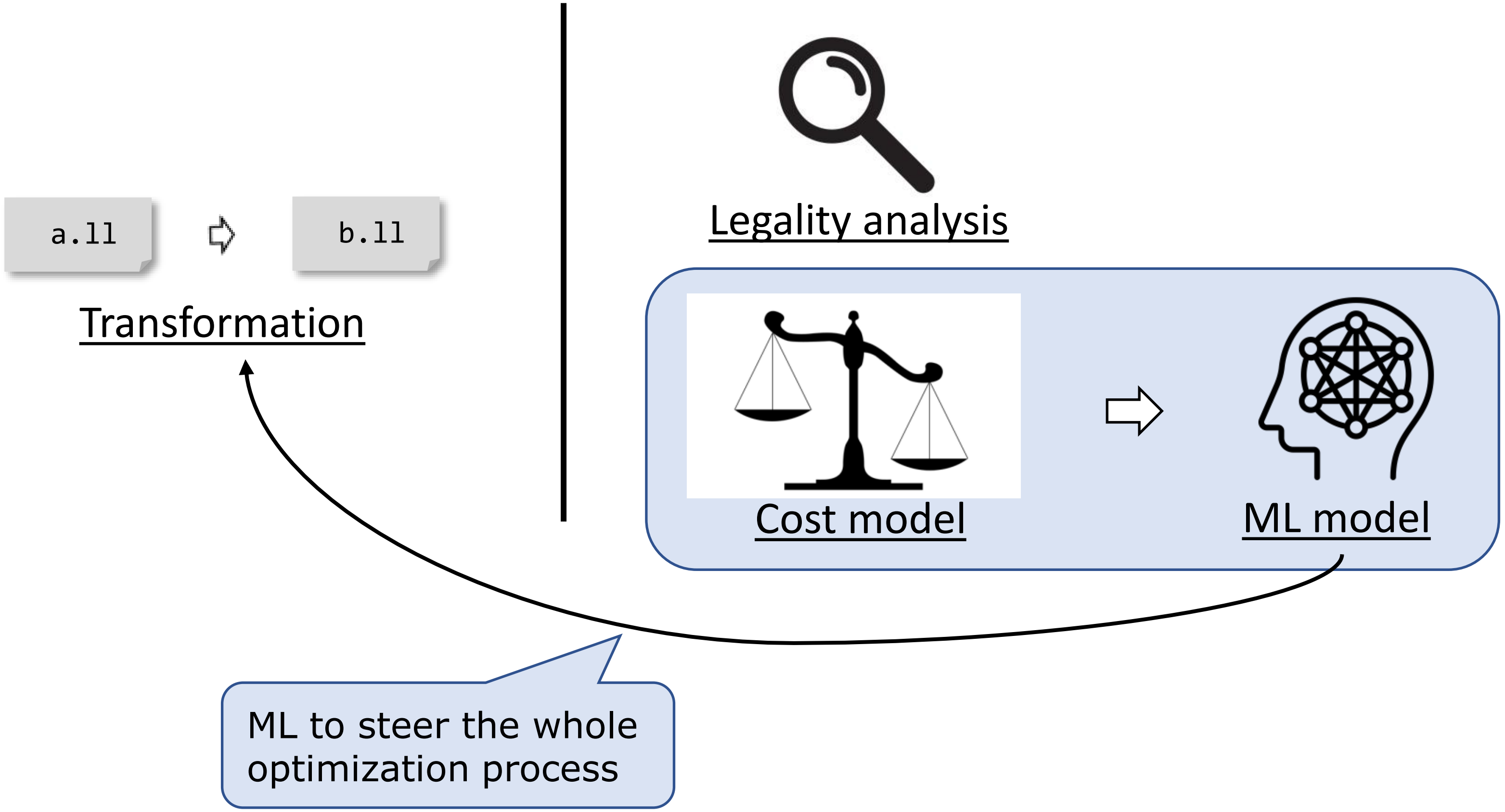
Transformation

Legality analysis

Cost model

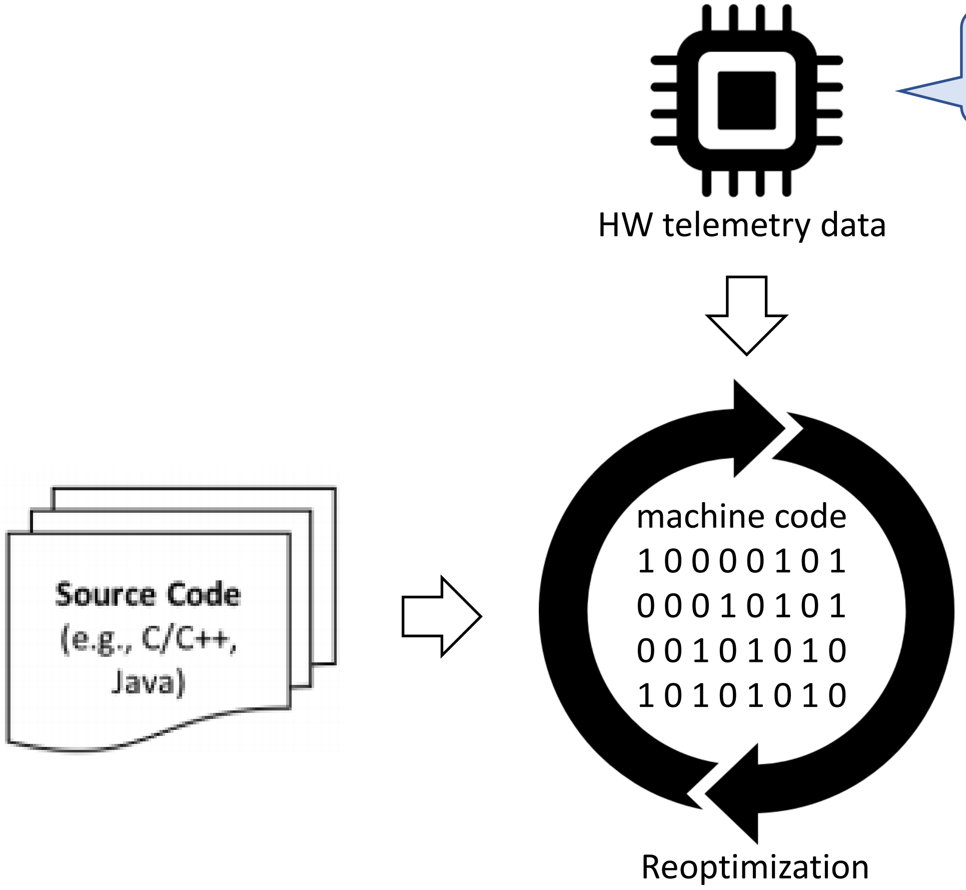
Separate the transformation from the rest of the pass.

Machine Learning models



ML to steer the whole optimization process

Autotuning, PGO



HW telemetry data gets more advanced

Potentially more passes could be guided by profiling data

[1]: H. Finkel, JITting in C++: https://www.youtube.com/watch?v=pDagqR0jAvQ

Optimizing Libraries & Abstractions

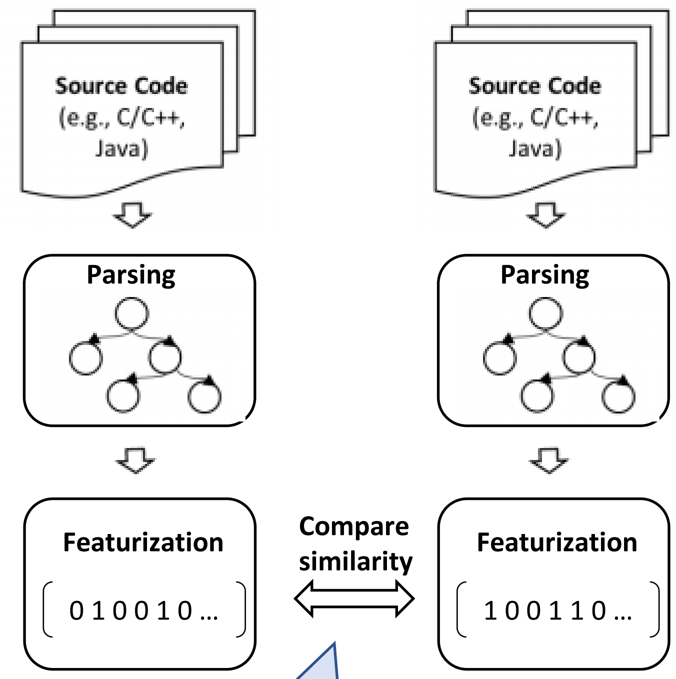
```
std::vector<int> a;
for (int i = 0; i < 1000; i++)
  a.push_back(x + i);
```

Could have call a.reserve() ahead of time

“We end up optimizing C++ as if it was C (analyze pointer and calls).”

Helping Developers

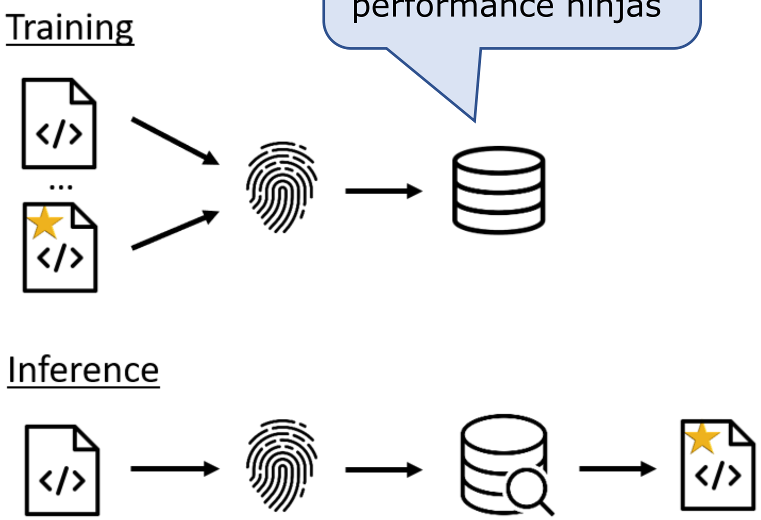
Code similarity analysis



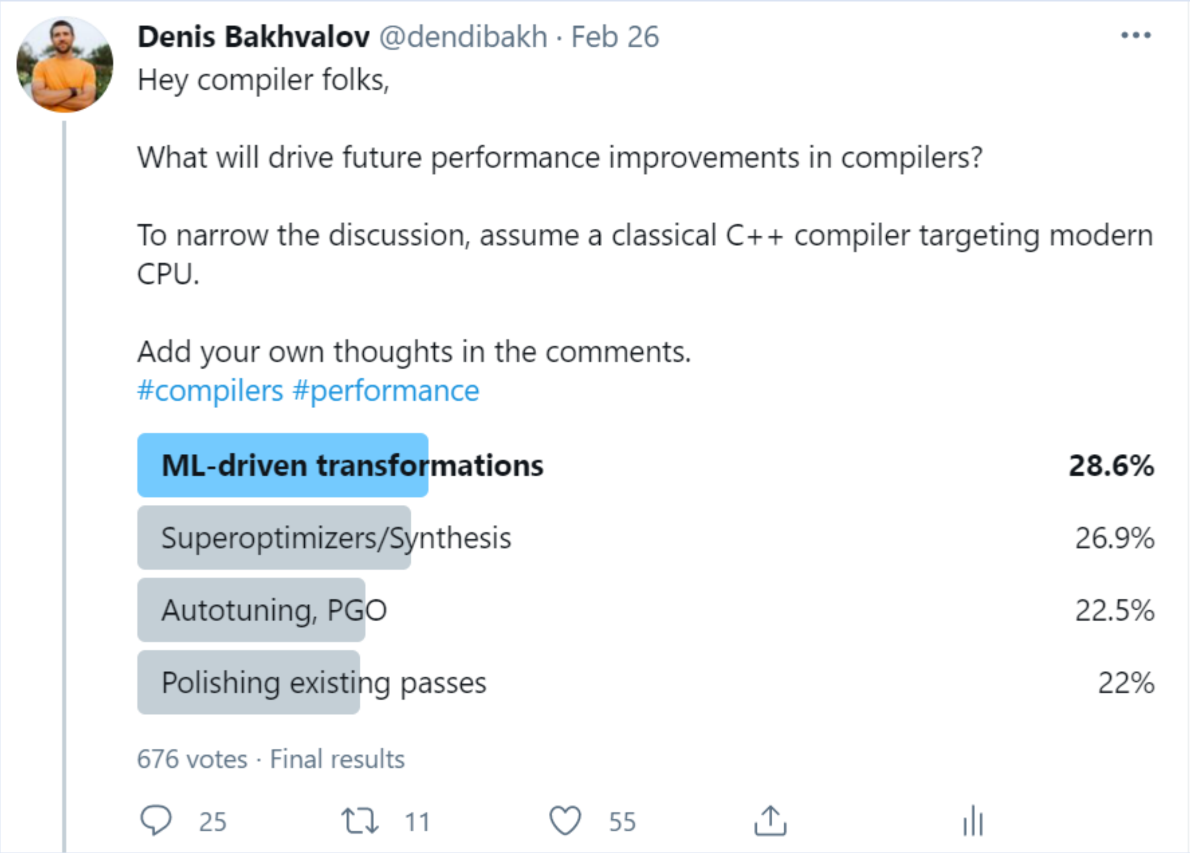
Solvers can further prove equality

Enables

Build a database of code written by performance ninjas



“Hey, we think you’re writing a matrix-multiply code. Dou you want to reuse the code written by experts? Here is the diff…”



Denis Bakhvalov @dendibakh · Feb 26

Hey compiler folks,

What will drive future performance improvements in compilers?

To narrow the discussion, assume a classical C++ compiler targeting modern CPU.

Add your own thoughts in the comments.
#compilers #performance

ML-driven transformations **28.6%**

Superoptimizers/Synthesis 26.9%

Autotuning, PGO 22.5%

Polishing existing passes 22%

676 votes · Final results

Challenges

- Verification of ML models.
- Compile-time tradeoffs [1].
- Changing LLVM is hard.

[1]: Current clang vs. 10-year old clang: 10% better runtime performance but 2x slower compiler-time.
https://gist.github.com/zeux/3ce4fcc3a43072b4315abde95319ecb6

Takeaways

- The free lunch for SW vendors is over. SW tuning will become one the major drivers for performance improvements. Obviously, compilers could and should help with SW tuning since not every developer is a performance ninja.

- Key areas for future compiler optimizations:
  - Replacing cost models and heuristics with ML.
  - Search-based approaches (superoptimizers and synthesizers).
  - Autotuning, Reooptimization, PGO.
  - Doing a better job at optimizing libraries.
  - Help developers solve performance issues.

Thank you

@dendibakh　@dendibakh　easyperf.net