Genetically Improved Software with fewer Data Cache Misses

William B. Langdon, Justyna Petke, Aymeric Blot, David Clark
{W.Langdon,j.petke,david.clark}@ucl.ac.uk
aymeric.blot@univ-rennes.fr
Department of Computer Science, University College London

ABSTRACT

Using MAGPIE (Machine Automated General Performance Improvement via Evolution of software) we show genetic improvement GI can reduce the cache load of existing computer programs. Cache miss reduction is tested on two industrial open source C programs (Google's Open Location Code OLC and Uber's Hexagonal Hierarchical Spatial Index H3) and two C++ 2D photograph image processing tasks, counting pixels and OpenCV's SEEDS segmentation algorithm. Magpie's patches functionally generalise. In one case they reduce data misses on the highest performance L1 cache by 47%.

CCS CONCEPTS

• **Software and its engineering → Search-based software engineering**.

KEYWORDS

genetic programming, genetic improvement, local search, SBSE, linear representation, software resilience, automatic code optimisation, tabu, nonstationary noise, perf, world wide location, plus codes, zip code, OpenCV, image segmentation

ACM Reference Format:
William B. Langdon, Justyna Petke, Aymeric Blot, David Clark. 2023. Genetically Improved Software with fewer Data Cache Misses. In *Genetic and Evolutionary Computation Conference Companion (GECCO '23 Companion), July 15–19, 2023, Lisbon, Portugal.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3583133.3590542

1 INTRODUCTION

For the next decade or more, computing will be increasingly parallel and increasingly the limiting factor will not be computing itself but instead getting data to the many compute engines in a timely fashion. That is, computing is cheap, it is data movement that is expensive. Data caches remain the only way to simultaneously feed data into the mouths of hundreds, even thousands, of CPUs at anything like their free running rate. We show evolutionary computing can automatically adapt existing software to reduce the load it places on data caches. Further, we seek to strengthen the claim that it can optimise any *comparable* aspect of existing software. A longer version is online https://arxiv.org/abs/2304.03235

Section 3 details the fitness test cases for Google's OLC and Uber's H3 digital mapping programs, the Blue image benchmark, and the OpenCV SEEDS resource intensive image segmentation algorithm. While Section 4 describes using a Coupon Collector argument to choose how much of the search space Magpie should sample. The results in Section 5 show on the image examples Magpie gave reduction in L1 data cache misses of up to 47% even on the existing compiler (GCC -O3) optimised code (see Table 1). In Section 6 there is a discussion of non-stationary noise, profiling and our use of a Tabu list to improve search. We conclude in Section 7, but first we describe Magpie.

2 GENETIC IMPROVEMENT WITH MAGPIE

Genetic improvement research is often via bespoke one-off experiments. David White recognised this and proposed GIN [11] as a generic GI tool for Java programs. Similarly Gabin An [1] proposed PyGGI for Python. Nevertheless recently a user study said that GI lacked user friendly tools [12]. To address this Magpie [2] was released last year as an open source project. It is freely available from https://github.com/bloa/magpie. We use it to hopefully convince the reader that evolution can in principle improve any *comparable* measure of software quality.

As of 27 November 2022, including examples and documentation, Magpie contains 4871 lines of code, mostly written in Python. It contains examples in Python, C, C++ and Ruby.

2.1 Updates to Magpie

In the course of previous work [9], we had enhanced Magpie to use Python's `ctypes` to directly call the patched code. This allows us to reduce noise by collecting data directly on the individual C/C++ routines rather than the whole Magpie sub-process. Also to exclude the Python interpreter from our measurement, we clear the L1 data cache before invoking the patch code by setting and reading a fixed array of 32K bytes (the size of our CPU's fastest, L1, data cache). To make the fitness robust, each patch is tested multiple times and since the mean is notoriously suspect to noisy outliers, we use instead the first quartile to summarise the measurements.

In the image segmentation example (Section 3.2) 17% of the lines consist of a single closing brace `}`. These are naturally interchangeable and so it was discovered that Magpie was wasting a lot of effort testing programs that were identical (apart from white space, etc). To prevent this, the compilation step keeps a "tabu" list [6]. Previously [6] we had a complicated tabu of both genotypic and phenotypic information (of up to 340 MBytes), here we simply keep a copy of each object file (on average 407 files per run, occupying about 30 MBytes). After compilation, semantically identical patches are rejected by simply comparing their object file with object files of the same size from previous patches. New unique patches are added to the tabu directory and identical ones are discarded without fitness testing. Although in the limit this could be slow as the tabu directory grows, in practice the time taken is negligible.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
GECCO '23 Companion, July 15–19, 2023, Lisbon
© 2023 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0120-7/23/07.
https://doi.org/10.1145/3583133.3590542

The Linux GNU perf utility allows access to many hardware performance counters. In particular we used the perf run time library `linux/perf_event.h` to collect L1 data and instruction cache misses, the count of instructions executed and elapsed time. (Only the L1 data cache misses are used by the fitness function.)

2.2 Datasets

We use four open source C/C++ examples. Two industrial geospatial programs, both written in C [8]. One from Google's OLC and the other from Uber's H3 (see Section 3.1). And two C++ photographic image processing examples (Section 3.2). These are: our Blue benchmark from the 2018 Tarot summer school [4] and OpenCV's image segmentation code [7].

3 FITNESS FUNCTION

Magpie attempts to run the patched program on all the test cases. If the patch passes them all, Magpie runs it again multiple times to try to get a good robust estimate of its performance.

In summary: Magpie uses multiple objectives to calculate a mutation's fitness. In sequential (priority) order: 1) does the patch compile without error (warnings are ignored), 2) does the mutant software run without crashing or timing out on every test case, 3) are its outputs the same as those of the original code and 4) how many L1 data cache misses does `perf` record.

(1) The source code is compiled using the GCC compiler (version 10.2.1) with the same options and switches, e.g. -O3, as are used by the developers (Google, Uber and the OpenCV team). To avoid wasting time on reporting multiple errors, `-fmax-errors=1` is used to stop GCC on the first error. If the compiler succeeds in compiling the patch, it is linked with non-evolvable code outside the patch, and the C code that calls the `perf` run time library, to form a shared object library. (The GCC options `-shared` and `-fPIC` are used to create the shared library `./prog.so`). The Python interpreter uses python `ctypes` on the shared library to call the C interface routine which calls the `perf` runtime library and the patched code. After the patch has been run, the `perf` runtime library extracts hardware counters from within the CPU, which the interface code passes back to the Python interpreter, along with the outputs generated by the patch.

(2) Both Magpie (via the Python interpreter) and the mutant itself, can signal a problem via the Unix exit status. In either case, the main Magpie thread will discard the patch giving it poor fitness and then move onto generating and testing the next patch.

(3) For each test case the Python subprocess will check that output of the patch is as expected for each user supplied test case. For example, with OLC, Magpie checks that the patched code returned the same 16 characters as Google's code for the test's pair of latitude and longitude. If any characters are different or missing the test fails and fitness testing for that patch stops immediately. Before using Magpie, we ran the original OLC, H3, Blue and SEEDS programs on each test case, recorded their output and then this was automatically converted into a Python list data structure. For example, with OLC, the fitness training consists of ten latitude and longitude floating point numbers and ten 16 character strings, formatted as ten Python bracketed tuples.

(4) Section 2.1 above describes how the `perf` C runtime library is integrated into Python. Magpie uses the first quartile (Q1) of all the patch's repeated measurements to give its fitness measure (see also Section 2.1). Even in supposedly deterministic programs, the hardware counters for cache statistics, instructions run and elapsed time are noisy. Despite the use of robust statistics like Q1 on `perf`'s L1 data cache misses, fitness remains noisy. But as we will see, in some cases Magpie is able to make progress.

3.1 Test Cases for Google's OLC and Uber's H3: GB Post Codes

We used the same test cases as before [8] when optimising OLC and H3.

Both Google's Open Location Code (OLC) https://github.com/google/open-location-code (downloaded 4 August 2022) and Uber's Hexagonal Hierarchical Geospatial Indexing System (H3) https://github.com/uber/h3 (downloaded the previous day) are open industry standards (total sizes OLC 14 024 and H3 15 015 lines of source code). They include C programs which convert latitude and longitude into their own internal codes (see Table 1). For OLC we used Google's 16 character coding and for H3 we used Uber's highest resolution (`-r 15`) which uses 15 characters. Following our earlier work [8], we use as test cases the position of actual addresses.

For Google's OLC we used the [8] dataset which was the location of the first ten thousand GB postcodes downloaded from https://www.getthedata.com/downloads/open_postcode_geo.csv.zip (dated 16 March 2022). The addresses are alphabetically sorted starting with AB1 0AA, which is in Aberdeen. For training ten pairs of latitude and longitude were selected uniformly at randomly 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.

Uber's H3 was treated similarly. (Full details of both OLC and H3 datasets are given in [8]).

3.2 Test Cases for Blue and OpenCV SEEDS

We had previously produced a simple example of the GISMO GI system for students attending the 2018 TAROT summer school on Software Testing, Verification and Validation [4] http://www.cs.ucl.ac.uk/staff/W.Langdon/ftp/gp-code/opencv_gp.tar.gz. We generated ten random images and calculated the number of "blue" pixels in each. These were used by Magpie as training data, with a goal of optimising their code to minimise L1 cache misses. Notice the training images contain $96 \times 128 = 12\,288$ coloured pixels, occupying 49 152 bytes, and so exceed the L1 data cache. We removed the comments, leaving 100 lines of C++ code.

In contrast OpenCV is an enormous suite of C++ image processing tools. At the beginning of 2023 OpenCV's open source repository on GitHub comprised more than two million lines of code (mostly C, C++ and XML). Therefore, we selected an important routine: the state-of-the-art OpenCV SEEDS superPixels image segmentation algorithm. This figured in the $50K OpenCV Challenge, and we had previously used it in GI experiments to reduce run time whilst respecting its API [7].

The OpenCV code of the SEEDS SuperPixel algorithm is 1500 lines of C++ code, but the important routines are held in one file `updatePixels.cpp`. After removing comments and empty lines

Table 1: Left: size of C/C++ sources to be optimised (comments and blank lines removed). Middle: averages for up to five Magpie runs. Columns 3–4 size of patch. Column 5 best training fitness (average reduction in L1 data cache misses). Right: Column 6 size of search space explored. Column 7 fraction of mutants which compile, run ok and give correct answer. Column 8 average Magpie run times for 1 core on a 3.6 GHz Intel i7-4790 desktop with GCC 10.2.1 and Python 3.10.1

Example	LOC	Mutant		Magpie		
		size	L1D	steps	% run ok	duration
OLC	134	1– 5	5%	700	23%	66 secs
H3	1615	6–18	6%	19077	33%	3.9 hours
blue	100	7–10	47%	904	30%	1.7 hours
SEEDS	319	2– 7	7%	3151	2%	5.2 hours

there are 319 lines. Unfortunately the SEEDS algorithm is compute intensive and so instead of using full images obtained from [7], we reduced the training data to 1/16. Notice the 204×153 training image contains 31 212 coloured pixels (124 848 bytes) and so the major data structure used by the SEEDS algorithm exceeds the L1 data cache.

4 MAGPIE SEARCH

Magpie generated 700 OLC, 19 077 H3, 904 Blue and 3151 SEEDS patches (see Table 1). The OLC and H3 (700 and 19077) values are taken from our previous work [8]. As before [8], we use a coupon collector [3] argument to calculate how many random samples would be needed to be almost certain of visiting every line of the C/C++ source code at least once. (The H3 source code to be optimised is much bigger than the others, see Table 1, hence the larger search effort.) In all four cases we used Magpie's run time reduction option: `python3 -m bin.magpie_runtime`.

Magpie used a single main thread on an otherwise mostly idle 32 GB 3.60 GHz Intel i7-4790 desktop CPU running networked Unix Centos 7, using Python 3 version 3.10.1 and version 10.2.1 of the GNU C compiler. In all four cases there was a lot of variation in values recorded by `perf`.

5 RESULTS

The results are summarised in Table 1. Note in particular that the results in columns 3 and 4 (Mutant size and L1D) are for the best fitness found by Magpie during its runs. That is, in Table 1, the L1 data cache misses, L1D, reduction is as measured during training. In the cases of the two geographic programs (OLC and H3), while the patches retain their functional ability to pass up to 10 000 holdout tests, the desired improvement in cache performance did not generalise. Indeed it appears after taking care of the noise, there is no real difference in L1 data cache misses between the original and patch code. This is in great contrast to the two data rich image programs, where the patches do give reduced data cache misses on images of the same size as the training data (See Figure 1). Again both Blue and SEEDS patches also generalise in terms of still giving the correct answer on unseen images. However, the right-hand side of Figure 1 (blue ×) shows the SEEDS patch does not give a reduction in L1 data cache misses in images four or more times larger than the training image.

Table 2: Summary of Magpie patches. Average of five runs on each L1 data cache experiment.

	Compile error	Test failed	time out	too big	ok
OLC	72%	5%	0%	0%	23%
H3	61%	4%	0%	2%	33%
Blue	56%	13%	1%	0%	30%
SEEDS	87%	10%	0%	0%	2%

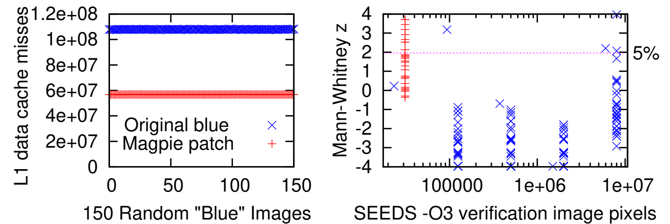

Figure 1: Left Magpie "Blue" patch out of sample performance on 150 hold out images. In all cases the patch incurs only about half as many L1 data cache misses. Right: Magpie SEEDS patch out of sample performance on 150 hold out images of various sizes. The patch tends to reduce L1 data misses on 153×204 images (red +) of the same size as the training image and worse on larger images (blue ×).

On average 72% of OLC patches fail to compile[1], about 5% fail one or more test cases, while the remaining 23% pass all ten fitness tests. Table 2 summarises the statistics for all four experiments. The patterns for H3 and Blue are similar to OLC. However the tabu list used with SEEDS means whenever a patch fails a tabu check, it is marked by Magpie as if it had failed to compile. Hence the low figure for SEEDS' ok column[2]. Note mostly patches which compile, run ok and pass all the tests.

Although Magpie has a nice tool for minifying patches, we did not use it due to the noisy nature of our `perf` based fitness measure.

In all four experiments, the final patch generated the same results as the original program. In the case of the two geographic tools (OLC and H3), the holdout set contains "missing data", i.e. postal addresses without a latitude, longitude location. In these 85 cases OLC produces a default output, while H3 aborts (with a non-zero Unix error code) and an error message. The H3 patches similarly detected and reported the error. On the other 9915 holdout locations the patch similarly returns the same output as the original H3. That is, both OLC and H3 patches pass all 10 000 hold out tests. However in neither case, were we able to show the fitness seen in the Magpie runs, translated to re-running them.

Our results are summarised in Table 1 column 5 "L1D" which gives the percentage reduction in L1 data cache misses during training. Unfortunately, as will be discussed in the next section, the improvements in cache use reported during training with OLC and H3, did not generalise and out of sample, there is no reduction in L1 data cache misses. In contrast on both image processing

[1]Previously we used specialist mutation operators with LLVM IR which ensured all mutants compiled successfully to machine code [8].
[2]In [5] we used a similar idea to test if mutated code is identical by comparing the X86 assembler generated by the GNU gcc compiler.

examples we find significant (non-parametric Mann-Whitney test) reduction in L1 data cache misses. For the Blue bench mark it is 47%. And 1% for the patch to the OpenCV image segmentation SEEDS C++ code on 30 unseen unrelated images of the same size as the training image.

6 DISCUSSION: NON-STATIONARY NOISE, PROFILING AND THE TABU LIST

As mentioned in Section 2.1, motivated by large positive outliers often seen in run time measurement, we have used the first quartile in the fitness function, as it is a robust statistic which is relatively immune to positive outliers. The distribution of cache misses is very noisy and is more symmetric and Gaussian like than expected. Given the apparent symmetry it may be that the median would give a more consistent fitness measure than the first quartile.

However, the data show another little discussed problem: The L1D noise is *not* stationary but subject to some unknown drift. Classical arguments, which assume multiple measurements are independent and identically distributed (IID), suggest increasing the number of measurements n will reduce the impact of noise in proportion to $\sqrt{n}$. However this misses the fact if the noise is non-stationary, then measurements taken during a Magpie run (or indeed during any evolutionary computing EC run) will drift. Not only during the EC run itself, but also when the evolved artifact is used. It may be we need, not only to take multiple L1D measurements per fitness evaluation, but also, during the EC run, to make estimates of the drift. Perhaps this might be done by running some known fixed example code, such as the original unmutated code [5]. If online drift estimation also turns out to be effective for L1D cache measurements, it is likely that effort spent on combating noise during EC runs would be well worth while, as it should lead to better more robust solutions.

Magpie has targeted whole functions that could possibly be called. Particularly in our largest example, H3, this is not sufficient. Since there are both lines of code and pre-set data values that are never used, but Magpie is wasting effort on trying to optimise them. It is common in GI to profile the program to be evolved, and then to target only code that is indeed executed [10]. In these examples, profiling was not used. Indeed the presence of a huge volume of unused data in H3 hints at a further problem. Earlier profiling has concentrated upon code execution. It appears not to have considered that data might be created (and so need maintaining) which is not used by the code during every-day mundane operations.

Of course Magpie has more sophisticated syntax aware approaches and they also might benefit from profile data. Apparently more recent version of Magpie, already rule out simple patches that move #include files or simply swap lines containing just a single curly bracket "}". These may help, but we fear that as usual, fitness driven evolution will find a way to exploit code changes by making other, as yet unthought of, apparently "useless" changes.

The trick of enforcing that each patch make a new semantic change by keeping a tabu list (as was used with SEEDS) will not deal with the problem of wasted effort being spent on generating patches that mutate either unused code or unused data. The tabu list it uses to prevent duplicates is based on the object file created by the compiler. Changes to either unused code or unused pre-set data would change the object file and so although useless would appear to be plausible semantic changes. The problem is exacerbated here with Magpie's local search, as apparently "useless" changes to un-executed code or unused data can, due to the noisy fitness function, appear to be beneficial and so drive search in unproductive directions. However we need to be cautious, as the behaviour of caches is often proprietary and the implications of even "obviously useless" changes is in practice unknowable. For example, even when the compiler issues a warning saying the patch introduces an "unused variable", it may change the machine code it generates. So potentially changing the behaviour of the caches.

7 CONCLUSIONS

We have taken a new open source genetic improvement tool written in Python (Magpie), and industrial C source codes from Google and Uber, C++ code for the "Blue" image processing problem and OpenCV's state-of-the-art image segmentation, and applied it to the never before attempted optimisation of the hardware data cache. For OpenCV's SEEDS a 1% reduction on compiler -O3 optimised code was found, whilst "Blue" L1 data cache misses were almost halved (47%).

ACKNOWLEDGMENTS

Supported by EP/P023991/1 and the Meta OOPS projects.

REFERENCES

[1] Gabin An et al. 2018. Comparing Line and AST Granularity Level for Program Repair using PyGGI. In *GI-2018, ICSE workshops proceedings*, Justyna Petke et al. (Eds.). ACM, Gothenburg, Sweden, 19–26. http://dx.doi.org/10.1145/3194810.3194814

[2] Aymeric Blot and Justyna Petke. 2022. MAGPIE: Machine Automated General Performance Improvement via Evolution of Software. arXiv. http://dx.doi.org/10.48550/arxiv.2208.02811

[3] William Feller. 1957. *An Introduction to Probability Theory and Its Applications* (2 ed.). Vol. 1. John Wiley and Sons, New York.

[4] W. B. Langdon. 2018. *Genetic Improvement GISMOE Blue Software Tool Demo*. Technical Report RN/18/06. University College, London, London, UK. http://www.cs.ucl.ac.uk/fileadmin/user_upload/blue.pdf

[5] W. B. Langdon. 2020. Genetic Improvement of Genetic Programming. In *GI @ CEC 2020 Special Session*, Alexander (Sandy) Brownlee et al. (Eds.). IEEE Computational Intelligence Society, IEEE Press, internet, paper id24061. http://dx.doi.org/10.1109/CEC48606.2020.9185771

[6] William B. Langdon et al. 2015. Improving CUDA DNA Analysis Software with Genetic Programming. In *GECCO '15*, Sara Silva et al. (Eds.). ACM, Madrid, 1063–1070. http://dx.doi.org/10.1145/2739480.2754652

[7] William B. Langdon et al. 2016. API-Constrained Genetic Improvement. In *Proceedings of the 8th International Symposium on Search Based Software Engineering, SSBSE 2016 (LNCS, Vol. 9962)*, Federica Sarro and Kalyanmoy Deb (Eds.). Springer, Raleigh, North Carolina, USA, 224–230. http://dx.doi.org/10.1007/978-3-319-47106-8_16

[8] William B. Langdon et al. 2023. Genetic Improvement of LLVM Intermediate Representation. In *EuroGP 2023: Proceedings of the 26th European Conference on Genetic Programming (LNCS, Vol. 13986)*, Gisele Pappa et al. (Eds.). Springer Verlag, Brno, Czech Republic, 244–259. http://dx.doi.org/10.1007/978-3-031-29573-7_16

[9] William B. Langdon and Bradley J. Alexander. 2023. Genetic Improvement of OLC and H3 with Magpie. In *12th International Workshop on Genetic Improvement @ICSE 2023*, Vesna Nowack et al. (Eds.). IEEE, Melbourne, Australia. http://www.cs.ucl.ac.uk/staff/W.Langdon/ftp/papers/langdon_2023_GI.pdf

[10] William B. Langdon and Mark Harman. 2015. Optimising Existing Software with Genetic Programming. *IEEE Transactions on Evolutionary Computation* 19, 1 (Feb. 2015), 118–135. http://dx.doi.org/10.1109/TEVC.2013.2281544

[11] David R. White. 2017. GI in No Time. In *GI-2017*, Justyna Petke et al. (Eds.). ACM, Berlin, 1549–1550. http://dx.doi.org/doi:10.1145/3067695.3082515

[12] Shengjie Zuo et al. 2022. Evaluation of Genetic Improvement Tools for Improvement of Non-functional Properties of Software. In *GECCO 2022*, Bobby R. Bruce et al. (Eds.). ACM, Boston, USA, 1956–1965. http://dx.doi.org/10.1145/3520304.3534004 Winner Best Paper.