Genetic Improvement of OLC and H3 with Magpie

William B. Langdon
CREST, Computer Science, UCL
Gower Street, London, WC1E 6BT, UK
W.Langdon@cs.ucl.ac.uk

Bradley J. Alexander
Optimatics
33 Franklin St., Adelaide, SA 5000, Australia
bradalexa@gmail.com

***Abstract*—Magpie (Machine Automated General Performance Improvement via Evolution of software) has been recently developed by Aymeric Blot from PyGGI 2.0. Like PyGGI, it claims to be able to optimise computer source code written in arbitrary programming languages. So far it has been demonstrated on benchmarks written in Python and C. Recently we have used hill climbing to customise two industrial open source programs: Google's Open Location Code OLC and Uber's Hexagonal Hierarchical Spatial Index H3 [W. B. Langdon *et al.*, "Genetic improvement of LLVM intermediate representation", in *EuroGP* 2023]. Magpie found much faster improvements (reducing instruction counts by up to 15% v. 2%) which generalise. Various glitches in Magpie are also reported.**

***Index Terms*—Genetic programming, GP, linear representation, SBSE, software resilience, automatic code customisation, world wide location, plus codes, zip code.**

I. Introduction

Genetic Improvement (GI) [1], [2] uses search based software engineering [3] techniques, often genetic programming (GP) [4]–[6], to improve human written software. GI has been applied to automatic porting [2], transplanting code [7]–[9], code optimisation [10], [11] and automatic bugfixing. GP [12] and other search algorithms are increasingly used to automatically repair programs (APR) [13]–[20]. However genetic improvement has been criticised as lacking off-the-shelf, robust and easy to use tools [21].

Magpie [22] aims to address these issues. It has recently been released as an open source project by Dr. Aymeric Blot. We download it from GitHub[1]. Including examples and documentation, it comprises 4871 lines of code, mostly written in Python. It includes examples in Python, C++ and Ruby. So far it has not been used on applications.

Recently we have sped up two industrial programs written in C (one from Google's OLC and the other from Uber's H3) by applying genetic improvement [1], [2] directly to LLVM intermediate representation IR produced by the Clang C compiler [23]. (Previously Jhe-Yu Liou et al., e.g. [24], had used grammatical evolution [25] on LLVM IR to improve nVidia GPU kernels.) Our hill climbing code [23] is somewhat specialised and so we wished to test Magpie; not just in terms of its performance, but also its extensibility and ease of use.

In Section III we describe our changes to Magpie to make its calculation of fitness reproducible with the Linux operating system. Section IV details reusing the OLC and H3 fitness test cases from our earlier work, whilst Section V describes how much of the search space Magpie will sample. The results, Section VI, are followed in Section VII by a brief description of the C source code changes made by Magpie to give speed ups on the existing GCC -O3 compiler optimised code, and a discussion of possible future work. We conclude in Section VIII that Magpie is a great research tool (the appendix lists some issues) and here it gave speedups of up to 15%. The patches to both OLC and H3 generalise to many thousands of zip codes (see Figure 4). But first we describe the background.

[1]https://github.com/bloa/magpie

II. Background

Computing is the dominant industry. Everything relies on computers. Indeed many of the richest people on the planet are rich because they founded very successful software companies. Computing technology in general and software in particular permeates and will continue to dominate the third millennium. Indeed the world is already addicted to software. However, although programming is more than 60 years old, software continues to be hand written. The goal of automatic programming has been long espoused but with little concrete progress.

At ICSE-2009 Westley Weimer, ThanhVu Nguyen, Claire Le Goues and Stephanie Forrest [12] showed that genetic programming could automatically fix bugs in computer software. For the first time artificial intelligence was being applied to a major problem in software engineering on realistically sized programs. Since then the field of automatic program repair (APR) has bloomed. Inspired by [12] we [2], [10] began applying genetic programming to improving human written software in many ways in addition to bug fixing [1].

Although genetic programming remains a common search technique in genetic improvement (GI), local search is increasingly popular. In addition to ever more powerful computers, GI is able to scale because it does not start from scratch at every run but builds on existing software. One great advantage of this is that GI can automatically double check with the existing painstakingly hand written code, both on its run time performance (be it elapse time, memory requirements, etc.) and also its functionality. The increasing success of automatically generated software tests [26], also naturally feeds into GI.

At present much GI research is via bespoke one-off experiments. David White recognised this and proposed GIN [27] as a generic GI tool for Java programs. Gabin An proposed PyGGI [28], [29] for Python. Both tools have been extensively

used and updated: [30]–[33] and [34]–[38] and further GI tools have been proposed [39]. Nevertheless last year a GI user study reported that GI lacked user friendly tools [21].

In response to this Aymeric Blot wrote Magpie. Like PyGGI 2.0 [34] (from which it was developed) it is freely available from GitHub and can improve any measurable quantity of computer programs. Although also written in Python, it aims to work with any computer language. It has been mostly tested on Apple Mac and Linux Laptops but aims to be generic enough to work under Microsoft windows. Here we test it on a Linux desktop and have not attempted to maintain compatibility with Microsoft.

III. Updates to Magpie

This section describes a number of modifications to the standard GitHub version of Magpie, whilst Section IV returns to improving OLC and H3. In particular to describing the fitness function.

A. Replacing Pytest

By default Magpie uses the Pytest tool as the the second component of its multi-objective fitness function (the first being did the patch compile). Pytest has the advantage of being available for many operating systems and so can be view as increasing the portability of Magpie. In particular Magpie should be available under Microsoft MS Windows. However Pytest adds another level of indirection, and so complexity, for the non-Python novice Magpie user.

In our networked Centos Linux computer Pytest (version 5.0.1) was enormously slow to start the first time, causing Magpie to time out at the first attempt but often succeeding on the second and subsequent runs.

By default, Magpie's final fitness component is the time taken by Pytest to run all the test cases three times.

B. Extending the Warmup from 4 to 11 Empty Patches

To counter variability in run time measurement (see Figures 1 and 2) by default Magpie runs Pytest three times. Also because run time variation is often largest at the start of the run, by default Magpie runs its whole fitness process on four null patches in a warmup process before starting in earnest.

Due to the enormous variability in Pytest runtime we extended the warmup process from four to eleven empty patches. In view of the other changes made to measuring run time performance (Sections III-A and III-C to III-F) our warmup change might have been unnecessary. However we feel it makes performance comparison for each patch against the baseline more reliable. (Published speed ups are based on precise measurements taken after customisation, not those quoted as Magpie runs.)

On our computer we tried greatly increasing the number of times tests were run. But even running tests several thousand times, we did not get satisfactory results.

Pytest was replaced by Python scripts very like those supplied for use with Pytest. Like Pytest, they run each patch on a number (10 or 40) of test cases and check that the resulting answer is as expected. We continue to use the existing checks in Magpie for successful termination, user supplied timeout (1 second) and overly large output (up to one million bytes).

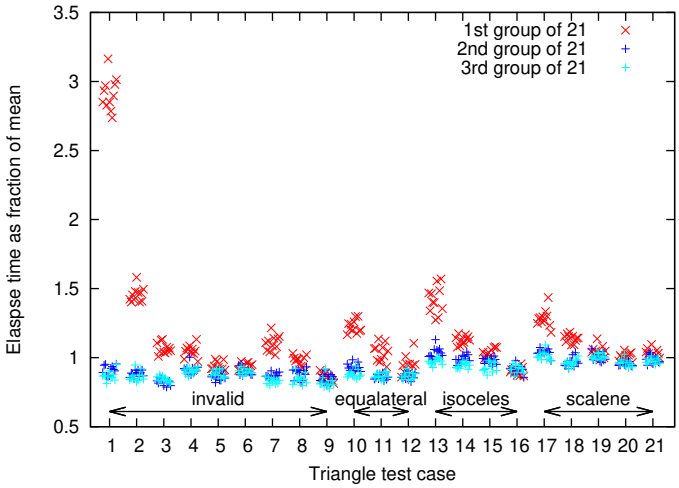

Fig. 1. Elapse times for Magpie's `triangle.py`'s 21 test cases. `check_classification` was run (without sleep 1 millisecond) in eleven groups of three (total 11×3×21=693 tests). Note both large systematic and random scatter. Despite being run continuously, the first (x=1) of each group of 11×3=63 (red ×) are three times slower than average. Whereas the same test (x=1) is close to the average speed when run in the second and third group of each 11 fold iteration (+). More than half the measurements are more than 7% from the average. Vertical axis normalised by dividing by observed mean: 485.5 nanoseconds. Small horizontal displacement added to spread data.

C. Moving Performance Measurement Closer to Patch

For convenience, sandboxing (see Section IV-B) and portability, Magpie runs its fitness tests on each patch in a subprocess. By default, the main Python code records the elapse time from when the subprocess is started to when it returns to the main thread. At least some of the observed variation was in the time taken by the operating system to create and start a new process running the tests and to pass the subprocess' output back to the main thread.

At the expense of generality, to exclude variation associated with communicating with the subprocess, it was decided to make the subprocess itself responsible for its own performance measurement and to pass fitness data back to the main thread. This is done (with suitable additional validity checks) via the Unix pipe connecting them. Indeed, to avoid variation in the Python interpreter, we took this one step further and moved the measurement from the interpreted Python code into the compiled C code. (The fitness measurement C code is protected from itself being patched by Magpie.)

D. Replacing Mean with First Quartile

By default Magpie uses total elapse time as part of its multi-objective fitness function. This is effectively the mean of n individual measurements. The mean is notoriously suspect to outliers and often the median (middle) is preferred. However run time is subject to one sided outliers. That is, an exceptionally long run time is not only possible but in practice

occurs much more often than equally short run times. Indeed since low outliers which deviate by more than the average would have a negative runtime they are not possible, whereas outliers that are more than twice the average do occur (see e.g. Figure 2).

A similar situation occurs in laser scanned genechips (as light intensity cannot be negative). One of the noise reduction techniques employed by Affymetrix (who manufacture genechips) is to use the first quartile. A simplistic arguments in their favour is: the noisier outliers tends to be in the larger half of the measurements, so we will take the median (which itself is known to be a robust statistic) of the lower half. The median of the lower half, is itself the first quartile of all the data. As with genechips, during evolution we are interested in comparing measurements, rather than absolute values. Hence although the first quartile may underestimate the true value, having consistent measurements helps drive evolution forward.

E. Replacing Runtime with Unix `perf` *Instruction Count*

Initially genetic improvement used fitness based on the number of lines of C++ executed [10]. However this required instrumenting the code and often direct measurement of the objective goal, e.g. reducing elapse time, is preferred. Unfortunately elapse time is often noisy (e.g. Figures 1 and 2) and this causes problems for GI [40], [41]. Sometimes taking multiple measurement and averaging (e.g. using the first quartile, as described in the previous section) is sufficient for the fitness function. In some cases, e.g. some NVIDIA GPUs, execution time is itself sufficiently stable to be used directly [11], [42]. Because of these problems, Eric Schulte et al. [43] and Aymeric Blot et al. [44] have advocated using the Linux Perf tool to measure run time performance for fitness. Indeed we followed their advice in earlier work with OLC and H3 [23]. Of course wall-clock time remains vital [45].

With Linux the GNU perf utility allows access to many performance measurements gathered by modern X86 computers. Using a small stub of C code (protected from being evolved by Magpie) we used the perf run time library `<linux/perf_event.h>` to collect the count of instructions executed by OLC and H3 for each test. The count of instructions executed by computer programs is typically much more stable than their execution time, cf. Figures 2 and 3.

F. Python Calling OLC and H3 Directly using `ctypes`

The Python `ctypes` function library allows the Python interpreter to pass arguments to and from a C function and to read its output. We used Python to convert the test cases into C standard argv and argc arguments for the OLC and H3 programs, thereby allowing Python to run them directly as functions. In both cases small tweaks to the OLC and H3 source code were made by hand to direct output from stdout to a buffer supplied by Python (e.g. replace `printf` by `sprintf`). On each test, the Python script checks the program returned a success status code, the value in the output buffer is correct and collects the perf performance data (see previous section). As before, problems where the patch causes

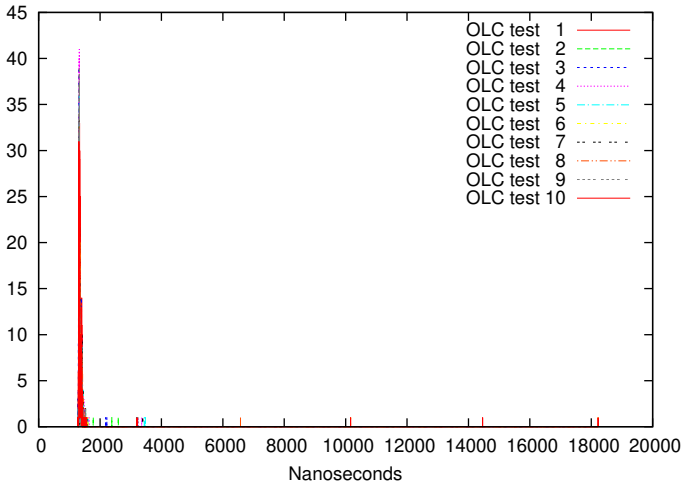

Fig. 2. Distribution of elapsed time for OLC (median 1332 nanoseconds). Each of the ten OLC test cases are run a thousand times. Note large spread (the inter quartile range, $\frac{Q3-Q1}{Q2} > 4\%$) and presence of enormous outliers (up to more than 13 times the average). Compared this to the number of instructions run, Figure 3, where there is almost no variation and differences between individual tests are clear.

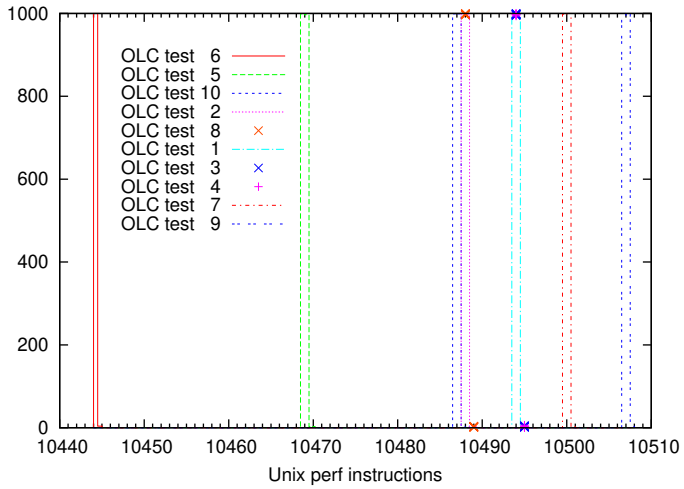

Fig. 3. Distribution of CPU instructions taken by OLC in nine separate runs each running the ten OLC test cases a thousand times. Note each of the nine runs has almost the same distribution and also each test case has almost no spread. Test cases 3, 4 and 8 overlap other tests and so are plotted as points $+$ or $\times$, rather than histograms. The key is ordered by the speed of each test case.

the program to fail or time out are trapped by the existing Magpie mechanism in the main thread.

Although, in the end we only needed to run 3×10 or 3×40 tests, to allow a huge number of tests to be run, the first quartile of the number of instructions run is calculated in the subprocess. This allows only (small) summary data, calculated by Python's `statistics.quantiles()`, to be passed back via the Unix pipe to the main Magpie thread.

IV. Fitness Function

Magpie attempts to run the patched program on all the test cases. If it produces correct answers for all of them, Magpie

runs the patch again multiple times to get a good robust estimate of its performance.

In summary: Magpie uses multiple facets to calculate a mutation's fitness: 1) does the patch compile without error (GCC compiler warnings are ignored), 2) does the mutant program run ok on every test case, 3) are its outputs the same as those of the unmutated code and 4) how long does it take.

In check (2), both Magpie and the mutant itself, can signal a problem via the Unix exit status. In either case, the main Magpie thread will discard the patch. (3) For each test case the Python subprocess will check that output of the patch is the same as the expected output. Outside Magpie we ran the unpatched OLC and H3 programs on each test case, recorded their output and then this was converted into Python source code for use by Magpie's subprocess. (4) Section III-E above describes how the perf C runtime library is integrated into Python. Magpie uses the first quartile of all the patch's repeated measurements to give its fitness measure (see Sections III-D to III-F).

A. Test Cases for OLC and H3: GB Post Codes

We used the same test cases as we had previously used when optimising the LLVM IR of Google's OLC and Uber's H3. Rather then require the reader to consult [23], we repeat some of our earlier description.

Both Google's Open Location Code (OLC)[2] and Uber's Hexagonal Hierarchical Geospatial Indexing System (H3)[3] are open industry standards. We obtained their human written sources from GitHub (total sizes OLC 14 024 and H3 15 015 lines of source code). They both include utilities written in C which convert global positions as pairs of latitude and longitude numbers into their own internal codes (see Table I). For OLC we used their 16 character coding and for H3 we used their highest resolution (`-r 15`) which uses 15 characters. Rather than work on abstract locations, we use as test cases the actual locations of homes and commercial premises.

For Google's OLC, the location of the first ten thousand GB postcodes were obtained[4]. For training ten pairs of latitude and longitude were selected uniformly at randomly (see Figure 4). The unmutated code was run on each pair and its output saved (16 bytes). For each test case each mutant's output is compared with the original output.

To summarise [23], Uber's H3 was treated similarly. However in the earlier work we had to used more diverse and more widely separated training data (see right of Figure 4).

B. Preventing Mutants Causing Harm

In [23] we describe the need for sandboxing, time outs and the need to prevent rouge mutants filling the disk [46]. In [23] we used the `timeout` and `limit` Unix commands. Here we simply use the default facilities of Magpie.

[2]https://github.com/google/open-location-code downloaded 4 August 2022.

[3]https://github.com/uber/h3 downloaded 3 August 2022.

[4]https://www.getthedata.com/downloads/open_postcode_geo.csv.zip dated 16 March 2022. The data are alphabetically sorted starting with AB1 0AA, which is in Aberdeen.

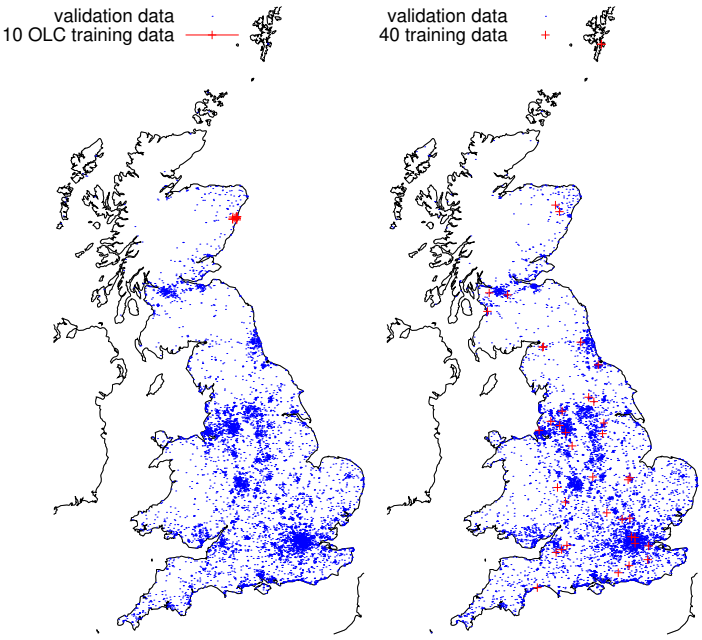

Fig. 4. Left: Ten OLC training points randomly selected in the neighbourhood of Aberdeen (+ red). Holdout set (blue dots) GB post codes. Right: Red + forty training points randomly selected from ten H3 runtime classes. Holdout set (blue dots), locations of ten thousand random GB post codes (no overlap with H3 training or OLC (left) holdout data). OLC and H3 (with and without -O3) pass all their holdout tests.

The principle sandboxing protection provided by Magpie (like [23]) is to run the mutated code in a subprocess. The OLC and H3 programs make little use of system calls and files. Hence their mutants tend to be quite benign. It may be in other programs, more care to prevent calamitous random actions, e.g. trapping or preventing generation of random file names, would be needed.

V. Magpie Search

Magpie allows a very nice separation of user supplied parameters for each experiment from generic code.

For both the OLC and H3 programs we ran experiments with the GNU GCC compiler without optimisation and with -O3 optimisation. (Although Profile Guided Optimisation, PGO, and Link Time Optimisation, LTO, are available, in their standard releases, both OLC and H3 are compiled with -O3) The four configurations are held in the examples/scenario directory. After the 11 warmup patches (Section III-B), Magpie generated 700 OLC patches and 19077 H3 patches.

At the start it was not clear how long we should run Magpie. Therefore we used a coupon collector [47] argument to calculate how many random samples would be needed to be almost certain of visiting every line of the C source code at least once. (The H3 source code to be optimised is much bigger than the OLC code, see Table I, hence the larger search effort.) In all four cases we used Magpie's run time reduction option: `python3 -m bin.magpie_runtime`.

Magpie was run in a single thread on an otherwise mostly idle 32 GB 3.60 GHz Intel i7-4790 desktop CPU running

TABLE I

Left: size of C sources for Google's OLC and Uber's H3 code to be optimised. Column 2 includes C .h header files. Column 3 size of human written C code, excluding header files, to be optimised (comments and blank lines removed). Right: averages for up to five Magpie runs. Columns 5–6 size of patch. Columns 7–8 minified patch. Column 9 best average reduction in perf's instructions per test case: OLC (without -O3) results were variable, sometimes as low as 0.1% speed-up, H3 first run run only. Column 10 gives the average run times for 1 core on a 3.6 GHz Intel i7-4790 desktop.

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

networked Unix Centos 7, using Python 3 version 3.10.1 and version 10.2.1 of the GNU C compiler. It may be that the networked disks are responsible for the excessive variation in elapsed time (see Figures 1 and 2) which seems much worse than reported with stand alone laptops. The variability seems to permeate the whole computer and running Magpie on a directly attached local disk did not solve this.

VI. Results

The results are summarised in Table I. For both OLC and H3 we conducted two experiments. Firstly with default parameters for the GNU C compiler and secondly using the -O3 optimisation flag.

Most patches which compile, run ok and pass all the tests. In detail: 45% of OLC patches fail to compile[5], 4% fail one or more fitness tests, whilst the remaining 52% pass all ten fitness tests. The pattern for H3 is similar: 32% of patches fail to compile and 4% fail at least one of the forty H3 tests. Perhaps because the search is longer and H3 is more complex, we also have 1.5% of patches being aborted when they try to generate more than a million bytes of output and 0.1% being timed out by Magpie. Leaving 63% which run ok and pass all 40 H3 fitness tests, see Figure 5.

Magpie has a nice inbuilt mechanism for minifying patches which we used and then tested the minified patch on the holdout dataset (10 000 randomly selected GB post codes). In most cases Magpie was able to reduce the number of individual changes to the lines of C source code. In all four cases, the minified patch generated the same results as the original program.

The holdout set contains "missing data", i.e. postal addresses without a latitude,longitude location. In these 85 cases OLC produces a default output, whilst H3 aborts (with a non-zero Unix status code) and an error message. Both minified H3 patches similarly detected and reported the error. For H3 we report the speed up relative to the original H3 program on the 9915 holdout locations with valid data.

The number of changes in the patches generated by Magpie and their speed up are given Table I. With the much longer runtime for H3, Magpie found patches comprised of several dozen changes to the lines of the H3 source code. Unfortunately, even after minification, outside Magpie, it proved impossible to automatically apply the patches. Instead the first H3 and the first H3 with -O3 patches were both applied by hand. Like OLC and OLC with -O3, they both past all the hold out tests, giving speed ups of 15% and 7%.

[5]Previously we used specialist mutation operators with LLVM IR which ensured all mutants compiled successfully to machine code [23].

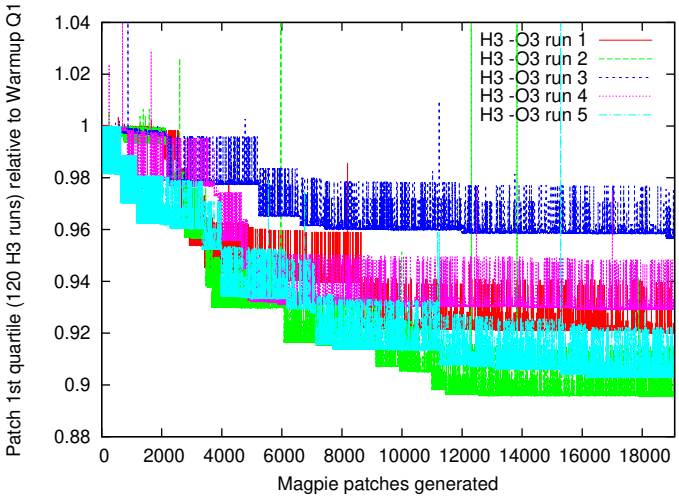

Fig. 5. H3 patches perf instruction counts for five Magpie runs with GCC -O3. Only patches which past all 40 training cases are plotted.

VII. Discussion

A. Types of Improvement Found

1) OLC: Several improvements found by Magpie to the OLC program are the same as we found in the early LLVM IR experiments [23]. E.g. removing code which normalises the inputs which is never used because the inputs (latitude and longitude) are already normalised. E.g. deleting the line `lat_degrees = -kLatMaxDegrees;` Removing the linefeed `\n` from the end of the output (which is there for aesthetic reason only and is not checked by the test cases). Also removing syntax checks on the command line. These checks are not needed in our data set because the command line is always valid. In some cases tests between `double` variables which always fail are replaced by others which also always fail but compare integers. However others modify code that is not executed and so it is not clear why the Magpie patch minification did not remove them.

2) H3: There are too many H3 source code improvements to describe them all in detail. The patches Magpie found with compiler optimisation GCC -O3 and without it have many changes in common. Also even though the H3 code is totally different from the OLC code, some of the changes have some things in common with those for the OLC code (both when optimising LLVM IR [23] and when using Magpie). For example: removing code for command line options (such as "help") which never occur in the fitness or holdout test suites, not normalising data which is always normalised and not checking for errors which never happen.

B. Future work, Co-evolution, `perf`, Profiling, Mutation and other Search Operators

So far we have avoided in-depth analysis of the target program's internal behaviour. The H3 example is an order of magnitude bigger than the OLC program. Both are open source and free to download. This makes fuzz testing and white box software engineering techniques which target edge cases and branch coverage feasible. In our earlier work [23], perhaps due to H3's greater size and complexity, it proved necessary to both increase the number (40 v 10) of fitness test cases and to use external measures, such as run time and geographic spread and to increase the fraction of "difficult" examples in the training test suite. Being external, they could in principle be used where access to the source code is restricted (cf. Eric Schulte's work on repairing black box network devices [50]).

In cases where improvements to the training data are needed, an adversarial co-evolutionary approach might be useful. Perhaps a population of training points could be optimised, e.g. using genetic programming, to antagonistically increase execution time.

As expected [44], `perf` offers considerable noise reduction compared to the Unix `time` command. Perf's runtime library makes it straightforward to include it directly in the code being optimised. Perf is also available via the Linux command line. However even perf's instruction count is noisy. Here we were able to use the first quartile statistic to minimise fitness noise. Surprisingly, especially when used via the command line, instruction count is also subjected to systematic variation. For example, it can increase when more data are held in Unix global environment variables.

We have targeted whole functions that could possibly be called. It is common in GI, to use profiling tools to target only code that is indeed executed [10]. In these examples, profiling was not used. Thus Magpie wasted effort trying to optimise un-executed lines of code. Although after search we used Magpie's minification of patches, this was not entirely successful at automatically producing the smallest or most beautiful change to the source code. We suggest profiling could help Magpie. In "line mode" Magpie is totally blind to the syntax of the program source code. For example, it treats a line containing a single closing brace `}` in the same way as any other line of the program source code. Of course Magpie has more sophisticated syntax aware approaches but they also might benefit from profile data.

Execution time potentially allows a multi-objective (Pareto) tradeoff between size of the GI change and the benefit [51].

With Magpie we only used a few types of mutation: `LineDeletion`[6], `LineInsertion` and `LineReplacement`. Many others, swaps and crossover could be devised. Sometimes individual changes are independent and can be applied to give an individual improvement. However some changes interfere, giving rise to an epistatic fitness landscape [53], [54] for which genetic search may be suitable.

[6]Delete is the most common way programmers speed up code [52].

VIII. Conclusions

We have taken a new open source genetic improvement tool written in Python (Magpie) and applied it to non-trivial industrial C source codes. Magpie proved easy to install and extend. Its modular design enabled the ready replacement of elapse time by a less noisy fitness measure. Although Magpie's Python code also readily supported the replacement in the fitness function of the mean by the more robust first quartile statistic, we feel, in our networked environment, the replacement of noisy elapsed time measurement by using the Linux perf's runtime library to access the count of instructions executed by the automatically produced patches was critical to Magpie's success. Run time in dedicated computers and laptops can be less variable, in which case Magpie may not need Linux specific performance measuring tools.

The appendix documents several "gotchas", but perhaps only our inability to automatically apply the long lists of changes generated for our second example (H3), impedes the widespread up take of Magpie in genetic improvement work[7]. Here the isolation provided by running in a separate process provided sufficient isolation to protect the user's computer from mutant code. There may be other circumstances, where a stronger sandbox is required.

On two examples from industry standard codes written in C (Google's OLC and Uber's H3) Magpie can in a few minutes or hours[8] give speed ups of up to 3.6% (OLC) and 15% (H3). Even giving 2% (OLC) and 7% (H3) improvement on compiler optimised code.

Acknowledgements

We are grateful for help from Aymeric Blot (Magpie), H.Wierstorf (gnuplot) and our anonymous reviewers.

The code and test cases are available in our reproduction artifact olc_h3_icse2023.tar.gz

Funded by the Meta Oops project.

Appendix
Problems with Magpie

Magpie is available from GitHub `https://github.com/bloa/magpie` and is under continuous development. Some of the problems we encountered (http://www.cs.ucl.ac.uk/staff/W.Langdon/magpie/) in the 27 November 2022 version, may already have been fixed.

A. Pytest

Several Magpie options use Pytest. Initial problems were solved by ensuring the Unix PATH environment variable included the location of Pytest and a compatible version of Python.

[7]More recent versions of Magpie already fix the long patch list problem.

[8]Magpie with the GNU C compiler processed between 3.6 and 8.6 patches per second depending on -O3, OLC or H3 (slower with -O3). Whereas with Clang 14.0.0 in earlier work [23] we processed between 0.25 and 1.5 LLVM IR patches per second, again depending on using -O3, OLC or H3

B. Python 3

Magpie uses Python 3. On our system this requires the use of the `python3`, command rather than simply `python`. When the wrong version of Python is used, it can be hard to diagnose the problem.

C. `--mode`, *–config files, and Timeouts*

Magpie supports many modes of operation. Incorrect use of the `--mode` command line option can cause hard to understand error messages.

Similarly incorrectly set up `scenario --config` files can give hard to understand error messages.

As mentioned in the main text, Magpie has been set up to work easily on laptops, and on our networked computer simple commands (such as `pytest`) can take far longer to execute the first time, causing hard to diagnose timeouts. (Most timeouts can be configured by the user editing their problem dependent `scenario --config` file.)

D. Python Line Numbers Start at 0

There was confusion with line numbers being incorrect until it was realised that Magpie reports line numbers in the source files starting at zero.

E. Compiling with Unix Make

When using compiled languages, such as C, Magpie requires the user to tell it how to compile each patch it finds using `compile_cmd =` in the problem's `--config` file. With care this can be simply the line `compile_cmd = make` However `make` is a sophisticated utility and when things go wrong (e.g. pulling unexpected files from RCS and cleaning up afterwards) having the error buried under Magpie, may complicate diagnosis.

It may be easier and more efficient to simply quote the compiler command line directly in the `--config` file. E.g.

```
compile_cmd = gcc -shared -o triangle.so
-fmax-errors=1 -fPIC triangle.c
```

F. Strange Python Errors

Often these can be resolved by ensuring you are using a recent version of Python 3, i.e. not Python 2. (The command `python --version` can be helpful.) Again not really Magpie's fault, but the error messages are aimed at the Python expert rather than the novice Magpie user.

G. Strange Shared Library Errors and `cdll .so` *files*

Another important Unix environment variable is `LD_LIBRARY_PATH`. Like `PATH`, hard to diagnose problems can occur if it is set incorrectly.

If using Python's `ctypes` library to call C functions, which can include the whole program via the `main()` function, `ctypes cdll` will need to find the shared library containing Magpie patches. It seems easiest to specify the directory holding the shared library and thus prevent Python looking in the system libraries and having to set up `LD_LIBRARY_PATH`. If a directory is not given, there is a complicated hierarchy of defaults that Python uses. However, in Unix simply adding `./` (meaning the default directory) to the file name (e.g. `cdll.LoadLibrary("./olc.so")`) ensures that Magpie will look in its current working directory for the shared library containing the compiled patch.

References

[1] J. Petke *et al.*, "Genetic improvement of software: a comprehensive survey," *IEEE TEVC*, vol. 22, no. 3, pp. 415–432, 2018. http://dx.doi.org/doi:10.1109/TEVC.2017.2693219

[2] W. B. Langdon and M. Harman, "Evolving a CUDA kernel from an nVidia template," in *WCCI*, P. Sobrevilla, Ed., 2010, pp. 2376–2383. http://dx.doi.org/10.1109/CEC.2010.5585922

[3] M. Harman and B. F. Jones, "Search based software engineering," *Information and Software Technology*, vol. 43, no. 14, pp. 833–839, 2001. http://dx.doi.org/10.1016/S0950-5849(01)00189-6

[4] J. R. Koza, *Genetic Programming: On the Programming of Computers by Means of Natural Selection*, 1992. http://mitpress.mit.edu/books/genetic-programming

[5] W. Banzhaf, P. Nordin, R. E. Keller, and F. D. Francone, *Genetic Programming – An Introduction; On the Automatic Evolution of Computer Programs and its Applications*, 1998. https://www.amazon.co.uk/Genetic-Programming-Introduction-Artificial-Intelligence/dp/155860510X

[6] R. Poli, W. B. Langdon, and N. F. McPhee, *A field guide to genetic programming*. Published via `http://lulu.com` and freely available at `http://www.gp-field-guide.org.uk`, 2008. http://www.gp-field-guide.org.uk

[7] E. T. Barr *et al.*, "Automated software transplantation," in *International Symposium on Software Testing and Analysis, ISSTA 2015*, Tao Xie and M. Young, Eds., 2015, pp. 257–269. http://dx.doi.org/10.1145/2771783.2771796

[8] A. Marginean, E. T. Barr, M. Harman, and Y. Jia, "Automated transplantation of call graph and layout features into Kate," in *SSBSE*, ser. LNCS, Y. Labiche and M. Barros, Eds., vol. 9275, 2015, pp. 262–268. http://dx.doi.org/10.1007/978-3-319-22183-0_21

[9] A. Marginean, "Automated software transplantation," Ph.D. dissertation, University College London, 2021. https://discovery.ucl.ac.uk/id/eprint/10137954/1/Marginean_10137954_thesis_redacted.pdf

[10] W. B. Langdon and M. Harman, "Optimising existing software with genetic programming," *IEEE TEVC*, vol. 19, no. 1, pp. 118–135, 2015. http://dx.doi.org/10.1109/TEVC.2013.2281544

[11] W. B. Langdon *et al.*, "Genetic improvement of GPU software," *GP&EM*, vol. 18, no. 1, pp. 5–44, 2017. http://dx.doi.org/10.1007/s10710-016-9273-9

[12] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest, "Automatically finding patches using genetic programming," in *ICSE*, S. Fickas, Ed., 2009, pp. 364–374. http://dx.doi.org/10.1109/ICSE.2009.5070536

[13] W. Weimer, S. Forrest, C. Le Goues, and T. Nguyen, "Automatic program repair with evolutionary computation," *Communications of the ACM*, vol. 53, no. 5, pp. 109–116, 2010. http://dx.doi.org/10.1145/1735223.1735249

[14] S. O. Haraldsson, J. R. Woodward, A. E. I. Brownlee, and K. Siggeirsdottir, "Fixing bugs in your sleep: How genetic improvement became an overnight success," in *GI-2017*, J. Petke, D. R. White, W. B. Langdon, and W. Weimer, Eds., 2017, pp. 1513–1520. http://dx.doi.org/10.1145/3067695.3082517

[15] C. Le Goues, M. Pradel, and A. Roychoudhury, "Automated program repair," *Communications of the ACM*, vol. 62, no. 12, pp. 56–65, 2019. http://dx.doi.org/10.1145/3318162

[16] M. Monperrus, "Automatic software repair: A bibliography," *ACM Computing Surveys*, vol. 51, no. 1, p. article no 17, 2018. http://dx.doi.org/10.1145/3105906

[17] N. Alshahwan, "Industrial experience of genetic improvement in Facebook," in *GI-2019, ICSE workshops proceedings*, J. Petke, Shin Hwei Tan, W. B. Langdon, and W. Weimer, Eds., 2019, p. 1. http://dx.doi.org/10.1109/GI.2019.00010

[18] M. Harman, "Scaling genetic improvement and automated program repair," in *International Workshop on Automated Program Repair (APR'22)*, M. Kechagia, Shin Hwei Tan, S. Mechtaev, and L. Tan, Eds., 2022. http://dx.doi.org/10.1145/3524459.3527353

[19] S. Kirbas *et al.*, "On the introduction of automatic program repair in Bloomberg," *IEEE Software*, vol. 38, no. 4, pp. 43–51, 2021. http://dx.doi.org/10.1109/MS.2021.3071086

[20] M. Kechagia, Shin Hwei Tan, S. Mechtaev, and L. Tan, Eds., *2022 IEEE/ACM International Workshop on Automated Program Repair (APR)*, 2022. https://ieeexplore.ieee.org/xpl/conhome/9474454/proceeding

[21] Shengjie Zuo, A. Blot, and J. Petke, "Evaluation of genetic improvement tools for improvement of non-functional properties of software," in *GECCO*, B. R. Bruce *et al.*, Eds. Association for Computing Machinery, 2022, pp. 1956–1965. http://dx.doi.org/10.1145/3520304.3534004

[22] A. Blot and J. Petke, "MAGPIE: Machine automated general performance improvement via evolution of software," arXiv, 2022. http://dx.doi.org/10.48550/ARXIV.2208.02811

[23] W. B. Langdon, A. Al-Subaihin, A. Blot, and D. Clark, "Genetic improvement of LLVM intermediate representation," in *EuroGP*, ser. LNCS, G. Pappa, M. Giacobini, and Z. Vasicek, Eds., vol. 13986, 2023. http://www.cs.ucl.ac.uk/staff/W.Langdon/ftp/papers/langdon_2023_EuroGP.pdf

[24] J.-Y. Liou, Xiaodong Wang, S. Forrest, and C.-J. Wu, "GEVO: GPU code optimization using evolutionary computation," *ACM Transactions on Architecture and Code Optimization*, vol. 17, no. 4, p. Article 33, 2020. http://dx.doi.org/10.1145/3418055

[25] M. O'Neill and C. Ryan, *Grammatical Evolution: Evolutionary Automatic Programming in a Arbitrary Language*, ser. Genetic programming, 2003, vol. 4. http://dx.doi.org/10.1007/978-1-4615-0447-4

[26] S. Schweikl, G. Fraser, and A. Arcuri, "EvoSuite at the SBST 2022 tool competition," in *SBST*, G. Guizzo and S. Panichella, Eds., 2022, pp. 33–34. http://dx.doi.org/10.1145/3526072.3527526

[27] D. R. White, "GI in no time," in *GI-2017*, J. Petke, D. R. White, W. B. Langdon, and W. Weimer, Eds., 2017, pp. 1549–1550. http://dx.doi.org/doi:10.1145/3067695.3082515

[28] Gabin An, Jinhan Kim, Seongmin Lee, and S. Yoo, "PyGGI: Python General framework for Genetic Improvement," in *Proceedings of Korea Software Congress*, ser. KSC 2017, 2017, pp. 536–538. https://coinse.kaist.ac.kr/publications/pdfs/An2017aa.pdf

[29] Gabin An, Jinhan Kim, and Shin Yoo, "Comparing line and AST granularity level for program repair using PyGGI," in *GI-2018, ICSE workshops proceedings*, J. Petke, K. Stolee, W. B. Langdon, and W. Weimer, Eds., 2018, pp. 19–26. http://dx.doi.org/10.1145/3194810.3194814

[30] A. E. I. Brownlee *et al.*, "Gin: genetic improvement research made easy," in *GECCO*, M. Lopez-Ibanez *et al.*, Eds., 2019, pp. 985–993. http://dx.doi.org/10.1145/3321707.3321841

[31] J. Petke and A. Brownlee, "Software improvement with Gin: a case study," in *SSBSE 2019*, ser. LNCS, S. Nejati and Gregory Gay, Eds., vol. 11664, 2019, pp. 183–189. http://dx.doi.org/10.1007/978-3-030-27455-9_14

[32] J. Petke and A. Blot, "Refining fitness functions in test-based program repair," in *The First International Workshop on Automated Program Repair (APR@ICSE 2020)*, Shin Hwei Tan, S. Mechtaev, M. Monperrus, and M. Prasad, Eds. Association for Computing Machinery, 2020, pp. 13–14. http://dx.doi.org/10.1145/3387940.3392180

[33] S. A. Licorish and M. Wagner, "On the utility of marrying GIN and PMD for improving Stack Overflow code snippets," ArXiv, 2022. https://dblp.org/rec/journals/corr/abs-2202-01490.bib

[34] Gabin An, A. Blot, J. Petke, and S. Yoo, "PyGGI 2.0: Language independent genetic improvement framework," in *Proceedings of the 27th Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering ESEC/FSE 2019)*, S. Apel and A. Russo, Eds., 2019, pp. 1100–1104. http://dx.doi.org/10.1145/3338906.3341184

[35] L. Kitt and M. B. Cohen, "Partial specifications for program repair," in *GI @ ICSE 2021*, J. Petke *et al.*, Eds., 2021, pp. 19–20. http://dx.doi.org/10.1109/GI52543.2021.00012

[36] M. Smigielska, A. Blot, and J. Petke, "Uniform edit selection for genetic improvement: Empirical analysis of mutation operator efficacy," in *GI @ ICSE 2021*, J. Petke *et al.*, Eds., 2021, pp. 1–8. http://dx.doi.org/10.1109/GI52543.2021.00009

[37] I. Mesecan *et al.*, "CRNRepair: Automated program repair of chemical reaction networks," in *GI @ ICSE 2021*, J. Petke *et al.*, Eds., 2021, pp. 23–30. http://dx.doi.org/10.1109/GI52543.2021.00014

[38] A. Blot and J. Petke, "Empirical comparison of search heuristics for genetic improvement of software," *IEEE TEVC*, vol. 25, no. 5, pp. 1001–1011, 2021. http://dx.doi.org/10.1109/TEVC.2021.3070271

[39] O. Krauss, "Amaru - a framework for combining genetic improvement with pattern mining," in *GECCO*, B. R. Bruce *et al.*, Eds. Association for Computing Machinery, 2022, pp. 1930–1937. http://dx.doi.org/10.1145/3520304.3534016

[40] S. O. Haraldsson and J. R. Woodward, "Genetic improvement of energy usage is only as reliable as the measurements are accurate," in *GI*, W. B. Langdon, J. Petke, and D. R. White, Eds., 2015, pp. 831–832. http://dx.doi.org/10.1145/2739482.2768421

[41] W. B. Langdon, J. Petke, and B. R. Bruce, "Optimising quantisation noise in energy measurement," in *PPSN*, ser. LNCS, J. Handl *et al.*, Eds., vol. 9921, 2016, pp. 249–259. http://dx.doi.org/10.1007/978-3-319-45823-6_23

[42] W. B. Langdon and Brian Yee Hong Lam, "Genetically improved BarraCUDA," *BioData Mining*, vol. 20, no. 28, 2017. http://dx.doi.org/10.1186/s13040-017-0149-1

[43] E. Schulte *et al.*, "Post-compiler software optimization for reducing energy," in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS'14*, 2014, pp. 639–652. http://dx.doi.org/10.1145/2541940.2541980

[44] A. Blot and J. Petke, "Using genetic improvement to optimise optimisation algorithm implementations," in *23ème congrès annuel de la Société Française de Recherche Opérationnelle et d'Aide à la Décision, ROADEF'2022*, K. Hadj-Hamou, Ed. INSA Lyon, 2022. http://www.cs.ucl.ac.uk/staff/a.blot/files/blot_roadef_2022.pdf

[45] J. Woodward, A. Brownlee, and C. Johnson, "Evals is not enough: why we should report wall-clock time," in *GI*, J. Petke, D. R. White, and W. Weimer, Eds., 2016, pp. 1157–1158. http://dx.doi.org/10.1145/2908961.2931695

[46] M. Harman, Yue Jia, and W. B. Langdon, "A manifesto for higher order mutation testing," in *Mutation 2010*, L. du Bousquet, J. Bradbury, and G. Fraser, Eds., 2010, pp. 80–89. http://dx.doi.org/10.1109/ICSTW.2010.13

[47] W. Feller, *An Introduction to Probability Theory and Its Applications*, 2nd ed. John Wiley and Sons, 1957.

[48] W. B. Langdon, "Genetic improvement of genetic programming," in *GI @ CEC 2020 Special Session*, A. S. Brownlee, S. O. Haraldsson, J. Petke, and J. R. Woodward, Eds., 2020. http://dx.doi.org/10.1109/CEC48606.2020.9185771

[49] M. Papadakis, Yue Jia, M. Harman, and Y. Le Traon, "Trivial compiler equivalence: A large scale empirical study of a simple fast and effective equivalent mutant detection technique," in *ICSE*, 2015. http://pages.cs.aueb.gr/~mpapad/papers/ICSE15B.pdf

[50] E. Schulte, W. Weimer, and S. Forrest, "Repairing COTS router firmware without access to source code or test suites: A case study in evolutionary software repair," in *GI*, W. B. Langdon, J. Petke, and D. R. White, Eds., 2015, pp. 847–854. http://dx.doi.org/10.1145/2739482.2768427

[51] P. Sitthi-amorn, N. Modly, W. Weimer, and J. Lawrence, "Genetic programming for shader simplification," *ACM Transactions on Graphics*, vol. 30, no. 6, p. article:152, 2011. http://dx.doi.org/10.1145/2070781.2024186

[52] J. Callan, O. Krauss, J. Petke, and F. Sarro, "How do Android developers improve non-functional properties of software?" *Empr. Soft. Eng.*, vol. 27, p. Article 113, 2022. http://dx.doi.org/10.1007/s10664-022-10137-2

[53] J. Petke *et al.*, "A survey of genetic improvement search spaces," in *GECCO*, B. Alexander, S. O. Haraldsson, M. Wagner, and J. R. Woodward, Eds., 2019, pp. 1715–1721. http://dx.doi.org/10.1145/3319619.3326870

[54] W. B. Langdon, N. Veerapen, and G. Ochoa, "Visualising the search landscape of the Triangle program," in *EuroGP 2017*, ser. LNCS, M. Castelli, J. McDermott, and L. Sekanina, Eds., vol. 10196, 2017, pp. 96–113. http://dx.doi.org/10.1007/978-3-319-55696-3_7