Genetic Improvement of OLC and H3 with Magpie

Getting more out of Magpie

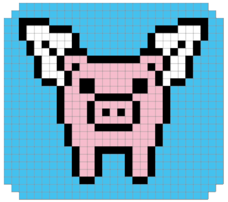
William Langdon (UCL) and Brad Alexander (Optimatics)

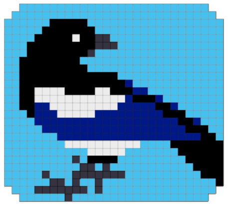
Motivation (1)

- 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)

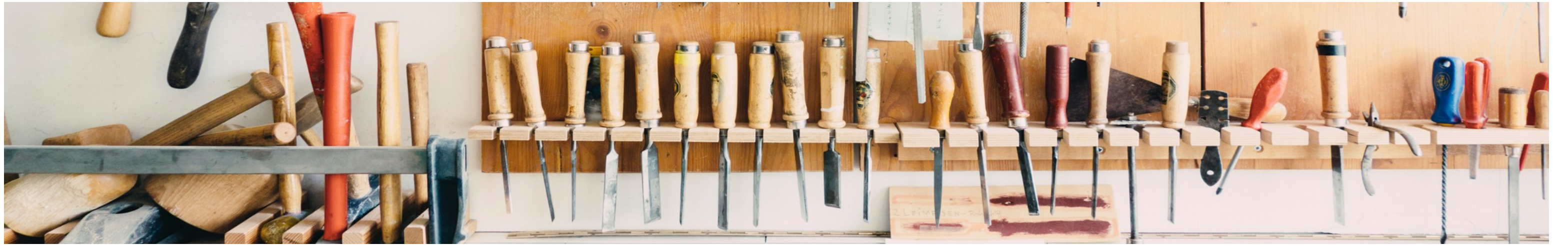
- Frameworks have been produced to make GI easier
- Examples
  - GIN – flexible GI for Java applications.
  - PyGGI – Python tool for GI in multiple languages
  - Magpie – Modular, flexible GI targetting multiple languages
- The tools are available – and they are getting better
  - but we need **a deeper body of practice** in using and improving them





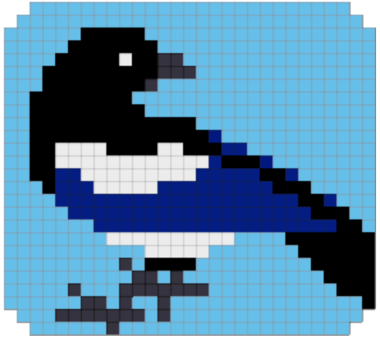
This work

- 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

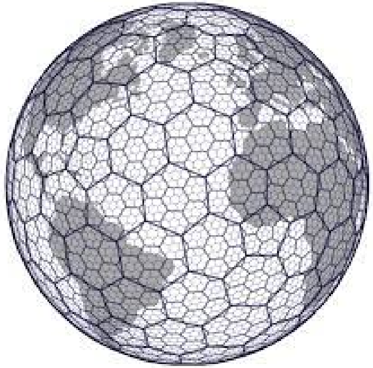
- **Machine Automated General Performance Improvement via Evolution of software**
- Developed from **PyGGI 2.0**
- Separates **search** from **operations**
    - In our case search is **local search**
        - proven to be effective
    - Operations can be in different domains.
        - Examples: compiler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

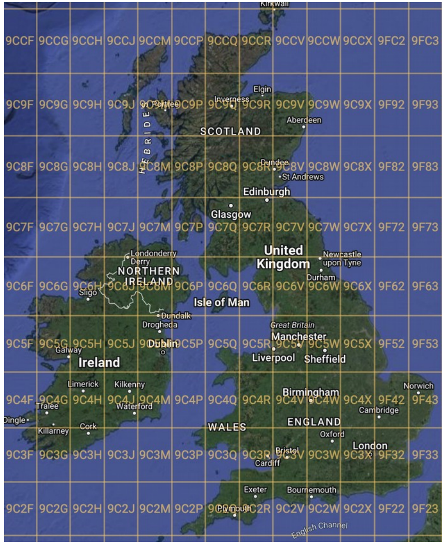


Target Applications

- Google OLC
- Uber H3
- Both do coordinate translation

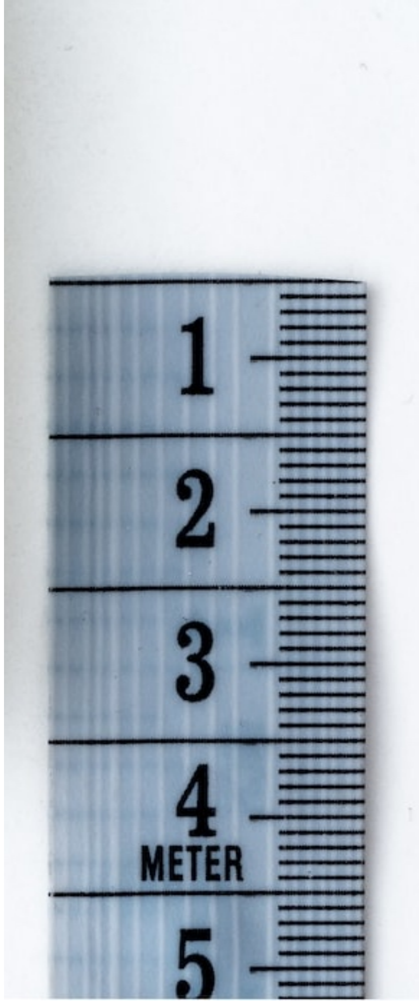
Plus Codes





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

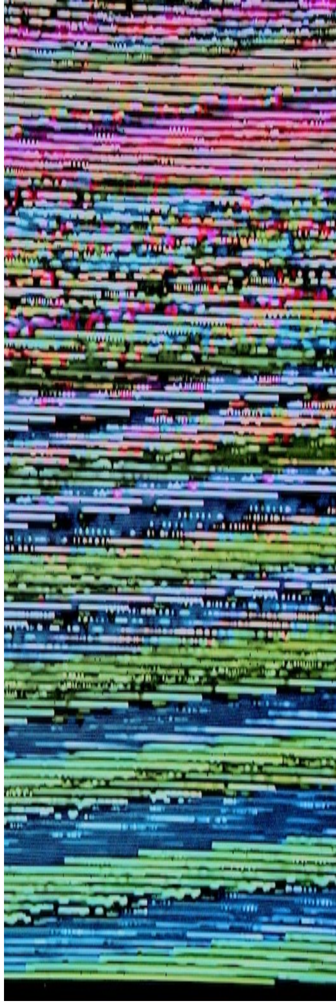


Setup

- 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

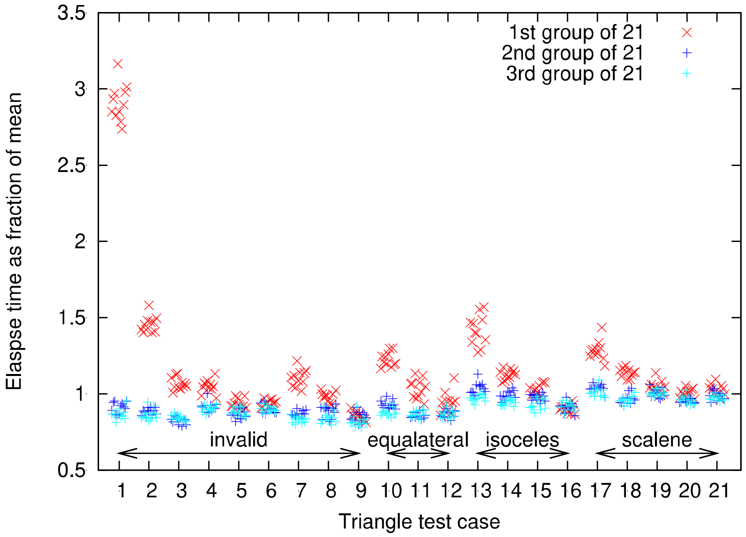
- Noise can really slow down evolution
- Need to adapt measurements to take account of the noise distribution...



Noise Distributions (1)

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

wall-clock run time for triangle program test cases



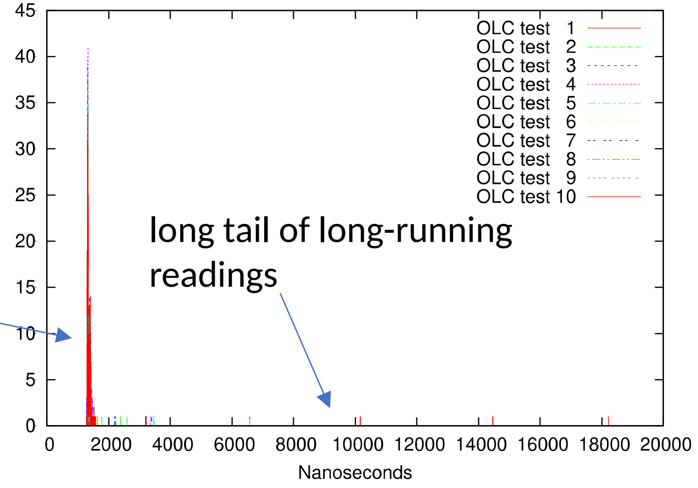
Earlier repetitions of tests for triangle (in red) are much slower!

Noise Distributions (2)

wall-clock run time for OLC test cases

- Runtimes are heavily skewed and tightly bounded from below

distribution skewed towards minimum run time for each case



long tail of long-running readings

can't distinguish test cases by using run times

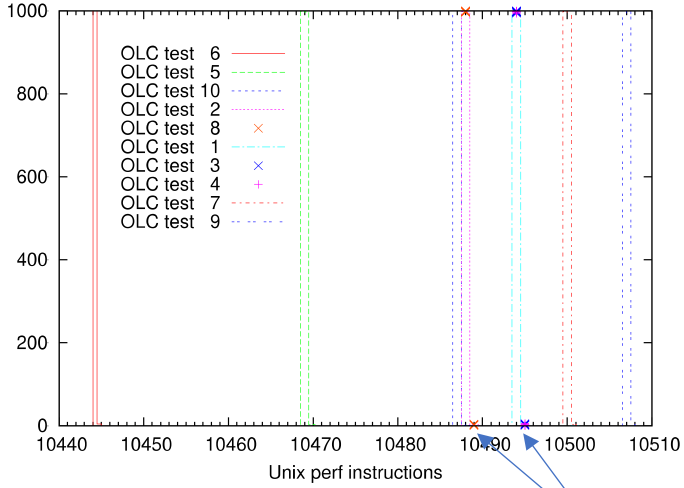
Noise Distributions (3)

- 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.

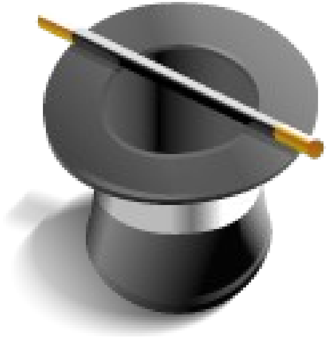
using CPU instructions, it is easy to distinguish tests just by their runtime!



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.



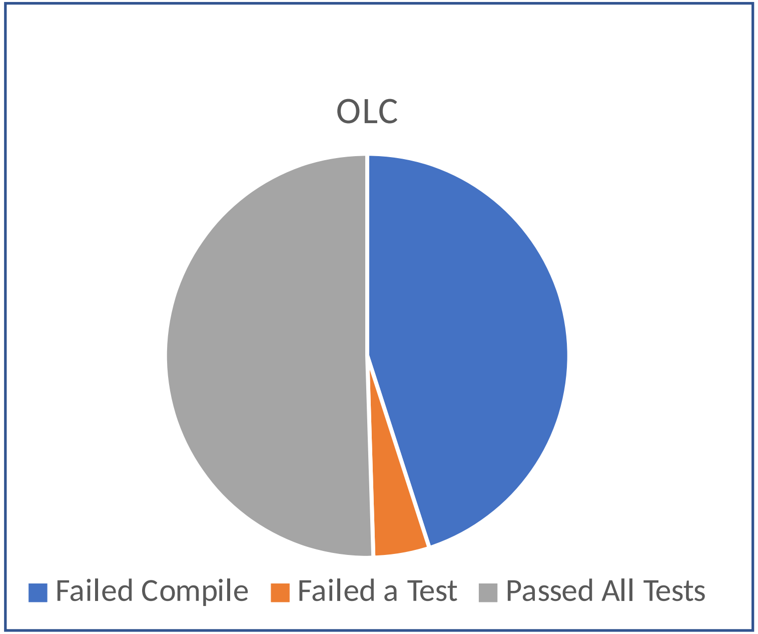
Results (1)

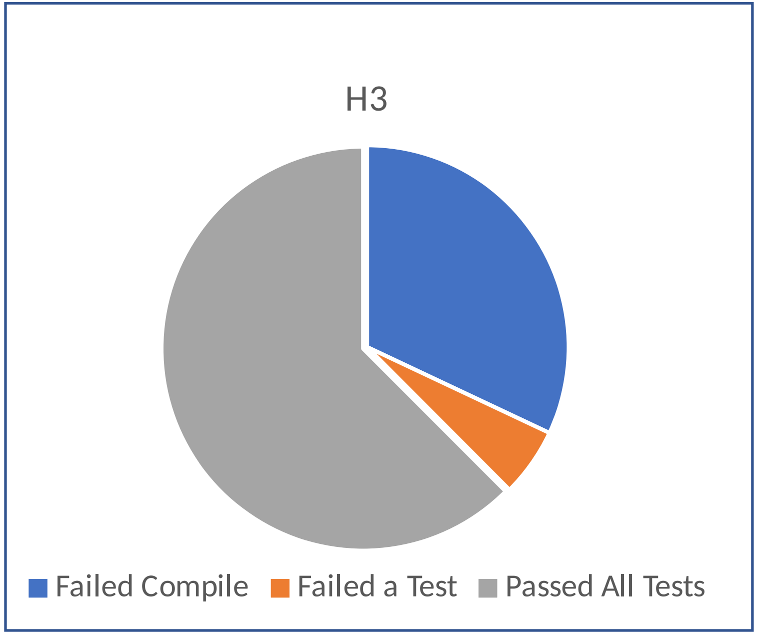
- Tested both evolved OLC and H3 variants
  - with and without GNU compiler -O3 flag
- Good speedups for both
- Passed all holdout tests

C	files	LOC		Mutant			Magpie
		no comments		size	minified	speed up	duration
OLC	4	207	(134)	4– 7	4– 7	3.6%	82 secs
-O3	4	207	(134)	8–13	6–11	2%	95 secs
H3	23	3321	(1615)	31–45	22–28	15%	1.1 hours
-O3	23	3321	(1615)	31–49	23–29	7%	1.5 hours

Results (2)

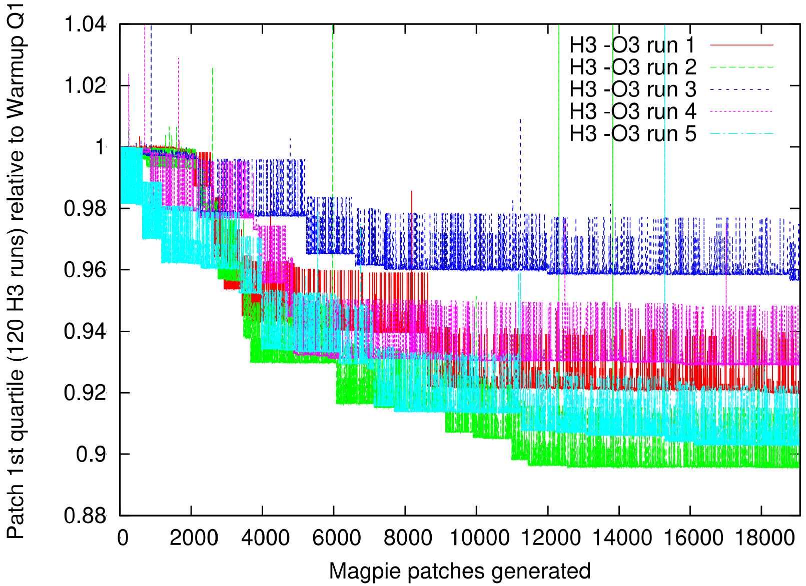
- Pass rates – most variants generated during search passed





Results (3)

- Large inter-run variation – due to local search?



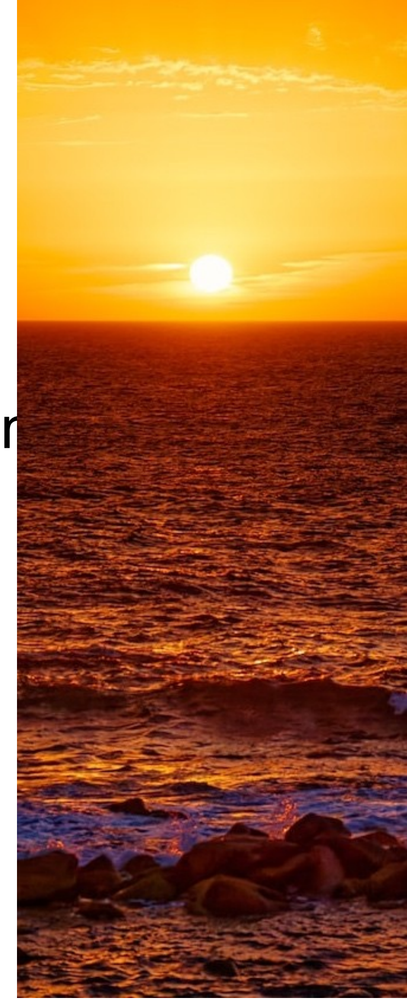
Code produced

- Examples include:
  - Removing redundant normalization and checks
  - Removing code that supports code paths that aren’t executed
- Overlap with code specialisation?

```
int OLC_Encode(const OLC_LatLon* location,
               int maxlen) {
  // Limit the maximum number of digits in
  if (length > kMaximumDigitCount) {
    length = kMaximumDigitCount;
  }
  // Adjust latitude and longitude so they
  double latitude = adjust_latitude(locati
  double longitude = normalize_longitude(l

  // Build up the code here, then copy it
  char fullcode[] = "12345678901234567";

  // Compute the code.
  // This approach converts each value to
  // the final precision. This allows us t
  // avoiding any accumulation of floating

  // Multiply values by their precision an
  // floating point operations.
  long long int lat_val = kLatMaxDegrees *
  long long int lng_val = kLonMaxDegrees *
  lat_val += latitude * 2.5e7;
  lng_val += longitude * 8.192e6;

  size_t pos = kMaximumDigitCount;
  // Compute the grid part of the code if
  if (length > kPairCodeLength) {
    for (size_t i = 0; i < kGridCodeLength
      int lat_digit = lat_val % kGridRows;
      int lng_digit = lng_val % kGridCols;
      int ndx = lat_digit * kGridCols + ln
```

Conclusions

- Magpie is easy to use and modify
- We were able to get useful and robust improvemen
- 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?