Improving a Parallel C++ Intel SSE SIMD Linear Genetic Programming Interpreter

William B. Langdon and Carol Hanna
w.langdon@cs.ucl.ac.uk,carol.hanna.21@ucl.ac.uk
University College London
United Kingdom

Abstract

We use evolution to speedup the Single Instruction Multiple Data (SIMD) parallel interpreter for Peter Nordin's linear genetic programming GPengine. MAGPIE (Machine Automated General Performance Improvement via Evolution of software) is provided with existing hand-optimised source code, its revision history and the Intel 256 bit SSE intrinsics documentation as XML. Fitness is measured via perf's hardware instruction count, while validity and safety are enforced through systematic test cases and memory sandbox protection via Linux mprotect. In a matter of hours local search discovered small, non-obvious program modifications that improve the performance of 128 lines of SIMD C++ code by 2%, without sacrificing correctness. We see genetic improvement can effectively exploit Intel Advanced Vector Extensions (AVX) parallelism, automatically refining complex code that is difficult for human developers to optimise reliably.

CCS Concepts

• **Computing methodologies → Parallel computing methodologies**; **Genetic programming**; • **Software and its engineering → Search-based software engineering**.

Keywords

Linear genetic programming, SBSE, Computer program tuning, MMX, Transplantation, Testing interpreters, Test output distribution, Non-functional GI

ACM Reference Format:
William B. Langdon and Carol Hanna. 2026. Improving a Parallel C++ Intel SSE SIMD Linear Genetic Programming Interpreter. In *15th International Workshop on Genetic Improvement (GI '26), April 12–18, 2026, Rio de Janeiro, Brazil.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3786162.3793233

1 Introduction

Often in modern computer hardware there are vector operations which can process multiple data items in parallel. For example, some Intel 256 bit SSE instructions allow 32 8-bit numbers to be processed simultaneously. More powerful Intel servers also support 512 bit AVX instructions. Since our 3.4 GHz i7-6700 supports SSE and we use GPengine with 8-bit byte data [17], during manual development we tried to use the 256 bit SSE intrinsics available to C++. However this proved difficult and manual development was initially abandoned.

Although it has long been obvious that the future of computing hardware is parallel, the software tools to support general programming on parallel hardware are lacking and fully automatic parallel programming has not been obtained. Instead manual parallel programming remains hard and calls for specialist programmers.

Rather than relying purely on manual effort, Conor Ryan [37, 43] suggested the use of search based tools in the form of genetic programming [13, 36] to aid the generation of parallel code. More recently Genetic Improvement (GI) has been used to speed-up [6] production parallel code (e.g. SSE [26] and CUDA [22, 25]) and even to improve evolutionary computing itself (EC to improve EC! [5]). For example, GI was used to speed up two different traditional tree genetic programming systems: Beagle puppy [31] and GPquick [14]. Here we apply Magpie to the performance critical component of linear genetic programming, towit GPengine's interpreter. For our Mackey-Glass experiments [17], GPengine needs to support eight bit, addition, subtraction, multiplication and (protected) division.

Genetic programming interpreters need high performance [15] but also require operations to be "protected" [13]. In particular they need to protect division by zero and so protected division by zero is often defined to give a result: 0, rather than, for example, throwing an exception. This simple way of protecting division causes implementation problems with parallel vector instructions SIMD [12, 21, 42] which require all data to be treated in the same way. In the manually written SSE interpreter this was eventually resolved by replacing actual division by looking up the answers in a 256 by 256 (8-bit by 8-bits, 65 536) table of precomputed results. Since the look up table involved complicated indexing operations, it was far from clear that the manual code was optimal and so it was initially abandoned in production [17]. Therefore in Section 2.1 we turn to Genetic Improvement. The GI improvements we found, together with those in [16], have now been manually incorporated into the parallel version of GPengine [17].

The next two sections describe the history of our target software: GPengine and then our "out of the box" use of Magpie, particularly setting up the C++ sources it is to optimise as XML files, test cases for a simple program interpreter and sandbox hardening the fitness function. Section 3 describes the general code improvements found and Magpie's performance, which is further discussed in Section 4. In Section 6 we conclude that Magpie can find correct and useful parallel SIMD speed-ups which exploit the available SSE instructions.

This work is licensed under a Creative Commons Attribution 4.0 International License.
GI '26, Rio de Janeiro, Brazil
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2394-0/2026/04
https://doi.org/10.1145/3786162.3793233

2 Experimental Design

In this section, we present the experimental design followed in this study.

2.1 Target System and GI Framework

2.1.1 GPengine. We investigate the genetic improvement of GPengine. GPengine was provided by Peter Nordin, who wrote the commercial linear genetic programming system Discipulus [10, 35]), used in our earlier work [20, 29]. It is a simple clean C++ implementation of linear GP and seemed ripe for conversion to modern parallel computing.

2.1.2 Genetic Improvement Framework. Magpie[1] [4] is a development of PyGGI [11, 38] but is language independent. Released by Aymeric Blot in 2022.

As for setting up Magpie, we present the defaults and the scenario File. Magpie's local search [3, 40] with 100 000 steps was used on three C++/XML source files (see next section). The XML edits were: `SrcmlArithmeticOperatorSetting`, `SrcmlComparisonOperatorSetting`, `SrcmlNumericSetting`, `SrcmlRelativeNumericSetting`, `SrcmlStmtDeletion`, `SrcmlStmtInsertion`, `SrcmlStmtReplacement`, `XmlNodeDeletion<stmt>`, `XmlNodeInsertion<stmt,block>`, `XmlNodeReplacement<stmt>`. Otherwise Magpie defaults (e.g. time outs) were used.

2.2 XML: Documentation, Revision History and Manual Code

srcml[2] version 1.0.0 was used to automatically create three XML files: IntrinsicsGuide.cpp.xml, diffs.cpp.xml and eval.cpp.xml. The first two are read only and are used as feedstock for Magpie, whilst the last contains the GPengine interpreter (i.e. the target SUT itself). Magpie modifies eval.cpp.xml and from the new XML it generates a mutated version of the eval.cpp C++ code, which it attempts to compile and run on four test programs.

2.2.1 Intel IntrinsicsGuide. IntrinsicsGuide.txt[3] documents Intel's C++ runtime library to support its AVX instruction set[4]. It is plain text and was automatically converted into C++ code. For example, the sixteen by 16-bit pabsw instruction is documented as
`__m128i _mm_abs_epi16 (__m128i a)` which is automatically converted to the C++ code `__m128i a = _mm_abs_epi16 (a);`

Also _MM_SHUFFLE was included, via the following code:

```
unsigned char a = _MM_SHUFFLE (z, y, x, w);
```

Comments and assert statements were removed and then srcml was used to generate XML. IntrinsicsGuide.cpp contains 3 393 declarations, such as `__m128i a = _mm_abs_epi16 (a);` and 211 direct library calls, for example:

```
_mm_mask_compressstoreu_epi32(base_addr,k,a);
```

As IntrinsicsGuide.txt is highly repetitive it gives a large but regular XML file which lacks diversity. In particular it contains no C++ comparison operators or simple statements. Therefore, there is nothing for XML edit operations `SrcmlComparisonOperatorSetting` or `SrcmlStmtInsertion` to operate upon. Unfortunately when Magpie randomly choose either and IntrinsicsGuide.cpp.xml, it found no suitable target and aborted. To avoid this, we appended the dummy code `if(a==a) {a=a+1;}`, which contains both a comparison, `==`, and a statement, `a=a+1;`.

2.2.2 GPengine Revision History. Although it is quite common for software engineering experiments to consider commits and revision metadata, this seems rare in genetic improvement [9, 28]. During manual development of the AVX version of GPengine's interpreter 24 snapshots had been saved into RCS [41] over 10 days

Each change was automatically extracted and split into individual changes (89 in total, 1–54 lines each, median 1). Comments and assert statements were again removed and empty files were removed leaving 69 C++ files of between 1 and 51 lines (median 2). These were concatenated in order, giving a single C++ file composed of the individual fragments. srcml was again used, creating a single file, diffs.cpp.xml, holding all the code changes as XML. Again Magpie is forced to treat diffs.cpp.xml as a read only store of potentially useful code fragments which it can incorporate into the evolved interpreter and the functions it calls. Like IntrinsicsGuide.cpp.xml (above), this is enforced by the fitness function rejecting illegal code changes before Magpie is allowed to compile the modified code.

2.2.3 GPengine Interpret16. Interpret16 and its six supporting functions were extracted from the GPengine's C++ source code, comments and assert statements were removed leaving 128 lines of code. Then srcml was run to give eval.cpp.xml.

2.3 Magpie Fitness Function

As mentioned above, the first part of the fitness function is to inspect the modified C++ sources and reject Magpie edits that try to impact read only code (i.e. IntrinsicsGuide.cpp and diffs.cpp, Tables 2 and 3). Magpie now supports read only `ingredient_files` and so these checks are no longer needed. At the same time, after warmup, XML changes which make no difference to the source code, e.g. replace a value with an identical value, are also rejected.

For simplicity, the evolved code is compiled with the test harness as an #include file. We use the GNU C++ compiler (version 11.5.0), with optimisation `-O3`, `-fmax-errors=1` and for our version of AVX (`-march=skylake`). Changes which failed to compile are rejected (refer to Tables 3 and 4).

2.4 Creating Test Cases, Four Random Programs

We randomly create four linear GPengine programs each composed of four instructions presented in Figure 1. Each is given 16 test cases to be processed in parallel, i.e. 64 test cases in total. As we anticipated that protected division would be the most problematic, we insist that all four programs start with division. The three remaining instructions in each program are selected at random. However in the third program we insist, that in total across all

[1]Downloaded 30 Sep 2025 https://github.com/bloa/magpie.

[2]srcML https://www.srcml.org/ is a popular tool for analysing the syntax of computer programming languages, such as C, C++ and Java. Like a compiler, it generates an Abstract Syntax Tree (AST), which describes the syntax of the source code. But instead of compiling the code, srcML generates an XML description of the AST. Magpie, along with many other tools, can manipulate the XML, e.g. editing it to copy, move or delete subtrees within the AST, and then create a new program from the modified XML. By respecting the AST tree structure, many mutated program are syntactically correct, and many compile and are runable Section 3.4.

[3]https://software.intel.com/sites/landingpage/IntrinsicsGuide/# 26 Jan 2017

[4] 512 bit operations, which are not supported by our "skylake" i7-6700 CPU, are ignored.

Table 1: 16 test input x, y pairs for each of the four test programs. *protected division* rows show output of each 0^{th} instruction. Figure 1.

Program																	
1	x = R4	170	255	243	221	150	130	255	154	4	200	96	17	232	202	99	213
1	y = R6	0	1	1	5	2	0	1	1	40	0	122	0	1	0	0	0
protected division x/y		0	255	243	44	75	0	255	154	0	0	0	0	232	0	0	0
2	x = R5	111	255	53	20	28	216	20	169	116	63	160	248	217	82	255	255
2	y = R7	0	1	0	189	127	2	79	1	0	0	2	236	8	0	1	1
protected division x/y		0	255	0	0	0	108	0	169	0	0	80	1	27	0	255	255
3	x = R4	166	118	130	125	255	250	255	205	198	11	224	191	246	130	91	240
3	y = R2	0	0	1	112	1	3	1	0	1	0	3	25	128	3	0	2
protected division x/y		0	0	130	1	255	83	255	0	198	0	74	7	1	43	0	120
4	x = R7	232	28	130	216	12	231	227	196	115	186	151	161	219	204	57	185
4	y = R3	0	54	1	2	171	2	2	125	0	1	0	0	0	0	32	2
protected division x/y		0	0	130	108	0	115	113	1	0	186	0	0	0	0	1	92

Program		Instruction
1 x = R4 y = R6	0	R6=R4/R6
	1	R6=R6/93
	2	R6=R4*32
	3	R6=R6+74
2 x = R5 y = R7	0	R0=R5/R7
	1	R2=R5/116
	2	R5=R7+18
	3	R4=R5/128
3 x = R4 y = R2	0	R2=R4/R2
	1	R7=R2-105
	2	R5=R2+75
	3	R1=R2+24
4 x = R7 y = R3	0	R2=R7/R3
	1	R0=R3*R7
	2	R5=R7/73
	3	R3=R2*118

Figure 1: Four GPengine test programs each with four instructions

four programs, there is at least one of each of the four possible arithmetic operations (+ − × and protected division).

GPengine's instructions comprise an opcode (+ − × or /), an output register and two inputs. One input is always a register and the second is either a constant or a register. There are 8 registers. For each of the four programs we choose uniformly at random two input registers and an output register. The first instruction uses as input the two input registers and uniformly at random chooses its own output register. All three registers are randomly chosen and so are free to overlap.

The second and third instructions similarly randomly choose their output registers. However, they are forced to choose their input registers from registers with known values, i.e. those previously written to or the two program input registers. However the opcode's second input is chosen like GPengine does, i.e. a fraction (20%) are one of these registers with a known value and the rest are constants (uniformly chosen from the range 0 to 127). The last instruction is the same, except its output is forced to be the program's chosen output register.

2.4.1 Creating Test Cases, Edge Cases and Uniform Output Distribution. Care was taken with the first instruction's (protected division) data values. The results of that instruction may be propagated to the remaining three instructions. Originally, to force execution of all paths, we force division by zero uniformly at random for half the data as presented in Table 1. Half the remaining 50% are chosen to force edge cases. One in eight pairs of input values are chosen to give output of 0, of 1, of 255 and a random value between 1 and 255. The remaining four in eight (i.e. 25% of all test cases) are chosen to give an output uniformly chosen between 2 and 127, cf. [33]. Figures 2 and 3 in [16] show examples of distributions of inputs and outputs.

2.4.2 Sandboxing using memory protection (Linux mprotect). "Sandboxes" are techniques to limit the damage that running random code might cause. By default Magpie provides some limited protection against evolved code running amok. These are chiefly; timing out evolved code that is stuck in indefinite loops and by running it in a separate process and thus taking advantage of the operating system's protection. Note eval.cpp does not contain file I/O statements or system calls, which also limits the scope for damage.

Early eval_avx experiments showed mutant code writing to array index -1, i.e. outside the legit range of the array, which C++ does not forbid (it is undefined). Early grammar based GI approaches enforced array bound checks, common in modern programming languages but absent from C++ [14]. Therefore we use Linux mprotect (Figure 2) to provide guards around data which is given to the evolved code. It appears that this is the first time the mprotect mechanism has been used with Genetic Improvement.

Inorder to measure the evolved code's performance, the C++ test harness calls the mutated function Interpret16(). Interpret16() has five arguments: the length of the program, the program itself, its inputs/outputs (using two arrays, an `unsigned char` array and an `int` array), and the protected division lookup table, i.e. in total a scalar and four arrays. Linux (Rocky 9.6) mprotect works on 4KB memory pages. Each of our four arrays is forced to start at a 4KB boundary (Figure 2). Either side of each array the test harness

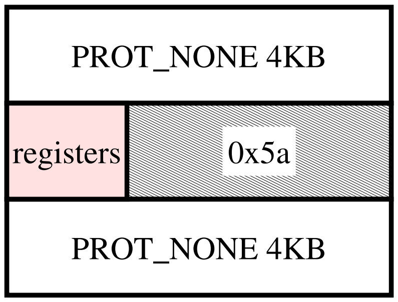


Figure 2: To detect many array indexing errors, the I/O registers are surrounded by 4K byte buffers where either read or write access will cause an illegal access violation SegFault. As the registers do not fill a 4KB window, the excess bytes are padded with 0x5a (i.e. 90, Z, 4 bits set 4 bits clear). After running the mutant, the test harness checks the padding has not been disturbed. The division look up table and the program are similarly surrounded by 4KB guards and inaddition made read only (PROT_READ).

declares empty 4KB arrays, which it uses mprotect to disable any access to. Thus if the evolved code attempts to access array element -1, it will try to access memory in one of the protected 4KB guard regions and the operating system will issue a segmentation error (SegFault), and the test harness will be aborted and Magpie will treat this as a failure to set a fitness and move on to generate the next code mutation. Also the pages holding the program and the lookup table are protected to allow only read access. To simplify the test harness, and avoid a SegFault on `main() return`, rather than undoing all the mprotect calls, the test harness simply uses the Linux process exit routine directly.

Neither I/O register array (`reg` and `registers`) fill a complete 4KB window. Therefore they are both padded up to 4KB and the unused memory is loaded with a non obvious data pattern. The same pattern is loaded into all the registers except those holding the test program's inputs. After each time the evolved code finishes, the test harness checks that the padding pattern has not been changed. Obviously this cannot check if a mutation read memory inside the 4KB window it should not have, but write access is likely to be detected (Status 2 in Table 5). Like SegFaults this is treated as a fatal error, the test harness stops immediately and Magpie does not assign the mutant a fitness.

This protection seems to be good enough. However, it is not 100% fool proof. For example, it only protects memory address, not array indexes. The test programs and the protected division lookup table are multi dimensional arrays and so have multiple indexes; any small misuse of these is liable to incorrectly access a different part of the array, which to the operating system, will appear as a legitimate address within the array and no SegFault will be issued. Similarly a large addressing error may step over the 4KB protection windows, possibly into an unprotected random part of the test harness.

Table 2: Mean out come of 100 003 mutants across five Magpie runs

Magpie cache	27 062	Section 3.2
diffs.cpp failed	25 196	Sections 2.2.2 and 3.4, Table 3
IntrinsicsGuide.cpp failed	21 112	Sections 2.2.2 and 3.4, Table 3
Compilation error	15 115	Section 3.4, Tables 3 and 4
Run time error	5 962	Sections 2.3 and 3.6, Table 5
All tests past	5 549	Section 3.1
RUN_TIMEOUT	3.2	Section 3.6
WARMUP	3	Section 2.3

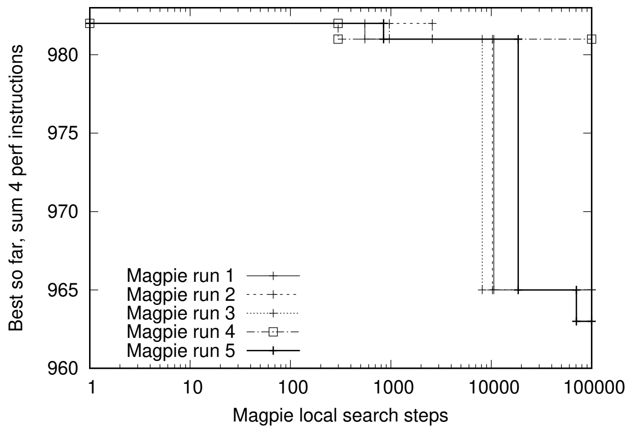


Figure 3: Best fitness in 5 Magpie runs (fewer instructions is better). Log x-scale

2.5 Measuring speed

The evolved code is free to put its answer in either of the arrays (`reg or registers`) corresponding to the randomly chosen output register (see above). Only this register is checked. In any optimisation we need to define a fitness in all circumstances. However, as we use Magpie's local search, if any of our mutants incorrectly calculates any output, the mutation will not be accepted.

Previously Blot [2, 5], ourselves [18, 19, 23, 24, 30] and Bouras et al. [7] have used Linux perf to gather statistics on run time (https://github.com/wblangdon/linux_perf_api). In particular, here we use perf's instruction count as it is far less noisy than elapsed time.

3 Results

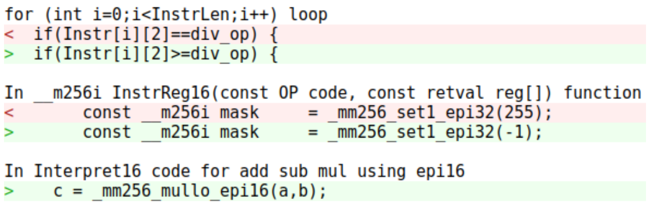
Table 2 summarises the outcome of the five Magpie runs. The next section describes in detail the fastest of the (average of) 5 549 correct mutants per run.

3.1 Code Changes and Fitness Improvement

Figure 3 shows the evolution of the best fitness, i.e. the sum of the number of instructions used by the four test programs in five independent runs. Surprisingly three runs found the same solution, with one other finding only the first part of it and the last extending it, see Figures 4 and 5.

```
SrcmlComparisonOperatorSetting((eval.cpp.xml, operator_comp,
3), >=) | SrcmlNumericSetting((eval.cpp.xml,number,92), -1)
 XmlNodeInsertion<stmt,block>((eval.cpp.xml, _inter_block,
              87), (eval.cpp.xml, stmt, 75))
```

Figure 4: Best solutions. Top: the common Magpie patch takes 965 instructions to execute the test cases. All runs found this solution (except run 4 did not find the `SrcmlNumericSetting` edit). It comprises two edits (separated by vertical bar |). `SrcmlComparisonOperatorSetting` replaces `operator_comp` number 3 which is a `==` by a `>=`. This was found first and saves one instruction. The second replaces number 92 (which is 255) by -1 and saves a further 16 instructions. Bottom: Run 5's additional edit `XmlNodeInsertion` copies statement 87 (`c = _mm256_mullo_epi16(a,b);`) and inserts it at statement 75. It saves one instruction.



```
for (int i=0;i<InstrLen;i++) loop
<   if(Instr[i][2]==div_op) {
>   if(Instr[i][2]>=div_op) {

In __m256i InstrReg16(const OP code, const retval reg[]) function
<        const __m256i mask     = _mm256_set1_epi32(255);
>        const __m256i mask     = _mm256_set1_epi32(-1);

In Interpret16 code for add sub mul using epi16
>     c = _mm256_mullo_epi16(a,b);
```

Figure 5: Best solution makes three C++ code changes (see also Figure 4 and Section 3.1). The first is in Interpret16() at top of the for loop which steps through the program. The second is in function InstrReg16(), which is called by Interpret16() for each of the non division opcodes. The last is at the start of Interpret16()'s non-division code.

The mutations and their speed ups are described in Figures 4 and 5. We can see that not only are the code changes valid but their impact can be explained by examining the C++ source code:

For the first Magpie edit, since `div_op` is the largest opcode, replacing an equality test by a $\geq$ makes no difference to the `Instr[i][2]` vs. `div_op` comparison, nevertheless the edited code is faster.

The second Magpie edit replaces 255 (0xff) by -1 (0xffffffff) and so appears to make `mask` useless, as now all its bits are set (`mask` is intended to ensure inputs to the non division opcodes are byte sized). The compiler appears to have recognised this and optimised it away [39]. However the non division opcodes are followed by a sequence of moves and masks which truncate their 16-bit answers back to 8-bits. Hence the edit speeds the code up without damaging it.

The third edit copies a statement from later in the same section of code. It writes an initial value into variable `c` but all subsequent paths also write to `c` (including the copied one), so the new assignment has no effect. It appears the optimising compiler takes notice of the two identical writes to `c` and reduces the number of instructions by one (fitness improved from 965 to 964).

The peak speed of the evolved GPengine interpreter (excluding crossover mutation etc.) is $4 \times 4 \times 16 \times 3.40\text{GHz}/964 = 903$ million GP operations per second (903 MGPops^{-1}), i.e. 2% faster.

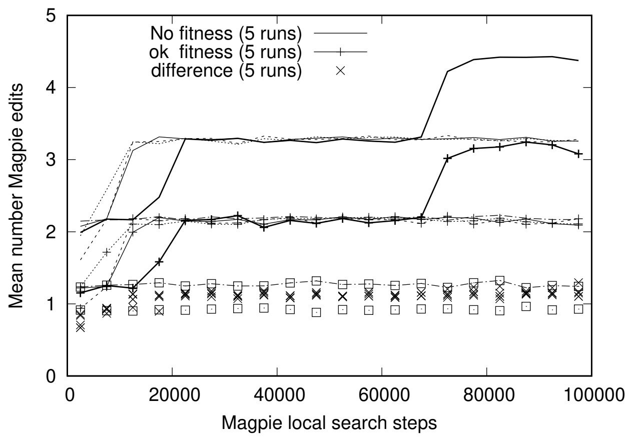


Figure 6: Mean number of edits in five Magpie runs (averages of 5000 steps). Lower crosses (×) show on average the difference between edits which failed fitness testing (no tick marks) and those with fitness (+) is ≈1. Note run 4 with worse fitness □, Figure 3, uses ≈1 fewer edits, whilst at the end the best run 5 (solid line) uses ≈1 more.

3.2 Size of XML Edits; Number of Mutations

In our five runs, 27% of Magpie mutations are not new but instead Magpie is trying again a mutation and so does not recalculate its fitness but instead pulls it from its cache (top row in Table 2). Excluding those cached, mutations typically contain between 1 and 7 individual edits (peak at 3) with on average Magpie concentrating 90% of its new trials on the most popular lengths (typically 2–4 edits).

Figure 6 shows the average length of Magpie edits as it searches. Figure 6 includes Magpie's use of its cache but the data are split into mutants which failed (top) and those which ran and produced the right answers (+). The local search strategy appears to give rise to most additional edits (i.e. increase length by 1) failing either compilation or run time checks (top lines in Figure 6).

The fact that the difference between top and bottom lines (+) in Figure 6 is approximately 1.0, is because of Magpie's local search strategy, which both randomly adds and deletes edits from its active search point. As our edits appear to be somewhat independent, random removal is likely to yield a viable mutant (lower lines in Figure 6), whereas a random additional edit is liable to fail (upper lines in Figure 6).

3.3 Success of XML File Mutations

Table 3 shows, as expected, on average across the five runs, approximately a third of new edits impact each of the three XML files. As explained above in Section 2.3, attempts to change the revision history and the documentation (two lower rows) are rejected. About 25% of changes are internal to each XML file and about 5% each bring changes from the other two files. Almost all new edits are rejected before or during runtime.

In total 7.5% of edits change only the target source code, eval.cpp, compile, run without aborting and give the right answers (2nd

Table 3: Source (columns) and destination (rows) of XML changes. Mean percentages across five runs. Note all attempts to change diffs and IntrinsicsGuide are rejected ("err"). 7 500 Magpie mutations are valid (row "valid").

		eval.cpp	diffs	Guide	total
eval.cpp	err	19.6%	4.4%	4.8%	28.9%
	valid	6.7%	0.7%	0.1%	7.5%
diffs	err	4.9%	24.8%	4.9%	34.6%
Guide	err	4.7%	4.9%	19.4%	29.0%

"valid" row in Table 3). Although one tenth of "valid" edits draw on the revision history (diffs) and about 1% come directly from Intel's documentation (IntrinsicsGuide), no change from either is incorporated into any of the best solutions found (Figures 4 and 5).

3.4 Compilation Errors

Table 4 gives the compilation errors encountered in five Magpie runs. It shows most compilation errors are related to problems with variable names (*id*). Only about 9% are due to syntax errors.

The compilation process (including checks that the Magpie edits have not tried to change IntrinsicsGuide.cpp or diffs.cpp, Section 2.3 above) never timed out. Typically it takes less than 2 seconds.

3.5 IntrinsicsGuide.cpp.xml Transplantation Errors

Section 3.3 (see "diffs" and "Guide" columns in Table 3) shows almost all (≈99%) edits which try to transplant code from Intel's AVX library fail to compile. Commonly this is because variable names transferred with the example library function call either do not exist (e.g. "error: *k* was not declared in this scope") or clash with existing usage (e.g. "error: conflicting declaration *__m128 a*").

It seems our text based conversion to XML is too simplistic. Transplantation work by Alexandru Marginean [1, 32] showed that search in the form of genetic programming can be used to fix up variable name differences between the donor code (here IntrinsicsGuide or diffs.cpp) and the host (eval.cpp). Alternatively, earlier work [14, 27, 28] automatically extracted type information from the Intel documentation and incorporated it into a grammar and used the grammar to enforce type constraints.

3.6 Runtime Errors

Table 5 gives a summary of errors detected after compilation during fitness testing in five Magpie runs. Most run time errors (81.9%, status 1, first row of Table 5) are caused by mutants returning one or more wrong answers. 15.9% (status 139) of run time errors are SegFaults, some of which are illegal reads or writes detected by mprotect (described above in Section 2.4.2). SIGABRT and SIGEMT are described in Table 5. The 166 status 2 errors indicate illegal writes by mutants (described in Section 2.4.2). It appears that the 81 SIGILL errors are not due to AVX-512 instructions, which are not supported by our skylake CPU, but due to mutations of arguments in existing 256 bit AVX (skylake compatible) code.

Table 4: 75 574 Compilation errors across five runs by type

33651	44.53%	*id* was not declared in this scope
15571	20.60%	*id* was not declared in this scope; did you mean *id*?
8698	11.51%	conflicting declaration *id*
4821	6.38%	redeclaration of *id*
3836	5.08%	expected primary-expression before ? token
1908	2.52%	invalid type argument of unary ? (have *id*)
1258	1.66%	*id* without a previous *id*
1223	1.62%	lvalue required as decrement operand
879	1.16%	cannot convert *id* to *id*
815	1.08%	break statement not within loop or switch
790	1.05%	expected primary-expression before *id*
776	1.03%	cannot resolve overloaded function *id* based on conversion to type *id*
365	0.48%	the last argument must be scale 1, 2, 4, 8
203	0.27%	cannot convert *id* to *id* in return
181	0.24%	cannot convert *id* aka *id* to *id*
152	0.20%	unterminated #ifndef
99	0.13%	jump to case label
79	0.10%	narrowing conversion of nnn from *id* to *id* [-Wnarrowing]
75	0.10%	#endif without #if
50	0.07%	invalid types *id* for array subscript
47	0.06%	inlining failed in call to *id*: target specific option mismatch
30	0.04%	decrement of read-only variable *id*
26	0.03%	assignment of read-only variable *id*
24	0.03%	declaration of 'retval reg [512]' shadows a parameter
12	0.02%	the last argument must be an 8-bit immediate
3	0.00%	invalid operands of types 'int [16]' and *id* to binary 'operator*'
2	0.00%	could not convert *id* to *id*

Table 5: 29 810 run time errors across five Magpie runs

Status			fraction	
1		24 419	81.9%	Ran ok but gave erroneous outputs
139	SIGSEGV	4 746	15.9%	SegFault
134	SIGABRT	396	1.3%	Deleted `return` (e.g. in `InstrReg`) corrupted test harness so `assert` failed [8]
2		166	0.6%	Forbidden write outside `reg`
132	SIGILL	81	0.3%	Illegal Instruction
RUN_TIMEOUT		16	$54\ 10^{-5}$	Exceed 30 seconds
135	SIGEMT	2	$7\ 10^{-5}$	Perhaps index error with `reg`

There are almost no runtime time outs. On average only 3.2 of Magpie's 100 000 mutations per run were timed out at run time (the default 30 seconds was used). Nevertheless, for example, a time out was caused by a mutation editing the support function InstrArg to call itself, giving rise to an infinite recursion. Note without a time-out, such an infinite loop would have caused the whole Magpie run to fail.

4 Discussion

Next, we present a discussion based on our experimental results.

4.1 Testing

Even with Magpie's cache, on average it takes 0.5 seconds to generate, check, compile and test each mutant. Most of this is consumed by the optimising compiler. In contrast running the fitness test harness typically takes about 5 milliseconds, although a few erroneous mutants take much longer. Both compilation and test harness time could be reduced further by re-organising so that only changed code is compiled and run for each mutation (Section 2.3). However this would not eliminate Magpie overheads, particularly of creating and checking the mutants.

4.2 Test Suite Effectiveness

Although we test only four short randomly created GPengine programs (Figure 1), with $4\times 16 = 64$ x, y pairs of inputs (Table 1), these are responsible for eliminating 82% of erroneous mutants (top of Table 5). It seems the forced use of all paths through the interpreter, and the use of both input and output edge cases and a wide range of *output* values [33] has been effective at ensuring mutants which pass the test cases are indeed correct.

4.3 mprotect

The Linux mprotect system routine gives a runtime efficient way of eliminating a small fraction of badly behaving C++ mutants. However Table 5 contains 396 `assert` failures, which detected corruption within the test harness (not the evolved code). Although Table 5 also shows 4746 SegFaults, the `assert` failures suggests mprotect is not fully effective. In principle mprotect could be extended to cover all of the test harness but this raises of the practical issues of turning it off again, without causing a SegFault, when the mutated code returns control to the test harness and making reasonable assumptions about how the optimising compiler will layout its use of memory. Nevertheless the mprotect windows either side of critical data structures appears to efficiently detect 90% of array index corruption issues without impacting measurement (by perf) of elapsed time.

5 Threats to Validity

Internal Validity. We performed five independent Magpie runs. All runs produced performance improvements. Together with consistent results from closely related AVX-512 experiments [16], this reduces the likelihood that this is just chance. Correctness is enforced through compilation checks, functional test cases, and runtime sandboxing using Linux mprotect. While this combination is effective, it cannot detect all forms of undefined behaviour in C++, such as subtle out-of-bounds reads within valid memory pages. Nevertheless, the large proportion of mutants rejected due to incorrect outputs suggests that undetected faults among accepted mutants are unlikely.

Construct Validity. We measure performance using Linux perf instruction counts, which are less noisy than wall-clock time and commonly used in genetic improvement studies. However, instruction count does not capture all performance factors, such as cache behaviour or memory bandwidth. The reported 2% improvement therefore reflects reduced instruction count under the tested conditions, rather than guaranteed reductions in elapsed time for all workloads. Fitness is evaluated using four short, randomly generated GPengine programs designed to exercise all interpreter paths and edge cases. While effective for correctness checking, these programs may not fully represent the instruction mix or memory behaviour of larger or more complex workloads.

External Validity. The experiments were conducted on a single hardware platform, compiler, and interpreter configuration, so results may differ on other architectures or toolchains. In addition, the target system is a SIMD-based linear genetic programming interpreter, and the results do not directly generalise to arbitrary software. Nevertheless, the discovered changes are small, comprehensible, and reusable, and have already been integrated into GPengine, supporting their practical relevance.

Conclusion Validity. The observed improvement is modest (2%) but the interpreter is a core bottleneck executed at very high frequency in large GP runs. Replication across multiple runs, deterministic measurement, and consistency with follow-up experiments support the reliability of the results.

6 Conclusions

At present Moore's Law [34] continues to increase computing power by increasing the degree of parallelism. However, although the importance of parallel programming has long been recognised, in general efficient programming of vector computing remains almost impossible for the ordinary human programmer. Nevertheless in about 14 hours, Magpie found small compact non-obvious, comprehensible, correct and reusable improvements (Figure 5) to performance critical parallel vector code, which had taken well over a week to write by hand. The Linux tools mprotect and perf worked well, giving efficient, clean and stable performance measures, leading to code changes we can be confident in. The automatically evolved code improvements (plus AVX512 experiments [16]) were easily integrated into GPengine. This enable linear genetic programming runs of 100 000 generations. Which were needed for long term evolution experiments (LTEE) of continued fitness improvement. They also give insights into software robustness and information theory [17].

Data Availability: XML files and test suites are available via https://github.com/wblangdon/GPengine_eval_SSE256 (The AVX512 versions for [16] are in https://github.com/wblangdon/GPengine_eval_AVX512.)

Acknowledgments

We would like to thank Aymeric Blot and our anonymous referees.

References

[1] Earl T. Barr, Mark Harman, Yue Jia, Alexandru Marginean, and Justyna Petke. 2015. Automated Software Transplantation. In *International Symposium on Software Testing and Analysis, ISSTA 2015*, Tao Xie and Michal Young (Eds.). ACM, Baltimore, Maryland, USA, 257–269. http://dx.doi.org/10.1145/2771783.2771796 ACM SIGSOFT Distinguished Paper Award.

[2] Aymeric Blot and Justyna Petke. 2020. Comparing Genetic Programming Approaches for Non-Functional Genetic Improvement Case Study: Improvement of MiniSAT's Running Time. In *EuroGP 2020 (LNCS, Vol. 12101)*, Ting Hu, Nuno Lourenco, and Eric Medvet (Eds.). Springer, Seville, Spain, 68–83. http://dx.doi.org/10.1007/978-3-030-44094-7_5

[3] Aymeric Blot and Justyna Petke. 2021. Empirical Comparison of Search Heuristics for Genetic Improvement of Software. *IEEE Transactions on Evolutionary Computation* 25, 5 (Oct. 2021), 1001–1011. http://dx.doi.org/10.1109/TEVC.2021.3070271

[4] 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

[5] Aymeric Blot and Justyna Petke. 2022. 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*, Khaled Hadj-Hamou (Ed.). INSA Lyon, Villeurbanne - Lyon, France. https://hal.archives-ouvertes.fr/hal-03595447

[6] Aymeric Blot and Justyna Petke. 2025. A Comprehensive Survey of Benchmarks for Improvement of Software's Non-Functional Properties. *Comput. Surveys* 57, 7 (2025), Article no. 168. http://dx.doi.org/10.1145/3711119

[7] Dimitrios Stamatios Bouras, Carol Hanna, and Justyna Petke. 2025. Optimised Fitness Functions for Automated Improvement of Software's Execution Time. In *SBSE 2025*. Springer. https://solar.cs.ucl.ac.uk/pdf/bouras_2025_ssbse.pdf

[8] Alexander Brownlee, Justyna Petke, and Anna F. Rasburn. 2020. Injecting Shortcuts for Faster Running Java Code. In *2020 IEEE Congress on Evolutionary Computation (CEC)*, Alexander (Sandy) Brownlee, Saemundur O. Haraldsson, Justyna Petke, and John R. Woodward (Eds.). IEEE, Internet. http://dx.doi.org/10.1109/CEC48606.2020.9185708 Special Session on Genetic Improvement.

[9] Khashayar Etemadi, Niloofar Tarighat, Siddharth Yadav, Matias Martinez, and Martin Monperrus. 2022. Estimating the potential of program repair search spaces with commit analysis. *Journal of Systems and Software* 188 (June 2022), 111263. http://dx.doi.org/10.1016/j.jss.2022.111263

[10] Frank D. Francone. 2001. *Discipulus Owner's Manual* (version 3.0 draft ed.). 11757 W. Ken Caryl Avenue F, PBM 512, Littleton, Colorado, 80127-3719, USA. https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=d2cdee5dac9a6eb903a713ff3329574d2b5b3c9c

[11] Gabin An, Jinhan Kim, Seongmin Lee, and Shin Yoo. 2017. PyGGI: Python General framework for Genetic Improvement. In *Proceedings of Korea Software Congress (KSC 2017)*. Busan, South Korea, 536–538. https://coinse.github.io/publications/pdfs/An2017aa.pdf

[12] Hugues Juille and Jordan B. Pollack. 1996. Massively Parallel Genetic Programming. In *Advances in Genetic Programming 2*, Peter J. Angeline and K. E. Kinnear, Jr. (Eds.). MIT Press, Chapter 17, 339–357. http://dx.doi.org/10.7551/mitpress/1109.003.0023

[13] John R. Koza. 1992. *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* MIT Press. https://mitpress.mit.edu/9780262527910/genetic-programming/

[14] W. B. Langdon. 2020. Genetic Improvement of Genetic Programming. In *GI @ CEC 2020 Special Session*, Alexander (Sandy) Brownlee, Saemundur O. Haraldsson, Justyna Petke, and John R. Woodward (Eds.). IEEE Computational Intelligence Society, IEEE Press, Internet, id24061. http://dx.doi.org/10.1109/CEC48606.2020.9185771

[15] W. B. Langdon. 2022. A Trillion Genetic Programming Instructions per Second. ArXiv:2205.03251. https://arxiv.org/abs/2205.03251

[16] William B. Langdon. 2025. Improving a Parallel C++ Intel AVX-512 SIMD Linear Genetic Programming Interpreter. ArXiv. https://arxiv.org/abs/2512.09157

[17] W. B. Langdon. 2026. Long Term Evolution Experiments with Linear Genetic Programming. In *Recent Advances in Linear Genetic Programming*, Wolfgang Banzhaf and Ting Hu (Eds.). Springer. forthcoming.

[18] William B. Langdon, Afnan Al-Subaihin, Aymeric Blot, and David Clark. 2023. Genetic Improvement of LLVM Intermediate Representation. In *EuroGP 2023 (LNCS, Vol. 13986)*, Gisele Pappa, Mario Giacobini, and Zdenek Vasicek (Eds.). Springer, Brno, Czech Republic, 244–259. http://dx.doi.org/10.1007/978-3-031-29573-7_16

[19] William B. Langdon and Bradley J. Alexander. 2023. Genetic Improvement of OLC and H3 with Magpie. In *GI@ICSE 2023*, Vesna Nowack, Markus Wagner, Gabin An, Aymeric Blot, and Justyna Petke (Eds.). IEEE, Melbourne, Australia, 9–16. http://dx.doi.org/10.1109/GI59320.2023.00011

[20] William B. Langdon and Wolfgang Banzhaf. 2005. Repeated Sequences in Linear Genetic Programming Genomes. *Complex Systems* 15, 4 (2005), 285–306. http://dx.doi.org/10.25088/ComplexSystems.15.4.285

[21] William B. Langdon and Wolfgang Banzhaf. 2008. A SIMD Interpreter for Genetic Programming on GPU Graphics Cards. In *EuroGP 2008 (LNCS, Vol. 4971)*, Michael O'Neill, Leonardo Vanneschi, Steven Gustafson, Anna Isabel Esparcia Alcazar, Ivanoe De Falco, Antonio Della Cioppa, and Ernesto Tarantino (Eds.). Springer, Naples, 73–85. http://dx.doi.org/10.1007/978-3-540-78671-9_7

[22] William B. Langdon, Brian Yee Hong Lam, Marc Modat, Justyna Petke, and Mark Harman. 2017. Genetic Improvement of GPU Software. *Genetic Programming and Evolvable Machines* 18, 1 (March 2017), 5–44. http://dx.doi.org/10.1007/s10710-016-9273-9

[23] William B. Langdon and David Clark. 2024. Genetic Improvement of Last Level Cache. In *EuroGP 2024 (LNCS, Vol. 14631)*, Mario Giacobini, Bing Xue, and Luca Manzoni (Eds.). Springer, Aberystwyth, 209–226. http://dx.doi.org/10.1007/978-3-031-56957-9_13

[24] W. B. Langdon and David Clark. 2025. Deep Imperative Mutations have Less Impact. *Automated Software Engineering* 32 (2025), article number 6. http://dx.doi.org/10.1007/s10515-024-00475-4

[25] William B. Langdon and Mark Harman. 2014. Genetically Improved CUDA C++ Software. In *EuroGP 2014 (LNCS, Vol. 8599)*, Miguel Nicolau et al. (Eds.). Springer, Granada, Spain, 87–99. http://dx.doi.org/10.1007/978-3-662-44303-3_8

[26] 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

[27] William B. Langdon and Ronny Lorenz. 2017. Improving SSE Parallel Code with Grow and Graft Genetic Programming. In *GI-2017*, Justyna Petke, David R. White, W. B. Langdon, and Westley Weimer (Eds.). ACM, Berlin, 1537–1538. http://dx.doi.org/10.1145/3067695.3082524

[28] William B. Langdon and Ronny Lorenz. 2019. Evolving AVX512 Parallel C Code using GP. In *EuroGP 2019 (LNCS, Vol. 11451)*, Lukas Sekanina, Ting Hu, and Nuno Lourenco (Eds.). Springer, Leipzig, Germany, 245–261. http://dx.doi.org/10.1007/978-3-030-16670-0_16

[29] William B. Langdon and Peter Nordin. 2001. Evolving Hand-Eye Coordination for a Humanoid Robot with Machine Code Genetic Programming. In *EuroGP 2001 (LNCS, Vol. 2038)*, Julian F. Miller, Marco Tomassini, Pier Luca Lanzi, Conor Ryan, Andrea G. B. Tettamanzi, and William B. Langdon (Eds.). Springer, Lake Como, Italy, 313–324. http://dx.doi.org/10.1007/3-540-45355-5_25

[30] William B. Langdon, Justyna Petke, Aymeric Blot, and David Clark. 2023. Genetically Improved Software with fewer Data Caches Misses. In *Proceedings of the Companion Conference on Genetic and Evolutionary Computation (GECCO '23)*, Sara Silva et al. (Eds.). Association for Computing Machinery, Lisbon, Portugal, 799–802. http://dx.doi.org/10.1145/3583133.3590542

[31] Victor R. Lopez-Lopez, Leonardo Trujillo, and Pierrick Legrand. 2019. Applying Genetic Improvement to a Genetic Programming library in C++. *Soft Computing* 23, 22 (Nov. 2019), 11593–11609. http://dx.doi.org/10.1007/s00500-018-03705-6

[32] Alexandru Marginean. 2021. *Automated Software Transplantation.* Ph. D. Dissertation. University College London, UK. https://discovery.ucl.ac.uk/id/eprint/10137954/ ACM SIGEVO Award for the best dissertation of the year.

[33] Hector Menendez Benito, Michele Boreale, Daniele Gorla, and David Clark. 2022. Output Sampling for Output Diversity in Automatic Unit Test Generation. *IEEE Transactions on Software Engineering* 48, 1 (2022), 295–308. http://dx.doi.org/10.1109/TSE.2020.2987377

[34] Gordon E. Moore. 1965. Cramming more components onto integrated circuits. *Electronics* 38, 8 (April 19 1965), 114–117.

[35] Peter Nordin. 1994. A Compiling Genetic Programming System that Directly Manipulates the Machine Code. In *Advances in Genetic Programming*, Kenneth E. Kinnear, Jr. (Ed.). MIT Press, Chapter 14, 311–331. http://dx.doi.org/10.7551/mitpress/1108.003.0019

[36] Riccardo Poli, William B. Langdon, and Nicholas Freitag McPhee. 2008. *A field guide to genetic programming.* Published via `http://lulu.com` and freely available at `http://www.gp-field-guide.org.uk`. http://www.gp-field-guide.org.uk (With contributions by J. R. Koza).

[37] Conor Ryan and Laur Ivan. 1999. An Automatic Software Re-Engineering Tool based on Genetic Programming. In *Advances in Genetic Programming 3*, Lee Spector, William B. Langdon, Una-May O'Reilly, and Peter J. Angeline (Eds.). MIT Press, Chapter 2, 15–39. http://dx.doi.org/10.7551/mitpress/1110.003.0005

[38] Shengjie Zuo, Aymeric Blot, and Justyna Petke. 2022. Evaluation of Genetic Improvement Tools for Improvement of Non-functional Properties of Software. In *GI @ GECCO 2022*, Bobby R. Bruce, Vesna Nowack, Aymeric Blot, Emily Winter, W. B. Langdon, and Justyna Petke (Eds.). ACM, Boston, USA, 1956–1965. http://dx.doi.org/10.1145/3520304.3534004 Winner Best Paper.

[39] Marta Smigielska, Aymeric Blot, and Justyna Petke. 2021. Uniform Edit Selection for Genetic Improvement: Empirical Analysis of Mutation Operator Efficacy. In *GI @ ICSE 2021*, Justyna Petke, Bobby R. Bruce, Yu Huang, Aymeric Blot, Westley Weimer, and W. B. Langdon (Eds.). IEEE, internet, 1–8. http://dx.doi.org/10.1109/GI52543.2021.00009

[40] Thanatad Songpetchmongkol, Aymeric Blot, and Justyna Petke. 2025. Empirical Comparison of Runtime Improvement Approaches: Genetic Improvement, Parameter Tuning, and Their Combination. In *GI@ICSE 2025*, Aymeric Blot, Vesna Nowack, Penn Faulkner Rainford, and Oliver Krauss (Eds.). Ottawa, 35–42. http://dx.doi.org/10.1109/GI66624.2025.00014

[41] Walter F. Tichy. 1982. Design, implementation, and evaluation of a Revision Control System. In *ICSE (ICSE '82)*, Yutaka Ohno, Victor R. Basili, Hajime Enomoto, Koji Kobayashi, and Raymond T. Yeh (Eds.). IEEE Computer Society Press, Tokyo, Japan, 58–67. https://dl.acm.org/doi/10.5555/800254.807748

[42] Patrick Tufts. 1995. Parallel Case Evaluation for Genetic Programming. In *1993 Lectures in Complex Systems*, Lynn Nadel and Daniel L. Stein (Eds.). Santa Fe Institute Studies in the Science of Complexity, Vol. VI. Addison-Wesley, 591–596. https://www.amazon.co.uk/Lectures-Institute-Sciences-Complexity-1995-08-13/dp/B01F9GXLC8/ref=sr_1_1

[43] Paul Walsh and Conor Ryan. 1995. Automatic conversion of programs from serial to parallel using Genetic Programming - The Paragen System. In *Proceedings of ParCo'95 (Advances in Parallel Computing, Vol. 11)*, Erik H. D'Hollander, Gerhard R. Joubert, Frans J. Peters, and Denis Trystram (Eds.). Elsevier, Gent, Belgium, 415–422. http://www.cs.ucl.ac.uk/staff/W.Langdon/ftp/papers/ryan_1995_paragen.pdf