Genetic Improvement of OLC and H3 with Magpie

Getting more out of Magpie
William Langdon (UCL) and Brad Alexander (Optimatics)
Motivation (1)

• Genetic Improvement (GI) works!
  • Fixing Bugs, Porting Code, Improving Speed, Reducing memory and energy consumption....
  • Real, verifiable, improvements in real software.

• But a lot of this work used bespoke tooling
  • Hard to set up
  • Hard to transfer results
Motivation (2)

• Frameworks have been produced to make GI easier

• Examples
  • GIN – flexible GI for Java applications.
  • PyGGI – Python tool for GI in multiple languages
  • Magpie – Modular, flexible GI targeting multiple languages

• The tools are available – and they are getting better
  • but we need a deeper body of practice in using and improving them
This work

• First use of Magpie on industrial source code
  • Google’s OLC and Uber’s H3
  • Improved program performance by changing C source code.
  • We changed tooling for running and measuring program performance
  • Speedup is better than previous work targeting LLVM IR.
Magpie

- **Machine Automated General Performance Improvement via Evolution of software**
- Developed from **PyGGI 2.0**
- Separates *search* from *operations*
  - In our case search is *local search*
    - proven to be effective
  - Operations can be in different domains.
    - Examples: compiler optimization options, runtime configurations, and **Genetic Improvement**
Target Applications

- Google OLC
- Uber H3
- Both do coordinate translation
Application Size

• OLC
  • 14024 lines of code
  • 207 to be optimized
  • 134 with comments/blanks removed

• H3
  • 15015 lines of code
  • 3321 to be optimized
  • 1615 with comments/blanks removed
Setup

• Set up to do GI on source code
• Using GI per-line operators
• Optimized for execution speed
• Applied tests (10 cases)
  • + checks for correctness
• Runtime chosen to ensure coverage
• Used Hill Climbing for search
  • keeps it simple!
Reducing Noise

• Noise can really slow down evolution
• Need to adapt measurements to take account of the noise distribution...
Noise Distributions (1)

• Wall clock time is noisy
• And is heavily dependent on the run-order of sample

Earlier repetitions of tests for triangle (in red) are much slower!
Noise Distributions (2)

• Runtimes are heavily skewed and tightly bounded from below

wall-clock run time for OLC test cases

distribution skewed towards minimum run time for each case

long tail of long-running readings

can’t distinguish test cases by using run times
Noise Distributions (3)

- Instruction counts show much less noise.
  - used `PERF_COUNT_HW_INSTRUCTIONS`
  - less noise => less samples needed
- We use only 3 samples per test and sample the lowest quartile of all the tests.

using CPU instructions, it is easy to distinguish tests just by their runtime!

Symbols denote tests with very small timing spreads.
Other tricks

• More warmup evaluations on null patches
• Wrote harness for measuring instruction counts in C.
• Called harness directly using Python’s c-types interface
• Output directed from harness to a buffer provided by python.
Results (1)

- Tested both evolved OLC and H3 variants
  - with and without GNU compiler -O3 flag
- Good speedups for both
- Passed all holdout tests

<table>
<thead>
<tr>
<th>C files</th>
<th>LOC no comments</th>
<th>size</th>
<th>Mutant minified</th>
<th>speed up</th>
<th>Magpie duration</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>207 (134)</td>
<td>4–7</td>
<td>4–7</td>
<td>3.6%</td>
<td>82 secs</td>
</tr>
<tr>
<td>-O3</td>
<td>207 (134)</td>
<td>8–13</td>
<td>6–11</td>
<td>2%</td>
<td>95 secs</td>
</tr>
<tr>
<td>23</td>
<td>3321 (1615)</td>
<td>31–45</td>
<td>22–28</td>
<td>15%</td>
<td>1.1 hours</td>
</tr>
<tr>
<td>-O3</td>
<td>3321 (1615)</td>
<td>31–49</td>
<td>23–29</td>
<td>7%</td>
<td>1.5 hours</td>
</tr>
</tbody>
</table>
Results (2)

- Pass rates – most variants generated during search passed.
Results (3)

• Large inter-run variation – due to local search?
Code produced

• Examples include:
  • Removing redundant normalization and checks
  • Removing code that supports code paths that aren’t executed

• Overlap with code specialisation?
Conclusions

• Magpie is easy to use and modify
• We were able to get useful and robust improvements.
• Measures to reduce noise are key
• Future work
  • Richer set of mutations + crossover
  • Move beyond hill climbing
  • Co-evolution of training data
  • Use profiling to focus search
Credits

• Thanks to
• Aymeric Blot (Magpie)
• H.Wierstorf (gnuplot)
• Funded by the Meta Oops project.
Contacts

• Bill Langdon - W.Langdon@cs.ucl.ac.uk
• Brad Alexander – bradley.alexander@optimatics.com

Questions?