Lost in Translation:
Claiming Back Hidden Potential of Compilers

Kyriakos Georgiou
Zbigniew Chamski
Kerstin Eder
Andres Amaya Garcia
David May

University of Bristol

1st ISC Workshop on LLVM, Frankfurt/Main, Germany, June 25-26 2020

TEAMPLAY

Time, Energy and security Analysis for Multi/Many-core heterogeneous PLAtforms

University of BRISTOL

In an ideal world. . .

. . . compilers do a perfect job:

- they find the global optimum of the preferred metric(s):
  - performance
  - code size
  - energy, etc.
  - combinations of the above
- they are bug-free
- they preserve all necessary debug/traceability information
- they run fast

(and compiler engineers are out of their jobs)

Reality is quite different

Compilers are big and complex systems:

- 3.3..3.8 MLOC (net Million Lines Of Code, without blank lines nor comments)
- 1100..1300 MY (man-years) of development effort
- ∼250+ successive passes
- quite some correctness bugs (“space”/“nuclear” quality ≈ 1 bug per 1 MLOC)
- lots of **non-functional** shortcomings wrt. performance, code size, energy, etc.

Correctness issues: handled by quality assurance

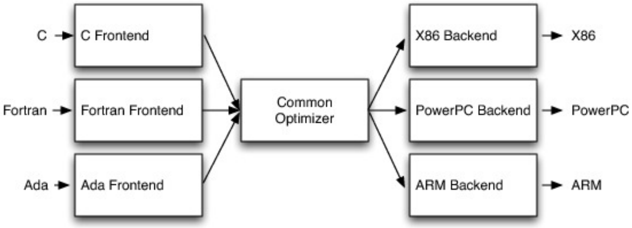
- code reviews
- unit tests
- nightly regression tests

But… **quality** also requires addressing **non-functional** properties!

Stakeholders of compiler improvement

	perspective	**aim**	**process**
compiler developers	● one compiler ● many configurations	● correctness ● flexibility ● efficiency	● fix issues ● tune settings ● refactor/improve
compiler users	● one class of applications ● one compiler	● increase performance ● reduce energy bill	● determine best settings ● adapt to shortcomings
managers	● ROI from assets: staff, HW, SW	● reduce lead time ● increase performance & throughput	● optimize costs ● improve efficiency of staff and resources

Structure of an LLVM-based compiler



http://blog.llvm.org/2013/04/static-analysis-tools-using-clang-in.html

A rough split of code size and development effort[1], LLVM 10.0.0:

- language front-ends: 1500 KLOC, 460 MY (man-years)
- **common, target-independent optimizer: 200 KLOC, 45 MY**
- target backends: 550 KLOC, 150 MY

---

[1] Estimates generated using David A. Wheeler's 'SLOCCount'.

Requirements for systematic optimizer tuning

How to spot **opportunities** inside 200 KLOC of critical code?

- Visual inspection? Too difficult
- Try all combinations of options and parameters? Too complex on daily basis: $\approx (2^{65}) \times$ a_lot combinations, *not counting pass reordering*
- Divide and ~~conquer~~ examine!

We need a method that is

- **portable** between compiler releases and supported targets,
- **agile** to be usable in the day-to-day development flow,
- **versatile** to uncover issues common across multiple use cases / targets,
- **insightful** to simplify the isolation of root causes of deficiencies

The (bug) train analogy

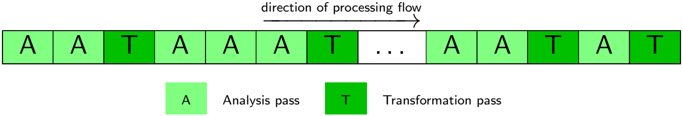
Photo by Drew Jacksich, https://www.flickr.com/photos/28101583@N07/5003755493/, CC BY 2.0

Start from the beginning, and successively check “where appropriate”

- **Portable? Yes**, to any sequential structure
- **Agile? Yes**, linear complexity
- **Versatile? Yes**, works with any load, any car type
- **Insightful? Yes**, spots the defect as soon as it is reached

Can we automate it?

Systematic tracking of optimizer behavior

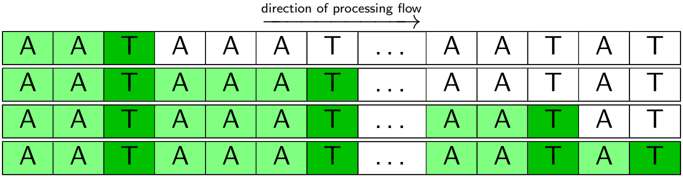


For each benchmark of interest, compile that benchmark using successive prefixes of the full optimizer pass sequence up to a transformation pass, inclusive.

- select an **optimization level** (-O2, -O3, -Os, -Oz, etc.)
- determine the **pass sequence** applied by that optimization level
- results achieved using the **full pass sequence** are the baseline
- results achieved using the **successive prefixes** of the full pass sequence directly expose improvements/degradations

Generation and interpretation of measurements

- compile and run the benchmark using successive “prefix configurations” of the optimizer



A	A	T	A	A	A	T	...	A	A	T	A	T
A	A	T	A	A	A	T	...	A	A	T	A	T
A	A	T	A	A	A	T	...	A	A	T	A	T
A	A	T	A	A	A	T	...	A	A	T	A	T

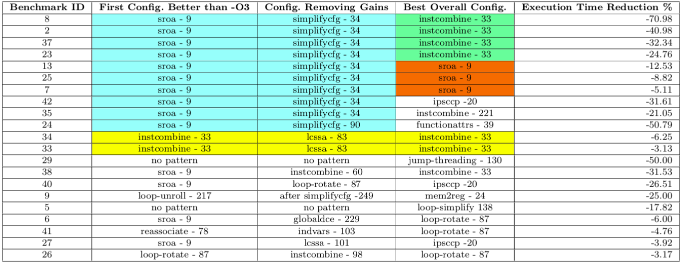
- check correctness of result: **incorrect executions** signal *optimizer bugs!*
- **absolute improvements** (those relative to the baseline) signal *missed opportunities*
- **incremental degradations** (relative to the previous prefix) indicate *potentially counterproductive* transformations

Integration into day-to-day development flow

1. Prerequisites:
   - Availability of representative benchmarks
   - Processing time to generate the successive variants of each benchmark
   - Space to store the generated executables and the intermediate files
   - Ability to run-and-measure each individual executable (not so trivial for some embedded applications)
2. Exploration phase (automatable):
   - Build individual executables of each benchmark, saving intermediate files
   - Run individual executables of each benchmark and collect runtime data (output correctness, metrics, traces. . .)
   - Process collected data to determine incremental and absolute changes in metrics
3. Analysis phase (mostly manual):
   - Analyze trends/patterns in the changes observed
   - Act upon the results of the analysis

Example condensed report

- platform: i5-6300U running Ubuntu 18.04 LTS
- compiler: LLVM 6.0
- baseline optimization level: -O3
- value tracked: best absolute time improvement wrt. baseline (threshold -3%)

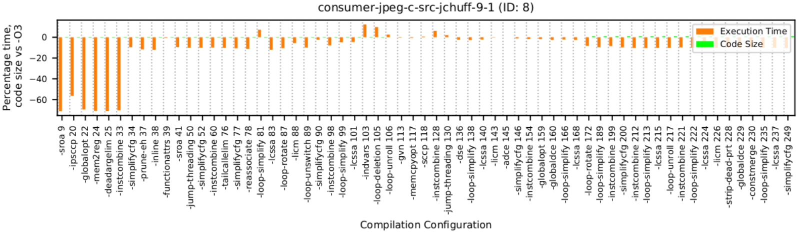


Benchmark ID	First Config. Better than -O3	Config. Removing Gains	Best Overall Config.	Execution Time Reduction %
8	sroa - 9	simplifycfg - 34	instcombine - 33	-70.98
2	sroa - 9	simplifycfg - 34	instcombine - 33	-40.98
37	sroa - 9	simplifycfg - 34	instcombine - 33	-32.34
23	sroa - 9	simplifycfg - 34	instcombine - 33	-24.76
13	sroa - 9	simplifycfg - 34	sroa - 9	-12.53
25	sroa - 9	simplifycfg - 34	sroa - 9	-8.82
7	sroa - 9	simplifycfg - 34	sroa - 9	-5.11
42	sroa - 9	simplifycfg - 34	ipsccp -20	-31.61
35	sroa - 9	simplifycfg - 34	instcombine - 221	-21.05
24	sroa - 9	simplifycfg - 90	functionattrs - 39	-50.79
34	instcombine - 33	lcssa - 83	instcombine - 33	-6.25
33	instcombine - 33	lcssa - 83	instcombine - 33	-3.13
29	no pattern	no pattern	jump-threading - 130	-50.00
38	sroa - 9	instcombine - 60	instcombine - 33	-31.53
40	sroa - 9	loop-rotate - 87	ipsccp -20	-26.51
9	loop-unroll - 217	after simplifycfg -249	mem2reg - 24	-25.00
5	no pattern	no pattern	loop-simplify 138	-17.82
6	sroa - 9	globaldce - 229	loop-rotate - 87	-6.00
41	reassociate - 78	indvars - 103	loop-rotate - 87	-4.76
27	sroa - 9	lcssa - 101	ipsccp -20	-3.92
26	loop-rotate - 87	instcombine - 98	loop-rotate - 87	-3.17

Pass `simplifycfg` appears to be a major performance killer!

Fine-grain analysis: review incremental changes

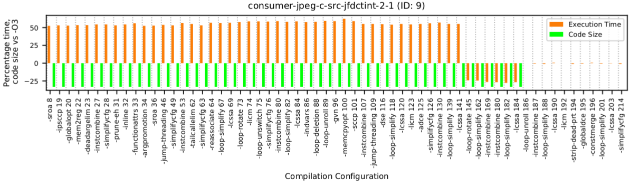
- platform: i5-6300U running Ubuntu 18.04 LTS
- compiler: LLVM 6.0
- baseline optimization level: -O3
- value tracked: absolute time and code size change wrt. baseline



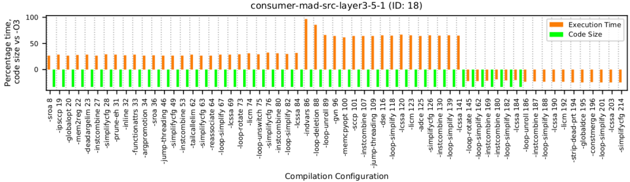
- pass `simplifycfg` (index 34) cancels a huge potential gain $\Longrightarrow$ deeper investigation needed
- actual root cause: overly aggressive if-conversion in tight loops

Analysis of transformation interactions

- Arm Cortex-A53 running Raspian, LLVM 6.0, baseline optimization level: -O3



consumer-jpeg-c-src-jfdctint-2-1 (ID: 9)



consumer-mad-src-layer3-5-1 (ID: 18)

- `loop-unroll 186` can remove both code size and performance potential
- lower graph: an artefact of our framework unlocked a hidden 25% perf gain!

Exploitation of findings from analysis phase

Detection, identification and elimination of bugs related to

- correctness (output validity is checked at each run)
- performance degradations, both overall and between passes
- energy usage
- code size (influence on performance!)

both target-specific and cross-target ones

Tuning of compiler heuristics towards specific workloads / target features:

- transformation thresholds
- default parameter values

Relationship to Machine Learning and ML-based compiler tuning

Applications of Machine Learning in our approach:

- Detection of patterns in nightly reports
- Classification/prioritization of identified patterns

Our approach in comparison to ML-based compiler tuning:

- different goals: feedback to developers vs. quest for the best
- (—) deliberately constrained search space: will miss gains from pass reordering/duplication etc.
- (+) predictable complexity improves day-to-day usability
- (+) clearbox approach: findings easily related to compiler source code
- (+) serves as a general compiler debugging aid

Future work

- Deploy our approach "in the field" at LLVM power users
- Set up an automated "Hidden Potential Tracking" service for LLVM
- Apply the approach to other compilers, in particular GCC (use pass control interface in the compiler)
- Apply the approach to speed up the tuning of support for new targets (RISC-V...)

Take Home Message

Our approach brings benefits to all stakeholders of compiler development:

- for **compiler developers**
  - direct insight into untapped potential of the compiler
  - potential functional/non-functional bugs identified early and precisely
  - fast turnaround time in improving the compiler
- for **compiler users**
  - way of unlocking the hidden potential of current tools
  - vehicle for precise and targeted feedback to compiler developers
- for **managers**
  - increased productivity of compiler and application teams
  - cost reductions in running HPC applications (performance, energy)

Thank you!

Questions?

Zbigniew.Chamski@bristol.ac.uk
Kerstin.Eder@bristol.ac.uk
Kyriakos.Georgiou@bristol.ac.uk

Further reading:

K. Georgiou, Z. Chamski, A. Amaya Garcia, D. May and K. Eder. *Lost in translation: Exposing hidden compiler optimization opportunities*. To appear in the The Computer Journal (submitted March 2019, revised March 2020, accepted June 2020). Preprint available from https://arxiv.org/abs/1903.11397.

K. Georgiou, C. Blackmore, S. Xavier-de-Souza, and K. Eder. 2018. *Less is More: Exploiting the Standard Compiler Optimization Levels for Better Performance and Energy Consumption*. In Proceedings of the 21st International Workshop on Software and Compilers for Embedded Systems (SCOPES'18). ACM, pages 35–42.
DOI: https://doi.org/10.1145/3207719.3207727

This work was partially supported by the EU H2020 TeamPlay project under grant agreement ID 779882.