25th UK Workshop on Computational Intelligence (UKCI 2026) Coventry Uni. 9–11 Sep.

Parallel Program Analysis for Linear Genetic Programming

William B. Langdon
Computer Science, University College London, Gower Street, London, UK
w.langdon@cs.ucl.ac.uk http://www.cs.ucl.ac.uk/staff/W.Langdon/

Abstract. Removing unnecessary instructions is commonly performed in a sequential single backward slice through the whole program from the output to its inputs. By keeping track of all registers the program can be broken into 100 or more sections which can each be simplified in parallel. In large evolved linear genetic programming (GP) programs the reduction in intron anti-bloat removal overhead (9.7 fold) can exceed the time to interpret the program, giving effective speeds of up to 726 billion GP operations/second (726 GigaGPOPs^{-1}).

Keywords: deadcode removal · automatic simplification · software optimisation · slicing · evolutionary computing · SBSE · optimisation

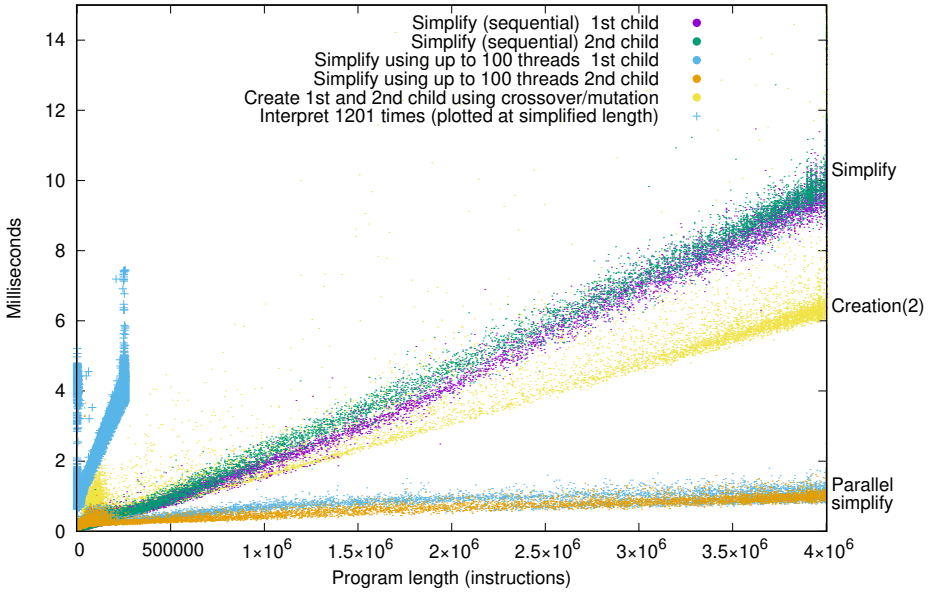

Fig. 1. Time taken by linear GP components. When programs are simplified sequentially (purple and green dots) this can take longer than interpreting the simplified programs 1201 times (+ blue). For long programs parallel simplification is 9.7 times faster (blue and orange dots). Two children created by each crossover (yellow).

1 Introduction: Need for Parallel Analysis

In artificial intelligence it is common for excessively complex solutions to be initially created, e.g. [14, p617]. It may be easier to find a large approximate solution and subsequently simplify and shrink it, than to find a minimal solution from scratch [33]. Smaller solutions are often preferred, either perhaps to make the solution more understandable [1,2] or to reduce computational load. As simplification is typically tedious, labour intensive and error prone, automatic simplification is common.

From the dawn of the integrated circuit, the growth in digital computing has been dominated by Moore's Law [28]. Even before the start of the third millennium, this growth has been manifest in ever more powerful parallel architectures. However typical automatic simplification by backward slicing is sequential. We show how it is possible to **parallelise** Peter Nordin's automatic dead code removal algorithm (as used commercially in Discipulus [7], [8, page 11]). Not only are the resulting shortened programs equivalent to that produced by the serial algorithm but in 99.9935% of cases they are identical. That is, in practice the parallel simplification algorithm is as effective as the serial version.

Rich Lenski's natural Long-Term Evolution Experiments (LTEE) [26], and the availability of large memory systems have both encouraged the exploration of the prolonged automatic evolution of large programs (both as tree structures [20] and more recently as lists of instructions [17,18] using linear genetic programming [3,30]). This in turn has required faster genetic programming systems [15].

The next section summarises our recent work on building digital computing analogues of Lenski's LTEE. Whilst Section 3 details how enormous saving are achieved by automatically removing redundant code and Section 4 explains how this can be done in parallel. The results and implementation are discussed in Section 5 before we conclude in Section 6 that parallel code analysis can give a **ten fold speedup**, effectively removing a performance bottleneck. Our **code is available** in https://github.com/wblangdon/GPengine.

2 LTEE Mackey-Glass 500 generations Benchmark

Rich Lenski's Long Term Evolutionary Experiment (which although he has retired, is still running after more than 38 years) showed that even in a stable environment, bacteria continue to evolve and increase their fitness even after 80 000 generations[1] [5,39]. Using a standard floating point symbolic regression benchmark [14] and our modified GPquick [38,15] we were able to evolve small artificial populations of tree shaped programs up to a million generations, in a matter of weeks rather than years. These genetic programming experiments showed continued evolution but the rate of innovation falling in proportion to the increase in tree size [20].

More recently we have run experiments with linear genetic programming, replacing our version of Andy Singleton's GPquick [38] generational tree based

[1] Homo Sapiens is about 9 300 generations old [37,18].

genetic programming system with one of Peter Nordin's steady state linear GP systems (GPengine [19]) and replacing the continuous floating point regression problem with a short 8 bit integer chaotic time series prediction benchmark (Mackey-Glass [32,19][2]). Using genetic improvement [23,36,35,13,4] multi-core and 512 bit vector instructions [22,16], we were able to run GPengine to 100 000 generations. Again showing continued fitness increase but again with the rate of innovation falling with program length. Detailed analysis of the evolved programs helps to explain the increasing robustness [34] with program length and also information theory sheds light on entropy within digital computing [18]. (Indeed this has lead to proposing entropy as a third implicit oracle in fuzz testing [6].)

In order to reach effective speeds of up to 350 billion genetic programming operations per second ($3.5\ 10^{11}$ GPOPS^{-1}) [18, Fig 7], not only was it necessary to exploit parallel hardware in GPengine's program interpreter (i.e. vector instructions and multi-core), but it was also essential to simplify the evolved bloated programs and remove unnecessary instructions. Lones and Tyrrell [27], in the search for better programs, use remove to modify programs, whereas here we require the smaller programs to be semantically identical. Figure 9 in [18] shows in most runs less than 1 in 15 instructions are actually needed. That is, the work load of the program interpreter can be massively reduced (3–700×) by removing unnecessary instructions. (In genetic programming these are often called introns [31].) However as Figure 1 shows, the time taken to serially analyse the programs can exceed the time taken by GPengine to interpret them on the 1201 Mackey-Glass test cases.

3 Serial Removal of Unneeded Instructions

Figure 2 shows that GPengine's instructions are composed of four `int` fields. There are eight 8-bit read-write registers and four operations (+, −, × and protected division). Each instruction has two inputs, performs one of these four operations and writes its result to the output register. The second input can either be one of the registers or a constant (0 to 127). Before the program is interpreted, GPengine loads the eight registers with one of the 1201 test cases; the program is interpreted from beginning to end and when it finishes its answer is extracted from register A. There are no gaps, branches or loops.

We can create a semantically identical program by removing all instructions which cannot impact the final value of register A. For example, if the last instruction is A=B+C; and the one before is D=E×F; then we can remove the one before and not change the final value in register A. At the program end only register A matters but just before the last instruction the program's output depends only on the contents of registers B and C (and not on register A). That is, scanning the program backwards from its last instruction towards its first, the active set of registers, which can influence the program's answer, has gone from {A} to {B,C}.

[2] GPengine and the discretised Mackey-Glass dataset are available via https://github.com/wblangdon/GPengine.

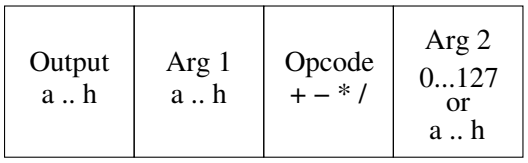

Fig. 2. Format of GPengine instruction. Output register (a..h) = Arg1 operation Arg2.

Figure 3 shows we can simplify the program by scanning it backwards from the program's end to the start. As we go we keep a record of which instructions might impact the program's output and which cannot (stored in array Needed). The array is the same length as the program. We then scan forward from the program's start to its end, appending each needed instruction (pink) to form a semantically equivalent program. This requires two passes through the program.

Highly evolved programs produced by crossover tend to be highly repetitive [19] and as we saw in [18] they can contain a lot of redundancy. Thus making automatic simplification very worthwhile.

Figure 4 details the simplification algorithm's maintenance of the active set during the backward pass and recording which instructions are necessary by setting a per instruction flag in the Needed array. If the current instruction overwrites a register which impacts the program's output (i.e. a register in the active set) conservatively we assume that the instruction is necessary. It is possible for such an instruction not to change the register, e.g. by replacing it with the same value, but we ignore this. We (possibly temporarily) remove the register from the active set but add the instruction's two input registers to the active set. This could involve adding the same register back again. E.g., if C is active then instruction C=C+1, would remove C and then re-insert C into the active set. Whereas instruction C=D−E would mean the output no longer depends on C but does depend on D and E. Also, if the second argument is a constant, e.g. 1, it does not impact the active set and only the first argument is added to the active set. Notice GPengine has no partial update instructions; if an instruction outputs to a register, it overwrites the whole of the register and, unless it is also an input to the instruction, its previous value is lost.

Special cases are possible, but for simplicity we want the analysis to be independent of the actual values in the registers and so the only special case we code for is instructions like C=D−D, which will always set the output register (C) to zero. Which in turn means the output of the program does not depend upon the value of C before this instruction and so we remove C from the active set, without adding anything to it.

Due to having two inputs per instruction, in principle the size of the active set could double each time we step back one instruction. However the number of registers is limited and so it cannot exceed the number of registers (8). Also in practice, the programs are highly redundant and at each instruction, the program's output typically only depends on two or three registers.

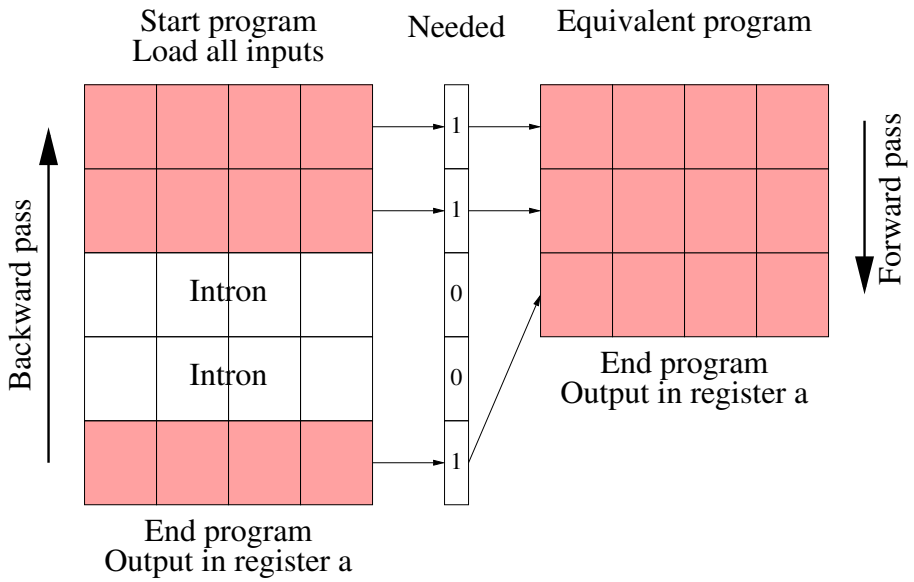

Fig. 3. Starting from the end of program to be simplified (left), simplification scans backwards, calculating Active (Figure 4) and Needed for each instruction. A forward pass copies only needed instructions (pink) to simplified program (right).

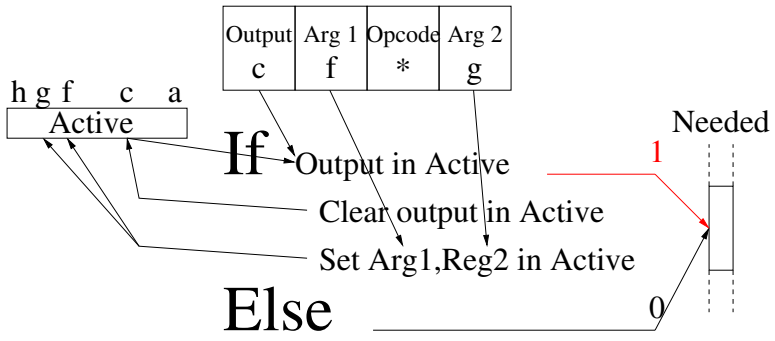

Fig. 4. At the program end only register A impacts its answer, so the active set is initialised to register A. If the current instruction (e.g. C=F×G) impacts any member of the active set, we remove its output register (e.g. C) from the active set and then add its arguments (e.g. F and G) to the active set. Otherwise the instruction is not needed and will not be copied to the equivalent program (Figure 3).

4 Parallel Removal of Unneeded Instructions

From Figure 1 (blue and purple dots), we can see that the time to simplify the programs grows linearly with their length and for long programs it *exceeds* the time to interpret them on 1201 test cases. Due to the ever increasing prevalence of parallel computing, we now describe in detail how we parallelise the simplification algorithm described in the previous section.

Figure 5 shows the simplification work is split more-or-less equally between n threads. (We use the GNU implementation of the C++ pthreads run time library.) Typically the operating system (we use Linux Rocky 9.7) will schedule each thread on a separate CPU core. Thus each thread effectively has its own L2 data cache. In the case of the sequential algorithm, the program's instructions have to be read twice, first in the backwards pass and secondly in the forward copying operation (Figure 3). If the program does not fit into the L2 cache (the AMD EPYC 9554's L2 cache is 1 MB, corresponding to 52 428 instructions) then the L2 cache may need to be refreshed from the 512 MB L3 data cache. As we shall see, the parallel algorithm adds more data and another pass and so our implementation tries to ensure each thread gets a fraction of the simplification task that will fit into the L2 cache. Due to the extra width of the Needed array we limit the work for each thread to 40 000 instructions each.

Whereas when processing the whole program, the serial algorithm knows that only one register will finally impact the program's output, this is not known when it is processed in separate units. Instead each thread of the parallel algorithm assumes any combination of the eight registers may matter and so on their initial backwards pass they each keep track of eight active register sets in parallel. These can be conveniently stored in an $8 \times 8 = 64$ bit wide integer. This scales as $\mathrm{O}(n^2)$ where n is the number of registers. Nevertheless this multiple mask implementation is feasible for many real cases where the number of simultaneously active registers is small. As each thread scans backwards from the end of the part of the program it is responsible for to its start, the use of 64 bit instructions allows it to efficiently keep track of all eight of its active sets in parallel. At the end of the backward scan it stores the last eight active sets in its element of array neededA.

We need two extra arrays, neededA and len2. Both have one element per thread. This element is written to by one thread, but subsequently the whole array can be read by any thread.

The algorithm to keep track of all possible sets of active registers simultaneously is essentially the same as the serial algorithm in Figure 4. Except it is augmented to deal with eight Active sets in parallel and instead of its output to the Needed array being a single flag, each element of Needed becomes an 8×8 bit mask.

The parallel algorithm is conservative in that it can include instructions, which the serial algorithm does not. However, on average all but 64 per million simplifications are identical. Of these, typically the parallel algorithm will remove 94% of the evolved code in contrast to 99%. Also the unneeded instructions only slows down the interpreter, the interpreter still produces identical answers.

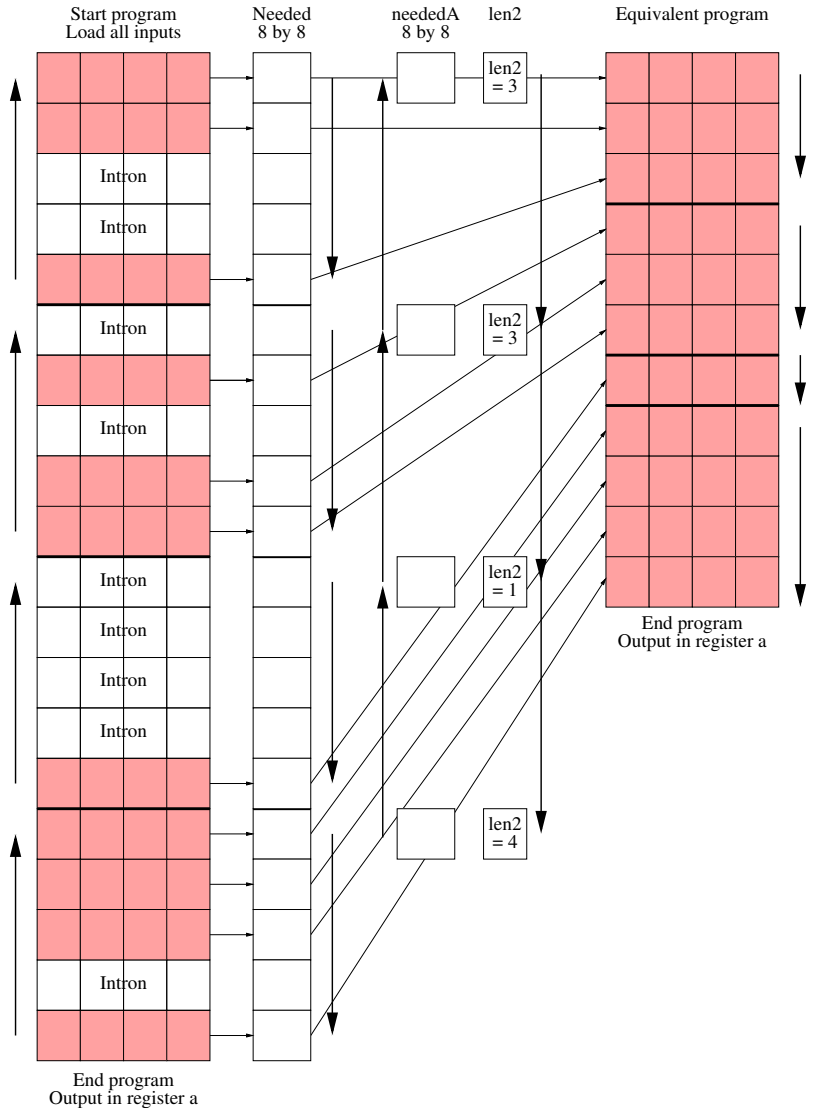

Fig. 5. The program to be simplified (left) is split between up to 100 independent threads (separated by thick horizontal lines). Needed 8 by 8 expands the Needed single bit mask in Figures 3 and 4 to 64 bits. Using 64 bit operations (e.g. _mm_cmpeq_pi8) limits the extra overhead to about two fold. neededA and len2 are small arrays with an element per thread. In the first backward pass, each thread keeps track of eight active sets, setting their own element in neededA to the last active set for all 8 registers. Each thread uses neededA to find a mask appropriate to their code section; sums len2 to locate the start of their section in the output (right) and then copies useful instructions in their section to the output.

When all the threads have completed their backward passes (we use pthread_barrier_wait to synchronise them) they each execute another backwards pass through each others elements of neededA, stopping when they reach their own element. Since in this pass they all start at the program's end (where they know only the value in the output register A matters) they can each use array neededA to determine which registers matter in their segment of the program. This is represented as a 64 bit mask. Notice neededA has one element per thread and thus is usually considerably smaller than the original program and so this backwards scan should be much faster than the first.

Using the mask they produced from neededA, each thread scans their section of the Needed array and counts how many instructions are needed. (For simplicity this is coded as a forward pass, but it could be in any order, indeed it could be a summation reduction.) Each thread writes its count into its element of the len2 array and again the threads synchronise.

The last part is for each of the threads to determine the offset where they should write the important instructions in their segment of the program to the output by summing len2 from its start to their position. (Again this could be done in either order or indeed done as a summation reduction, but it is implemented as a forward pass through len2.) Finally, having determined their offset they reuse their own 64-bit mask on their part of the Needed array to determine which instructions to copy. Again for simplicity this is done as a forward pass, but it would be possible to use vector gather instructions to do it in parallel.

Only when all the threads have finished, does the last one signal the main process that it can continue (again implemented using pthread_barrier_wait but using a second barrier).

To avoid the overhead of creating multiple thread processes each time a program is simplified, all the threads remain active but return to waiting for the main thread to signal that a new program needs to be simplified. (This is implemented using pthread_mutex_lock/pthread_cond_wait and pthread_cond_broadcast.)

5 Timing Measurements and Discussion

The parallel algorithm was implemented in Peter Nordin's C++ GPengine (compiled with GCC 11.5.0) and run on an unladen 256 core 64 bit AMD EPYC 9554 3.1 GHz CPU with 1.5TB RAM.

In all cases the simplification algorithm produces equivalent programs.

For small programs keeping track of all eight registers means the parallel algorithm is two times slower than the sequential version until multiple processing cores are used. Figure 1 makes clear, as expected, that the run time of the serial simplification algorithm rises linearly with the length of the program. The runtime of the parallel algorithm also rises more-or-less linearly with program length. It may be that the apparently linear dependence on program size is down to increasing overhead as more threads are used, possibly due to memory cache and multi-core hardware interactions, such as "false sharing", which

Table 1. On average removing dead code in parallel at least doubles overall speed. Mean effective speeds (10^9 GP operations per second) in ten linear genetic programming Mackey-Glass runs to 100 000 generations.

run	1	2	3	4	5	6	7	8	9	10	
Serial simplification[18]	204	216	268	23	282	240	24	236	106	343	Giga GPOP/sec
Parallel simplification	380	585	639	31	664	548	32	441	214	726	Giga GPOP/sec
Ratio speeds	1.9	2.7	2.4	1.3	2.4	2.3	1.3	1.9	2.0	2.1	

might be ameliorated by aligning data with cache lines, etc. [24,21,10]. Once the programs are big enough to warrant using multiple cores, the parallel version is about ten times faster. Table 1 shows for selected extended runs to 100 000 generations, parallelization makes the simplification overhead almost negligible (confirming lowest lines in Figure 1) and so, depending upon run, approximately doubles the effective speed.

The timing measurements were taken on an otherwise unladen networked server. However the operating system, as well as servicing interrupts, runs various back ground house keeping tasks. Hence, in addition to any hardware variability, the operating system is also a source of elapse time measurement noise.

Figure 1 also shows some structure in the data, for example in the serial code, the second child produced by crossover takes slightly longer to simplify than the first. (In the parallel code, the first child takes slightly longer than the second.) It is not clear why but appears to be an implementation artifact and is not inherent in the children themselves. E.g. reversing processing order, keeps the artifact with the new order, not with the child. The two children are both created before they are simplified (combined time in yellow in Figure 1) but typically each is much bigger than the L2 cache. Therefor the timing differences may be related to the exact ordering of processing and L2/L3 cache interactions. An alternative implementation might have separated processing the two children's crossover/mutation/simplification/fitness calculation to achieve better use of the caches but later (GPengine is more than 26 years old) it proved tricky to isolate the two (particularly the mutation step) without changing the details of how pseudo random numbers are processed and thus impacting the details of the implementation.

Although compiling genetic programming systems are possible [30,9,12], interpreting large GP programs can be very quick and scalable. Here GPengine is generating, analysing and testing programs of 4 million instructions more than 100 times a second (on average 8 milliseconds each). In contrast, both the GNU C++ and Clang 21 compilers failed on programs this big. The GNU compiler takes a minute to compile GPengine programs of half a million instructions. If we could assume compilation time scaled in proportion to program length, i.e. compilation (if feasible) would take 8 minutes, then compilation would be more than 60 000 times slower than GPengine.

6 Conclusions

Although great strides have been made in program analysis, most analysis tools remain resoundingly sequential. To obtain acceptable performance when running evolved register based programs exceeding a million instructions, it is necessary to remove junk before evaluating them. However when run sequentially, the backward slicing analyse exceeded the programs runtime. We have shown such analysis can be performed in parallel without the need for specialist hardware such as graphics cards (GPUs [25,11]) or AI accelerators [29] and here in practice the parallel analysis is as effective as the sequential analysis.

We hope parallel backward slicing will be more widely used. We have concentrated upon parallel dead code removal in very bloated linear genetic programming. Nevertheless we anticipate that it will hold in many large redundant straight through register based programs with a limited number of registers. The basic trick is to limit the overhead caused by making analysis units independent and then to spread them across multiple CPU cores. With 8 registers keeping track of the powerset (64 bits) of all the dependencies is only twice as expensive as keeping track of one and yet allows the work to be split between more than one hundred parallel cores. Each core working on a fraction of the program and then combining their results to seamlessly produce a smaller equivalent program.

On mundane commercial hardware we obtain about a ten fold speedup.

References

1. Angelov, P.P., et al.: Explainable artificial intelligence: an analytical review. WIREs Data Min Knowl **11**(5) (2021), http://dx.doi.org/10.1002/WIDM.1424
2. Bacardit, J., et al.: Evolutionary computation and explainable AI: a year in review. In: Mora, A.M., Esparcia-Alcazar, A.I. (eds.) Evostar 2023 Late breaking abstracts. pp. 38–41. Brno (2023), https://arxiv.org/abs/2403.13950
3. Banzhaf, W., Nordin, P., Keller, R.E., Francone, F.D.: Genetic Programming. Morgan Kaufmann (1998), https://www.amazon.co.uk/Genetic-Programming-Introduction-Artificial-Intelligence/dp/155860510X
4. Brownlee, A.E.I., et al.: Large language model based mutations in genetic improvement. Automated Software Engineering **15**, article number 15 (2025), http://dx.doi.org/10.1007/s10515-024-00473-6
5. Chihoub, D., Pintard, C., Lenski, R.E., Tenaillon, O., Couce, A.: The evolution of robustness and fragility during long-term bacterial adaptation. PNAS **122**(16), e2501901122 (2025), http://dx.doi.org/10.1073/pnas.2501901122
6. Even-Mendoza, K., Obiri, J., Dakhama, A., McMinn, P., Langdon, W.B.: Fuzz3: Entropy as a third oracle. In: SSBSE 2026 - Challenge Track. LNCS, Montreal, https://solar.cs.ucl.ac.uk/pdf/Even-Mendoza_2026_SSBSE.pdf
7. Francone, F.D.: Discipulus Owner's Manual. 11757 W. Ken Caryl Avenue F, PBM 512, Littleton, Colorado, 80127-3719, USA, version 3.0 draft edn. (2001), https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=d2cdee5dac9a6eb903a713ff3329574d2b5b3c9c
8. Francone, F.D.: Dynamics and Performance of a Linear Genetic Programming System. Licentiate degree, Chalmers University of Technology

(2009), https://www.researchgate.net/publication/239581024_Dynamics_and_Performance_of_a_Linear_Genetic_Programming_System
9. Fukunaga, A., Stechert, A., Mutz, D.: A genome compiler for high performance genetic programming. In: Koza, J.R., et al. (eds.) Genetic Programming 1998. pp. 86–94. Morgan Kaufmann (1998), http://metahack.org/gp98-compiler.pdf
10. Geeson, L.J.: Detecting Relaxed Memory Concurrency Bugs in C and C++ Compilers. Ph.D. thesis, UCL (2026), https://discovery.ucl.ac.uk/id/eprint/10224678/1/Luke_Geeson_thesis.pdf
11. Ginsbach, P., Collie, B., O'Boyle, M.F.P.: Automatically harnessing sparse acceleration. In: Pouchet, L.N., Jimborean, A. (eds.) CC 2020: 29th International Conference on Compiler Construction. pp. 179–190. ACM (2020), http://dx.doi.org/10.1145/3377555.3377893
12. Harding, S.L., Banzhaf, W.: Distributed genetic programming on GPUs using CUDA. In: Hidalgo, I., et al. (eds.) Workshop on Parallel Architectures and Bioinspired Algorithms. pp. 1–10. Universidad Complutense de Madrid, Raleigh (2009), http://www.evolutioninmaterio.com/preprints/CudaParallelCompilePP.pdf
13. Kocsis, Z.A., Drake, J.H., Carson, D., Swan, J.: Automatic improvement of Apache Spark queries using semantics-preserving program reduction. In: Petke, J., et al. (eds.) Genetic Improvement 2016 Workshop. pp. 1141–1146. ACM, Denver (2016), http://dx.doi.org/10.1145/2908961.2931692
14. Koza, J.R.: Genetic Programming: On the Programming of Computers by Means of Natural Selection. MIT Press (1992), https://mitpress.mit.edu/9780262527910/genetic-programming/
15. Langdon, W.B.: A trillion genetic programming instructions per second. ArXiv:2205.03251 (6 May 2022), https://arxiv.org/abs/2205.03251
16. Langdon, W.B.: Improving a parallel C++ Intel AVX-512 SIMD linear genetic programming interpreter. ArXiv (9 Dec 2025), https://arxiv.org/abs/2512.09157
17. Langdon, W.B.: Open-ended evolution with linear genetic programming. In: Volpi, N.C., et al. (eds.) IMOL 2025. University of Hertfordshire (2025), http://www.cs.ucl.ac.uk/staff/W.Langdon/ftp/papers/Langdon_2025_IMOL.pdf
18. Langdon, W.B.: Long term evolution experiments with linear genetic programming. In: Banzhaf, W., Ting Hu (eds.) Advances in Linear Genetic Programming, chap. 4. Springer (2026), http://solar.cs.ucl.ac.uk/pdf/Langdon_2026_raLGP.pdf, forthcoming
19. Langdon, W.B., Banzhaf, W.: Repeated sequences in linear genetic programming genomes. Complex Systems **15**(4), 285–306 (2005), http://dx.doi.org/10.25088/ComplexSystems.15.4.285
20. Langdon, W.B., Banzhaf, W.: Long-term evolution experiment with genetic programming. Artificial Life **28**(2), 173–204 (Summer 2022), http://dx.doi.org/10.1162/artl_a_00360
21. Langdon, W.B., Clark, D.: Genetic improvement of last level cache. In: Giacobini, M., et al. (eds.) EuroGP 2024. LNCS, vol. 14631, pp. 209–226. Aberystwyth (2024), http://dx.doi.org/10.1007/978-3-031-56957-9_13
22. Langdon, W.B., Hanna, C.: Improving a parallel C++ Intel SSE SIMD linear genetic programming interpreter. In: Blot, A., Krauss, O. (eds.) Genetic Improvement @ICSE 2026. Rio de Janeiro (2026), http://gpbib.cs.ucl.ac.uk/gi2026/langdon_2026_GI.pdf
23. Langdon, W.B., Harman, M.: Optimising existing software with genetic programming. IEEE TEVC **19**(1), 118–135 (Feb 2015), http://dx.doi.org/10.1109/TEVC.2013.2281544

24. Langdon, W.B., Petke, J., Blot, A., Clark, D.: Genetically improved software with fewer data caches misses. In: Silva, S., et al. (eds.) GECCO. pp. 799–802. ACM, Lisbon (2023), http://dx.doi.org/10.1145/3583133.3590542
25. Langdon, W.B., et al.: Genetic improvement of GPU software. GP&EM **18**(1), 5–44 (Mar 2017), http://dx.doi.org/10.1007/s10710-016-9273-9
26. Lenski, R.E., et al.: Sustained fitness gains and variability in fitness trajectories in the long-term evolution experiment with Escherichia coli. Proceedings of the Royal Society B **282**(1821) (22 December 2015), http://dx.doi.org/10.1098/rspb.2015.2292
27. Lones, M.A., Tyrrell, A.M.: Modelling biological evolvability: Implicit context and variation filtering in enzyme genetic programming. BioSystems **76**(1–3), 229–238 (Aug–Oct 2004), http://dx.doi.org/10.1016/j.biosystems.2004.05.015
28. Moore, G.E.: Cramming more components onto integrated circuits. Electronics **38**(8), 114–117 (April 19 1965)
29. Mrazek, V., Sekanina, L., Vasicek, Z.: Using libraries of approximate circuits in design of hardware accelerators of deep neural networks. In: AICAS 2020. pp. 243–247. Genova (2020), http://dx.doi.org/10.1109/AICAS48895.2020.9073837
30. Nordin, P.: Evolutionary Program Induction of Binary Machine Code and its Applications. Ph.D. thesis, der Universitat Dortmund am Fachereich Informatik, Germany (1997), http://www.amazon.co.uk/Evolutionary-Program-Induction-Machine-Applications/dp/3931546071
31. Nordin, P., Banzhaf, W.: Complexity compression and evolution. In: Eshelman, L.J. (ed.) ICGA95. pp. 310–317. Morgan Kaufmann, Pittsburgh (1995), http://www.cs.mun.ca/~banzhaf/papers/icga95-1.pdf
32. Oakley, H.: Two scientific applications of genetic programming: Stack filters and non-linear equation fitting to chaotic data. In: Kinnear, Jr., K.E. (ed.) Advances in Genetic Programming, chap. 17, pp. 369–389. MIT Press (1994), http://dx.doi.org/10.7551/mitpress/1108.003.0023
33. Olsson, J.R.: Inductive functional programming using incremental program transformation and Execution of logic programs by iterative-deepening A* SLD-tree search. Dr scient thesis, University of Oslo, Norway (1994), http://www.cs.ucl.ac.uk/staff/W.Langdon/ftp/papers/01-report.pdf
34. Petke, J., Clark, D., Langdon, W.B.: Software robustness: A survey, a theory, and some prospects. In: Avgeriou, P., Dongmei Zhang (eds.) ESEC/FSE 2021, Ideas, Visions and Reflections. pp. 1475–1478. ACM, Athens (2021), http://dx.doi.org/10.1145/3468264.3473133
35. Petke, J., Haraldsson, S.O., Harman, M., Langdon, W.B., White, D.R., Woodward, J.R.: Genetic improvement of software: a comprehensive survey. IEEE TEVC **22**(3), 415–432 (Jun 2018), http://dx.doi.org/doi:10.1109/TEVC.2017.2693219
36. Petke, J., Langdon, W.B., Harman, M.: Applying genetic improvement to MiniSAT. In: Ruhe, G., Yuanyuan Zhang (eds.) SSBSE. LNCS, vol. 8084, pp. 257–262. Leningrad (2013), http://dx.doi.org/10.1007/978-3-642-39742-4_21
37. Richard J. Wang, Al-Saffar, S.I., Rogers, J., Hahn, M.W.: Human generation times across the past 250,000 years. Science Advances **9**(1), eabm7047 (6 Jan 2023), http://dx.doi.org/10.1126/sciadv.abm7047
38. Singleton, A.: Genetic programming with C++. BYTE pp. 171–176 (Feb 1994), http://www.assembla.com/wiki/show/andysgp/GPQuick_Article
39. uz Zaman, M.H., D'Alton, S., Barrick, J.E., Ochman, H.: Promoter recruitment drives the emergence of proto-genes in a long-term evolution experiment with Escherichia coli. PLOS Biology **22**(5), e3002418 (7 May 2024), http://dx.doi.org/10.1371/journal.pbio.3002418