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
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
- \( \sim 250+ \) successive passes
- quite some correctness bugs ("space"/"nuclear" quality \( \approx 1 \) 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

<table>
<thead>
<tr>
<th></th>
<th>perspective</th>
<th>aim</th>
<th>process</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>compiler developers</strong></td>
<td>• one compiler&lt;br&gt; • many configurations</td>
<td>• correctness&lt;br&gt; • flexibility&lt;br&gt; • efficiency</td>
<td>• fix issues&lt;br&gt; • tune settings&lt;br&gt; • refactor/improve</td>
</tr>
<tr>
<td><strong>compiler users</strong></td>
<td>• one class of applications&lt;br&gt; • one compiler</td>
<td>• increase performance&lt;br&gt; • reduce energy bill</td>
<td>• determine best settings&lt;br&gt; • adapt to shortcomings</td>
</tr>
<tr>
<td><strong>managers</strong></td>
<td>• ROI from assets: staff, HW, SW</td>
<td>• reduce lead time&lt;br&gt; • increase performance &amp; throughput</td>
<td>• optimize costs&lt;br&gt; • improve efficiency of staff and resources</td>
</tr>
</tbody>
</table>
Structure of an LLVM-based compiler

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:** $\approx (2^{65}) \times \text{a lot combinations, not counting pass reordering}$
- **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

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?
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%)

<table>
<thead>
<tr>
<th>Benchmark ID</th>
<th>First Config.</th>
<th>Better than -O3</th>
<th>Config. Removing Gains</th>
<th>Best Overall Config.</th>
<th>Execution Time Reduction %</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>sroa - 9</td>
<td>simplifycfg - 34</td>
<td>instcombine - 33</td>
<td>-76.98</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>sroa - 9</td>
<td>simplifycfg - 34</td>
<td>instcombine - 33</td>
<td>-40.98</td>
<td></td>
</tr>
<tr>
<td>37</td>
<td>sroa - 9</td>
<td>simplifycfg - 34</td>
<td>instcombine - 33</td>
<td>-33.34</td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>sroa - 9</td>
<td>simplifycfg - 34</td>
<td>instcombine - 33</td>
<td>-24.76</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>sroa - 9</td>
<td>simplifycfg - 34</td>
<td>sroa - 9</td>
<td>-12.53</td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>sroa - 9</td>
<td>simplifycfg - 34</td>
<td>sroa - 9</td>
<td>-8.82</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>sroa - 9</td>
<td>simplifycfg - 34</td>
<td>sroa - 9</td>
<td>-5.11</td>
<td></td>
</tr>
<tr>
<td>42</td>
<td>sroa - 9</td>
<td>simplifycfg - 34</td>
<td>ipsc - 20</td>
<td>-31.61</td>
<td></td>
</tr>
<tr>
<td>35</td>
<td>sroa - 9</td>
<td>simplifycfg - 34</td>
<td>instcombine - 221</td>
<td>-21.05</td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>sroa - 9</td>
<td>simplifycfg - 90</td>
<td>functionattrs - 39</td>
<td>-50.79</td>
<td></td>
</tr>
<tr>
<td>34</td>
<td>instcombine - 33</td>
<td>lose - 83</td>
<td>instcombine - 33</td>
<td>-4.25</td>
<td></td>
</tr>
<tr>
<td>33</td>
<td>instcombine - 33</td>
<td>lose - 83</td>
<td>instcombine - 33</td>
<td>-3.13</td>
<td></td>
</tr>
<tr>
<td>29</td>
<td>no pattern</td>
<td>no pattern</td>
<td>jump-threathing - 130</td>
<td>-50.00</td>
<td></td>
</tr>
<tr>
<td>38</td>
<td>sroa - 9</td>
<td>instcombine - 60</td>
<td>instcombine - 33</td>
<td>-31.53</td>
<td></td>
</tr>
<tr>
<td>40</td>
<td>sroa - 9</td>
<td>loop-rotate - 87</td>
<td>ipsc - 20</td>
<td>-25.41</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>loop-unroll - 217</td>
<td>after simplifycfg - 249</td>
<td>mem2reg - 24</td>
<td>-25.00</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>no pattern</td>
<td>no pattern</td>
<td>loop-simplify - 138</td>
<td>-17.82</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>sroa - 9</td>
<td>globalize - 229</td>
<td>loop-rotate - 87</td>
<td>-6.00</td>
<td></td>
</tr>
<tr>
<td>41</td>
<td>reassociate - 78</td>
<td>indvars - 103</td>
<td>loop-rotate - 87</td>
<td>-4.76</td>
<td></td>
</tr>
<tr>
<td>27</td>
<td>sroa - 9</td>
<td>less - 101</td>
<td>ipsc - 20</td>
<td>-3.92</td>
<td></td>
</tr>
<tr>
<td>26</td>
<td>loop-rotate - 87</td>
<td>instcombine - 98</td>
<td>loop-rotate - 87</td>
<td>-3.17</td>
<td></td>
</tr>
</tbody>
</table>

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 ⇒ 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

  - 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:


DOI: https://doi.org/10.1145/3207719.3207727

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