Genetic Improvement of LLVM Intermediate Representation

William B. Langdon, Afnan Al-Subaihin, Aymeric Blot and David Clark

{W.Langdon,a.alsubaihin,david.clark}@ucl.ac.uk aymeric.blot@univ-littoral.fr
CREST, Department of Computer Science,
UCL, Gower Street, London, WC1E 6BT, UK

Abstract. Evolving LLVM IR is widely applicable, with LLVM Clang offering support for an increasing range of computer hardware and programming languages. Local search mutations are used to hill climb industry C code released to support geographic open standards: Open Location Code (OLC) from Google and Uber's Hexagonal Hierarchical Spatial Index (H3), giving up to two percent speed up on compiler optimised code.

Keywords Genetic programming, GP, linear representation, Clang, static single assignment (SSA), mutational robustness, SBSE, software resilience, automatic code optimisation, world wide location, plus codes, zip code.

1 Introduction

LLVM `https://www.llvm.org/` is now a well established freely available open source software package containing the clang C/C++ language compiler and other tools to support human software engineers with maintaining and developing software. Clang converts program source code to LLVM's intermediate representation (IR). We speedup two programs (one from Google's OLC and the other from Uber's H3) by applying genetic improvement [1,2] directly to IR. LLVM IR is independent of both the source code language and the target hardware. The clean separation of the two has facilitated LLVM support for additional imperative and functional languages (e.g. Fortran, Rust, Haskell) and multiple processor types (e.g. Intel X86, ARM and nVidia). LLVM IR is like a typed hardware-independent assembly language, with a clean separation of code, memory and single assignment registers, meaning all registers are created with a fixed typed value which they keep until they deleted, e.g. on exiting the function containing them. The task of mapping the code, memory and this infinite set of registers into real hardware is left to the compiler backed. The compiler comes with many optimisation passes which transform the LLVM IR. Indeed it is possible to write in C++ additional LLVM IR passes. Although LLVM IR can be stored both in memory and binary files, we use the human-readable text files format.

Genetic Improvement [1,2] applies search-based software engineering [3] techniques, principally genetic programming [4], to existing human written software. Genetic Improvement has been applied to automatic porting [2], transplanting code [5,6] code optimisation [7] and automatic bugfixing. Indeed genetic programming [8] and other optimisation techniques are increasingly being used to automatically repair programs [9]–[16].

In Section 3 we describe our chromosome's representation: a variable length list of 3 different LLVM IR deletion mutations. (Delete is the most common way programmers speed up code [17].) Fitness (Section 4) is based on speed up whilst retaining each program's ability to process the locations of many thousands of zip codes. Our local search GI is detailed in Section 5 and the results given in Section 6 and Table 1. In three cases GI gives modest generalised speed-ups by specialising industrial C code for global locations to a definite application (postal delivery addresses in Great Britain, see Figure 1). However one of the four cases also gives a speed up and passes 9 999 holdout tests but fails the very last holdout test. The GI code changes, generalisation and future work are discussed in Section 7 before Section 8 summarises. But first the next section describes the existing GI work on evolving LLVM IR.

2 Background

We have demonstrated genetic improvement of real world GPU applications [18,19,20,21], including BarraCUDA [22], the first GI code to be accepted into actual use [23]. At EuroGP'19 [24], we showed GI could also speed up parallel CPU code, this time Intel AVX vector instructions were optimised. The resulting GIed RNAfold [25] was accepted into production and like the GI version of BarraCUDA has been downloaded many thousands of times (for example [26]). We applied genetic programming to human written CUDA (or C) source code, whereas Tony Lewis showed GP could be used to evolve nVidia's PTX GPU assembler [27]. More recently Jhe-Yu Liou et al. [28,29,30] have applied grammatical evolution (GE) [31] to LLVM IR for CUDA applications running on nVidia parallel hardware and shown further real world examples where GI finds considerable improvement on hand optimised high level GPU code. GE runs were either for two or seven days. Shuyue Li and Hannah Peeler, et al. [32,33,34] applied linear genetic programming [35] to selecting and ordering existing LLVM optimisation passes (Section 1). Their GP automatically tailors the compiler pass sequence to examples from Thomas Stuetzle's ACOTSP and Parth Shirish Nandedkar's backtrack algorithm for the subset sum problem (SSP).

3 Mutating LLVM IR

3.1 Representation

The changes to the LLVM IR are stored as a list of line numbers and local registers to be mutated separated by semicolons ";". After all the changes have been made to the LLVM IR, Clang converts it to binary executable machine code. Due to neutrality [36,37], the changes may or may not alter the overall program's behaviour.

Where multiple changes to an individual line are possible, e.g. conditional branches, which branch is to be deleted is indicated by appending its number to the line number, separated by a colon ":". For example, 2046:2 means the second branch option on line 2046 is deleted. This is actually implemented by setting the one bit (`i1`) conditional `<cond>` to true. In the following LLVM IR code snippet, the local register variable `%32` is replaced with `1` forcing the code to branch to label %37. Notice LLVM IR local label identifiers, such as %37, have the same format as local register identifiers such as %32.

`br` syntax	`br i1 <cond>, label <iftrue>, label <iffalse>.`
clang .ll line 2046	`br i1 %32, label %37, label %33.`
mutation 2046:2	`br i1 1, label %37, label %33 ;deleted 2`

3.2 LLVM IR `define` functions

The LLVM IR call instruction is used to pass control to LLVM IR subroutines. These are delimited by `define` and `}` and contain local registers, whose names always start with a % character. Local registers names are reused by each subroutine. The closing `}` shows where local registers go out of scope.

3.3 Mutable LLVM IR

Our system is able to mutate the following lines of LLVM IR (unless we have already deleted them):

- store
- call
- conditional branches
- assignments to local registers (except from `alloca`). E.g. `%25 = load i32, i32* %3, align 4`
  There are at least 33 types of mutable assignments to local registers.

  We chose not to make `alloca` mutable since it declares the local register to be a pointer. Although we can delete a pointer by setting it to null, this will usually cause a run time exception. This means that there is usually a group of unmutable local registers at the start of a function. These correspond to the function's arguments and its variables. In the following example the C program entry point `main(int argc, char *argv[]) {...` in the human written source code is translated by the clang compiler into the `define` statement and the following `alloca` assignments for the local registers which correspond to `main`'s arguments and some of its variables.

```
define dso_local i32 @main(i32 noundef %0, i8** noundef %1) local_unnamed_addr #1 {
  %3 = alloca %struct.LatLng, align 8
  %4 = alloca i64, align 8
  %5 = alloca %struct.LatLng, align 8
  %6 = alloca i64, align 8
  %7 = alloca i32, align 4
  %8 = alloca double, align 8
  %9 = alloca double, align 8
```

3.4 Compiling C/C++ etc. to generate LLVM IR

The source code is compiled in the usual way, except instead of generating an object file the clang `-emit-llvm -S` command line option is used to direct the clang compiler to generate an .ll file holding LLVM IR. Similarly the linker is replaced by using the LLVM linker command `llvm-link -S` to create a single file containing all the LLVM IR.

3.5 Selecting which LLVM IR to optimise

The LLVM linker will generate LLVM IR for all the compiled code, including functions which are not called. Either the user can list the functions they wish to be optimised or we can recursively select all the functions which can be called by the program's `main` routine.

3.6 Deleting LLVM IR

LLVM IR `store` instructions and `call` of functions without a return value can be deleted by removing the line. (Actually to improve traceability they are commented out using the LLVM IR comment character ";".)

Assignment statements are *not* deleted. Instead all other occurrences of the left hand local register are replaced with zero. We use LLVM IR's `zeroinitializer` to ensure the zero matches the type of the "deleted" local register. Notice here the fact that LLVM is static single assignment (SSA) means that we are guaranteed that the register is only set once.

Functions which do return a value are actually assignment statements, with their return value being written into a local register. If the return value is not a pointer, to avoid disrupting the LLVM IR naming convention, the call instruction is replaced by a dummy `add` or `fadd` instruction. This adds two zeros together to generate a zero value of the same type as the removed function. To deal with integer, floating point and other types, we use the LLVM IR `zeroinitializer` to generate zero. Thus ensuring the local register has the same type as before but the function is not called. Note when the LLVM IR is compiled to executable binary code, the clang compiler may optimise the code and so remove unneeded instructions and memory.

If the function (which may be a system call) returns a pointer, then the call instruction is replaced with an `alloca` instruction, again ensuring the local register's type is unchanged. As with other assignments the local register is flagged as having been deleted and replaced by zero (i.e. null) everywhere else in the LLVM IR. Again for traceability, the original LLVM IR code is retained as a comment.

As mentioned above (Section 3.1), conditional branches are deleted by forcing the condition to be either true or false. Unconditional branches `br` and return `ret` instructions cannot be deleted.

By taking care of both syntax and types we ensure the mutated code remains valid LLVM IR and it compiles into executable binary code.

4 Fitness Function

There are multiple aspects of a mutation's fitness: 1) could we perform the mutation, 2) does the mutated LLVM IR compile without error, 3) is the mutated binary code different from the original version, 4) does the mutant program run ok on each test case, 5) does it produce output files, 6) how different are those outputs from the outputs of unmutated code, 7) how long does it take.

In these experiments all mutants pass (1) and (2). In a few cases, e.g. due to the clang compiler's optimisations, although the LLVM IR is changed, the binary machine code executable file is identical to that of the human written code (3). Since we already know their performance will be identical to that of the original code, such mutants are discarded without fitness testing[1]. Note, except for using `-S -emit-llvm` to generate the LLVM IR, we use the same compiler options as are normally used to compile the program.

In check (4), both the framework running the mutant on each of the test cases (see Sections 4.3 to 4.5) and the mutant itself, can signal a problem via the usual unix exit status. In either case, the framework attempts to continue as usual. For (5) and (6) it will attempt to inspect the expected output files (one per test case) and compare them with those produced by the unmutated human written code, which (on the test cases) always successfully terminates. However due to the exit status error, fitness for that test case will be reduced. Finally (7) the GI framework will extract timing output generated by the unix perf command (see Section 4.2).

The GI framework will attempt to run the mutant program on all the test cases, summing the fitness for each test case. Section 4.2 describes how times for individual test cases are combined to lessen the impact of noisy outliers.

4.1 Test cases for Google's OLC and Uber's H3: GB post codes

Both Google's Open Location Code (OLC)[2] and Uber's Hexagonal Hierarchical Geospatial Indexing System (H3)[3] are open industry standards. We obtained their human written sources from GitHub (total sizes OLC 14 024 and H3 15 015 lines of source code, LOC). Both OLC and H3 include C programs which convert global positions (i.e. pairs of latitude and longitude numbers) into their own internal codes (see Table 1). For OLC we used their 16 character coding and for H3 we used their highest resolution (`-r 15`) which uses 15 characters. Rather than work on abstract locations, we use as test cases the actual locations of homes and commercial premises.

[1] In [38] we used a similar idea to test if mutated code is identical by inspecting X86 assembler generated by the GNU gcc compiler. Also Mike Papadakis et al. [39] compared compiler output to look for equivalent mutants.

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

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

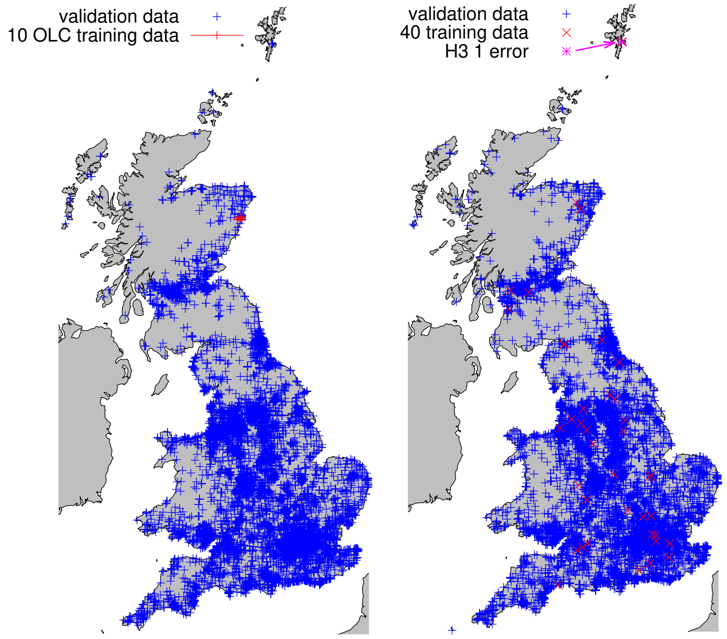

Fig. 1. Left: Ten OLC training points randomly selected in the neighbourhood of Aberdeen (red). + holdout set (blue) GB post codes (zip codes). Right: Forty training points randomly selected from ten H3 runtime classes (see also Figures 2 and 3). + holdout set (blue), locations of ten thousand random GB post codes (no overlap with H3 training or OLC (left) holdout data). Both OLC mutants and H3 -O3 pass all their holdout tests.

For Google's OLC, the location of the first ten thousand GB postcodes (zip codes) were obtained[4]. For training (see next section) ten pairs of latitude and longitude were selected uniformly at randomly (see Figure 1). 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. However the H3 utility comprises about 13 times as much C code as the OLC utility does (see Table 1). Although significant speed up could be obtained with the same 10 training data as OLC and for more than 90% of post codes the mutated programs generalised, it was decided to increase the number of training data to 40 selected from a much wider pool of GB post codes (see red × right of Figure 1). To get not only a geographic spread but also a spread of difficulty, the original code was timed on 10 000 uniformly chosen post codes and divided into ten classes see Figure 2). Where possible, four points were chosen uniformly at random from each class. Some run time classes

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

were empty (see Figure 3), in which case their training points were allocated to the next slower non-empty class (shown with crosses in Figure 2).

Although (see previous section) a series of fitness scores were defined to deal with partial matches between the correct and the mutant's output. For brevity they are omitted, since in practise all worthwhile mutants produced exactly the required output on all tests. Similarly if the mutant aborted or itself reported an error, on any test case, it was discarded. The secondary aspect of fitness is run time.

4.2 Counting instructions with `perf stat -e instructions -x,`

In some previous GI work we had used actual run time, e.g. [40]. However in [40], we evolved subroutines which could be called directly by our GI system, whereas here we will test complete programs and so need the unix process time. Also runtime is notoriously noisy and we had previously found success using the unix perf tool, e.g. [41], which easily reports statistics for a complete program, including reporting the number of instructions actually used.

Although `perf stat -e instructions` is much less variable than elapse time, we run each mutant ten or forty times. Since run time typically has a noisy long tailed distribution [40], we sort the ten (H3 40) instruction counts and use the 3rd (11th) fastest. This means there are about three times as many (7, 29) larger counts than there are smaller (2, 10). Thus giving a stable average. We need not worry about a systematic bias, as the fitness function only ever compares the average count with other average counts obtained in the same way.

4.3 Sandboxing to prevent running mutations causing harm

In software engineering mutation testing [42] there may be the possibility of rogue mutants doing unwanted things, such as writing to unprotected files. Therefore it may be necessary to protect system calls, such as fopen, or to run the mutants in a sandbox. In our experiments, the mutations are constrained and, for example, they cannot change file names but we still needed to guard against mutants consuming excessive resources, such as running into indefinite loops or the output file becoming too big (see next two sections).

4.4 Timeouts to stop poor mutants delaying search

In these experiments each test case normally completes in well under a second. We used two Linux tcsh commands to impose a limit on mutants:

`limit cputime 2` The tcsh limit command can impose run time limits on many resources consumed by a unix process. `limit cputime 2` prevents a process using more than two seconds of CPU time. Sadly this was not sufficient, as during development, a mutant managed to open an empty input (stdin) and then wait indefinitely (consuming no CPU time) for the first byte to arrive.

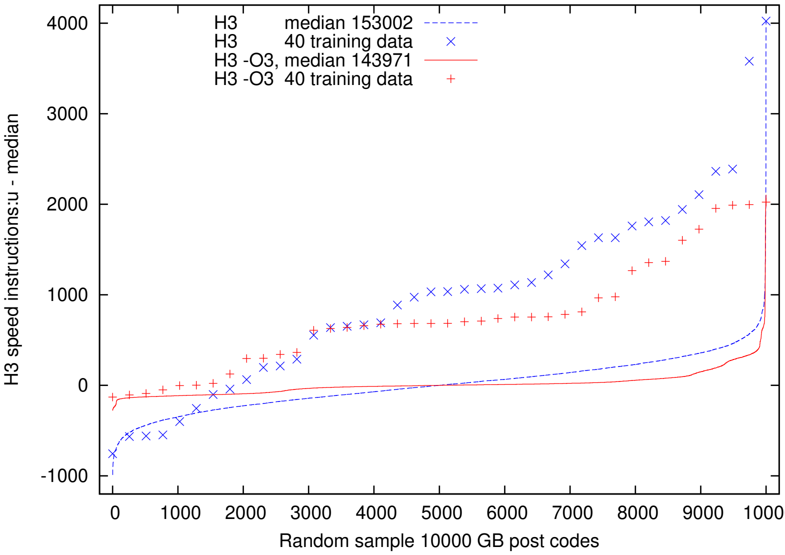

Fig. 2. Lines show distribution of H3 run times (perf instruction counts), when compiled without compiler optimisation (dashed blue) and with -O3 (red solid). GP training points are allocated to try to cover full range of H3 run times (see also Figure 3). To plot -O3 data on the same vertical scale, run times have been adjusted by their median (5000) value (153 002 instructions, 143 971 with -O3).

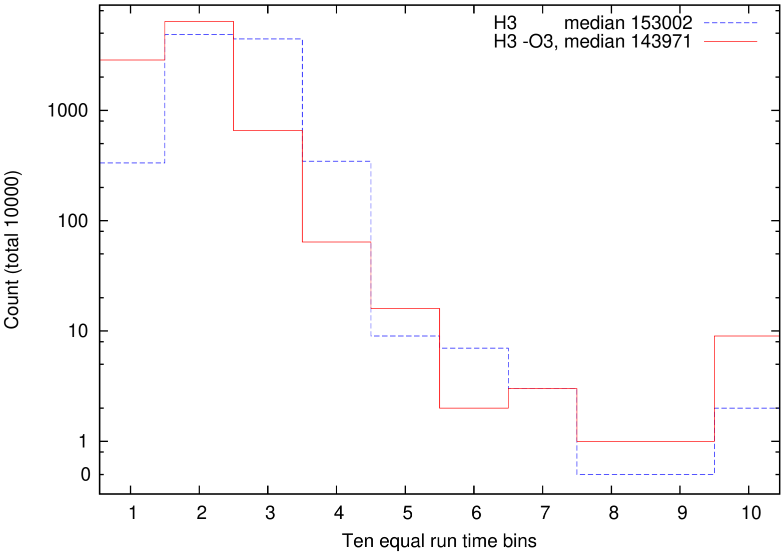

Fig. 3. H3 run time distributions (see lines in Figure 2) separately split into ten equal run time ranges. Note post codes where H3 is slow (right hand side) are rare. To include a wide range of difficulty in the training data, where possible 4 training points are chosen randomly from each bin. The chosen data are plotted as crosses in Figure 2. Note non-linear vertical scale.

`timeout 2` This timeout command aborts a process if it runs for more than a couple of seconds.

Although it should not be necessary to use both `limit cputime 2` and `timeout 2`, it seemed safer to retain both and the overhead of using both appears to be negligible.

Where a system imposed limit is exceeded and the process terminated, the fitness function will detect the non-success exit status and give the mutant a zero score on that test case.

4.5 Limiting output size to avoid filling disk or exceeding disk quota

In our example, typical output size is 18 or 19 bytes. Nonetheless we use the linux tcsh `limit filesize 1M` command to ensure a rogue mutant does not fill the disk. The 1 megabyte limit is deliberately excessive, since it will avoid the disk filling problem and we found (with Centos 7) limits close to the expected output size could trigger the file size exceeded exception early.

5 Hillclimbing Search

The complete C source is compiled to LLVM IR but the search is focused on the functions in the LLVM IR which can be indirectly called by the program's `main` C entry point. Depending which geopositioning example we are considering, this gives between 219 and 2113 possible individual mutations (column 6 in Table 1). In the first pass we test them all one at a time. Between 37% and 63% of individual mutations pass all the test cases ($\frac{\text{column 7}}{\text{column 6}}$ in Table 1).

In the second (hill climbing) pass, we start from the fastest individual mutation and try adding the first of the other non-fatal mutants. For example, in one run of H3 compiled with -O3, the fastest individual mutation which still passes all the test cases was 10633:1. This mutates the conditional branch instruction on LLVM IR line 10633 (see Section 3.1). The first non-fatal mutation is on line 1972 (speed up 7 instructions). Adding it gives the double mutation 10633:1;1972;. However the combination of both mutations does not improve on mutation 10633:1 by itself. Therefore we do not include 1972, and instead move on to consider the next non-fatal mutation. The first additional mutation to give a speed up is that on LLVM IR line 2044. So we add 2044 to our current search point (giving 10633:1;2044;) and try the next non-fatal mutant (line 2045). In this way we work through all the non-fatal mutations in a single pass. This is fast, O(n), but does not consider all possible combinations, O(2^n). Nevertheless it does find a combined set of mutations which give still further speed up on pass one and continues to pass all the test cases.

6 Results

The results are summarised in Table 1. For both OLC and H3 we conducted two experiments. Firstly with default parameters for the clang C compiler and secondly using the -O3 optimisation flag.

Table 1. Size of C sources for Google's OLC and Uber's H3 code optimised. The rows labeled -O3 are for the same programs but when compiled with LLVM 14.0.0 clang's optimisation flag -O3. Columns 2 and 3 refer to the source files, including C .h header files. The (used) column gives the size of C code, excluding header files, to be optimised. Column 5 is the total size of the intermediate representation, whilst column 6 shows how many .ll lines we try to optimise. Column 7 gives the number of mutants which may run faster or slower, but do not change the program's output. The complete optimisation (columns 8–10) is assembled from these. Speedup is the average reduction in unix perf's instructions per test case. Average run times are for 1 core on a 3.6GHz Intel i7-4790 desktop.

C	files	LOC	(used)	LLVM IR		no output-	Mutant			GI duration
				total	mutable	change	size	speed up	holdout	
OLC	4	586	(127)	2546	294	141	2	698	682	5 minutes
-O3	4	586	(127)	2248	219	82	5	683	681	7 minutes
H3	43	5708	(1615)	19415	2113	955	51	2897	2631[a]	2.5 hours
-O3	43	5708	(1615)	15680	1762	1108	46	3272	2985	3.25 hours

[a] One holdout test failed

For OLC clang generates 2546 lines of LLVM IR (2248 with -O3). Considering only mutable LLVM IR in functions which are reachable from `main` (see Section 3.5), there are 294 (-O3 219) possible individual mutations[5]. Of these 141 (-O3 82) can be individually applied without impacting OLC's output on the ten training cases. The hill climbing search described in the previous section, finds six `2323%11;2185;2052%168;2329;2356;2323%25;` (-O3 15) which together give an average reduction of 698 (-O3 683) instructions. Of these four can be removed without changing OLC's performance, leaving just two `2323%11;2052%168;` (-O3 5), which together give a speedup of 698 (-O3 683). When tested on ten thousand uniformly chosen post codes (excluding those used to select the training data) the combined mutation gives an average reduction in number of instructions of 682 (-O3 681).

The results for H3 are given in the lower two lines of Table 1. The hill climbing search described in Section 5 finds 89 (-O3 113) individual changes which together give an average reduction of 2874 (-O3 3267) instructions. Again some (38, -O3 67) can be removed without changing H3's output and with little impact on its speed (see columns 8 and 9 in Table 1). When tested on ten thousand uniformly chosen post codes (excluding all those used to select the training data) the combined mutation gives an average reduction in number of instructions of 2631 (-O3 2985). However, although the 46 changes to H3 -O3 LLVM IR pass all 10 000 holdout tests, the 51 changes to LLVM IR compiled without -O3 fail just the last holdout test ZEX XXX (see purple `*` in Figure 1). ZEX XXX is unusual in being almost the last post code (99.97%$^{\text{th}}$). Also, H3's run time on ZEX XXX of 156 466 is 3845 above the average and so there is only

[5] Mutable conditional `br` instructions give rise to two mutations per line, Section 3.1.

one of the forty H3 training points which has a similar run time (see Figure 2). Section 7.2 suggests ways to perhaps further increase the number or diversity of the training data, which might increase the mutant code's generalisation.

7 Discussion

7.1 Types of Improvement Found

OLC two deletions 2323%11;2052%168; Fortunately OLC compiled without -O3, gives us a simple example to start with. The two changes are independent.

2323%11 deletes local register %11 (from the scope defined by the `main` routine on line 2323). This has the effect of disabling a sanity check:

```
fprintf(stderr,"need two arguments latitude longitude\n");
  return 1;
```

which, as all the tests are well formed, is never invoked. Oddly this gives a greater speed up than the equivalent conditional branch mutation 2334:1 which disables the preceding `if(argc != 1+2){`. Although both mutations enter pass two (Section 5), 2323%11 is first in the list and as adding 2334:1 gives no additional speed up, only 2323%11 is retained.

2052%168 removes the line setting local register %168 (declared in the scope starting on LLVM IR line 2052, which is where function print_OLC_Encode is defined). In the LLVM IR produced by clang with no optimisation, this has the effect of removing the call to `printf("\n")` at the end of print_OLC_Encode. (With -O3 clang converts `printf("\n")` into a more efficient putchar(10) but gives a much more complicated mapping between C and LLVM IR.) Removing `printf("\n")` reduces the size of the output by one byte and so reduces the OLC's run time but makes no difference to its functionality. Again the mutated OLC code has been made slightly faster by removing non-essential code.

OLC -O3 five deletions 2148:2;1895%6;1905:2;1895%40;1895%178;
1895%178 deletes local register %178 (from scope starting on line 1895 print_OLC_Encode). This has the effect of deleting the call to putchar(10) mentioned in the previous section as having been generated by clang -O3 to replace `printf("\n")`. I.e. GI has found the same optimisation in more convoluted LLVM IR.
2148:2 `br i1 1, label %7, label %4 ;deleted 2` disables the `if(argc != 1+2){` mentioned in the previous section. So again skipping the number of arguments sanity check in `main`. With the -O3 optimised code, the mutation equivalent to 2323%11 is retained in the second pass (Section 5) but then is correctly found to be redundant and eliminated in the cleanup pass (Section 6). The three other deletions are all in print_OLC_Encode.

print_OLC_Encode is only called by `main` and its third parameter is always 16, whereas kMaximumDigitCount is 15. Thus:

```
  if (length > kMaximumDigitCount) {
    length = kMaximumDigitCount;
  }
```

always sets length to 15. Mutation 1895%6 deletes local register %6 (i.e. sets it to zero). This has the knock on that the following `%7 = select i1 %6, i64 %5, i64 15` is forced to set register %7 to 15 (i.e. kMaximumDigitCount). Effectively eliminating `if (length > kMaximumDigitCount)`.

With -O3 clang inlines the calls to `adjust_latitude(lat, length)` and `normalize_longitude(lon)`. However both `lat` and `lon` are always already normalised. Therefore the conditional branch mutation 1905:2 is able to force the branch on line 1905 to effectively skip over much inlined code.

1895%40 deletes local register %40 so removing the comparison `lon_degrees < -kLonMaxDegrees` of the first while loop in `normalize_longitude` and forcing the following conditional branch to always jump to over the never needed adjustment `lon_degrees += kLonMaxDegreesT2;`

Thus again GI has sped up OLC (above that obtained by clang's -O3 optimisations) by specialising it to the training data and removing some internal checks and branches which either can never be taken or which must be taken. It seems that GI has been helped by clangs -O3 extensive inlining greatly expanding the LLVM IR in the print_OLC_Encode scope. Although the LLVM IR with and without -O3 are different, it is not yet clear why GI was unable to exploit the same opportunities when the LLVM IR was split into several smaller called functions with equivalent functionality, or if there are further similar but unexploited opportunities.

H3 51 deletions speedup 2631, H3 -O3 46 deletions speedup 2985 There are too many H3 improvements to describe them all in detail. Several follow the same ideas as OLC, with redundant operations being removed. Such as removing calls to normalise data which are always already normalised and simplifying H3 command line processing. (For example, the post code tests never invoke H3's "help" command line option.)

As an example consider 10508%74, which in one run gave the biggest individual saving (872 instructions). Again with -O3, clang inlines functions. In particular, doCoords (which converts the inputs, given in degrees, into radians and so must be called) is inlined into `main`'s LLVM IR. Mutation 10508%74 forces local register %74 to be zero, so causing the immediately following conditional branch to always call doCoords. The direct mutation 10633:1 has the same effect, but due to noise in perf gets a speed up measurement of 871 and so 10508%74 is preferred. (In the run described in Section 5 mutation 10633:1 was the fastest.) Naturally 10633:1 gives no additional improvement in the hill climbing phase and so is dropped in favour of the conceptually slightly more complicated (but equivalent) 10508%74 mutation.

7.2 Discussion: future work, co-evolution, perf, fitness landscape

The H3 example is an order of magnitude bigger than the OLC. In retrospect, we should have been surprised if the simplistic choice of training data which works so well for OLC was sufficient for H3. Although the open source makes white box software engineering techniques (e.g. fuzzing) to target edge cases and branch coverage feasible, we have, so far tried to avoid in-depth analysis of the program's internal behaviour. Instead we used external measures, such as run time and geographic spread to increase the fraction of "difficult" cases in the training data, Figure 2. It seems further improvements in the training data may be necessary, in which case an antagonistic co-evolutionary approach, perhaps where a population of training points is optimised to adversarially increase run time, might be beneficial.

As expected [41], when perf is used to measure whole program performance, it offers considerable noise reduction compared to the unix time command. However even perf's count of instructions executed is noisy. It is also subjected to systematic variation, e.g. between test cases, and also due to changes in the program's environment. For example, systematic changes in the harness running the test program, such as when more data are held in unix global environment variables as training progresses, can increase measured run time.

We have targeted only functions that can be called (see Section 3.5). In principle it should be possible to use LLVM profiling tools to target more finely individual LLVM IR instructions that are executed, possibily multiple times, during training. At present we use run time as a final pass to eliminate individual changes which appear to have no or little effect (see Section 6). This could be because they are never executed or because their beneficial effect is also obtained by other changes in the combined mutation. Although noisy, using run time potentially allows a (Pareto) tradeoff between size of the GI change and the benefit it gives [43][6].

We have considered only a few types of mutation. Many others, swaps and crossover could be included. It seems that in a few cases individual changes are independent and can all be applied to give each's own improvement. However some changes interfere, giving rise to an epistatic fitness landscape [36,44] for which genetic search may be suitable.

8 Conclusions

LLVM is a mature open collection of tools to support human programmers working with high level language comprised of compilers, linkers, profilers, debuggers and other tools. Although initially targeting C and C++ and Intel x86, the range of languages, supported hardware and analysis tools continues to grow. LLVM IR offers an intermediate target for genetic improvement (GI) which is independent of both the source code language and underlying hardware. Its simple line originated syntax offers a universal GI target without the need for specialised grammars. However the current black box training needs strengthening, perhaps using additional LLVM tools, or white box analysis. So far we have taken two examples from industry standard codes (Google's OLC and Uber's H3) and shown GI on IR can in a few minutes or hours (rather than days or weeks) give 0.5% (OLC) and 2% (H3) speed up even on compiler optimised code.

[6] Our LLVM IR representation allows ready calculation of how many lines of LLVM IR are impacted, as an alternative to counting the number of mutations.

Acknowledgements

We are grateful for help from H.Wierstorf (gnuplot) and F.Pfenning (ϕ nodes). Funded by the Meta Oops project.

References

1. Petke, J., et al.: Genetic improvement of software: a comprehensive survey. IEEE TEVC 22(3), 415–432 (2018), http://dx.doi.org/doi:10.1109/TEVC.2017.2693219
2. Langdon, W.B., Harman, M.: Evolving a CUDA kernel from an nVidia template. In: Sobrevilla, P. (ed.) WCCI. pp. 2376–2383 (2010), http://dx.doi.org/10.1109/CEC.2010.5585922
3. Harman, M., Jones, B.F.: Search based software engineering. Information and Software Technology 43(14), 833–839 (2001), http://dx.doi.org/10.1016/S0950-5849(01)00189-6
4. Poli, R., Langdon, W.B., McPhee, N.F.: A field guide to genetic programming. Published via http://lulu.com and freely available at http://www.gp-field-guide.org.uk (2008), http://www.gp-field-guide.org.uk
5. Marginean, A., Barr, E.T., Harman, M., Jia, Y.: Automated transplantation of call graph and layout features into Kate. In: Labiche, Y., Barros, M. (eds.) SSBSE. LNCS, vol. 9275, pp. 262–268 (2015), http://dx.doi.org/10.1007/978-3-319-22183-0_21
6. Marginean, A.: Automated Software Transplantation. Ph.D. thesis, University College London (2021), https://discovery.ucl.ac.uk/id/eprint/10137954/1/Marginean_10137954_thesis_redacted.pdf
7. Langdon, W.B., Harman, M.: Optimising existing software with genetic programming. IEEE TEVC 19(1), 118–135 (2015), http://dx.doi.org/10.1109/TEVC.2013.2281544
8. Weimer, W., Nguyen, T., Le Goues, C., Forrest, S.: Automatically finding patches using genetic programming. In: Fickas, S. (ed.) ICSE. pp. 364–374 (2009), http://dx.doi.org/10.1109/ICSE.2009.5070536
9. Weimer, W., Forrest, S., Le Goues, C., Nguyen, T.: Automatic program repair with evolutionary computation. Communications of the ACM 53(5), 109–116 (2010), http://dx.doi.org/10.1145/1735223.1735249
10. Haraldsson, S.O., Woodward, J.R., Brownlee, A.E.I., Siggeirsdottir, K.: Fixing bugs in your sleep: How genetic improvement became an overnight success. In: Petke, J., White, D.R., Langdon, W.B., Weimer, W. (eds.) GI-2017. pp. 1513–1520 (2017), http://dx.doi.org/10.1145/3067695.3082517
11. Le Goues, C., Pradel, M., Roychoudhury, A.: Automated program repair. Communications of the ACM 62(12), 56–65 (2019), http://dx.doi.org/10.1145/3318162
12. Monperrus, M.: Automatic software repair: A bibliography. ACM Computing Surveys 51(1), article no 17 (2018), http://dx.doi.org/10.1145/3105906
13. Alshahwan, N.: Industrial experience of genetic improvement in Facebook. In: Petke, J., Shin Hwei Tan, Langdon, W.B., Weimer, W. (eds.) GI-2019, ICSE workshops proceedings. p. 1 (2019), http://dx.doi.org/10.1109/GI.2019.00010
14. Harman, M.: Scaling genetic improvement and automated program repair. In: Kechagia, M., Shin Hwei Tan, Mechtaev, S., Tan, L. (eds.) International Workshop on Automated Program Repair (APR'22) (2022), http://dx.doi.org/10.1145/3524459.3527353

15. Kirbas, S., et al.: On the introduction of automatic program repair in Bloomberg. IEEE Software 38(4), 43–51 (2021), http://dx.doi.org/10.1109/MS.2021.3071086
16. Kechagia, M., Shin Hwei Tan, Mechtaev, S., Tan, L. (eds.): 2022 IEEE/ACM International Workshop on Automated Program Repair (APR) (2022), https://ieeexplore.ieee.org/xpl/conhome/9474454/proceeding
17. Callan, J., Krauss, O., Petke, J., Sarro, F.: How do Android developers improve non-functional properties of software? Empr. Soft. Eng. 27, Article 113 (2022), http://dx.doi.org/10.1007/s10664-022-10137-2
18. Langdon, W.B., Harman, M.: Genetically improved CUDA C++ software. In: Nicolau, M., et al. (eds.) EuroGP. LNCS, vol. 8599, pp. 87–99 (2014), http://dx.doi.org/10.1007/978-3-662-44303-3_8
19. Langdon, W.B., Modat, M., Petke, J., Harman, M.: Improving 3D medical image registration CUDA software with genetic programming. In: Igel, C., et al. (eds.) GECCO. pp. 951–958 (2014), http://dx.doi.org/10.1145/2576768.2598244
20. Langdon, W.B., Harman, M.: Grow and graft a better CUDA pknotsRG for RNA pseudoknot free energy calculation. In: Langdon, W.B., Petke, J., White, D.R. (eds.) GI. pp. 805–810 (2015), http://dx.doi.org/10.1145/2739482.2768418
21. Langdon, W.B., et al.: Genetic improvement of GPU software. GP&EM 18(1), 5–44 (2017), http://dx.doi.org/10.1007/s10710-016-9273-9
22. Klus, P., et al.: BarraCUDA - a fast short read sequence aligner using graphics processing units. BMC Research Notes 5(27) (2012), http://dx.doi.org/10.1186/1756-0500-5-27
23. Langdon, W.B., Brian Yee Hong Lam: Genetically improved BarraCUDA. BioData Mining 20(28) (2017), http://dx.doi.org/10.1186/s13040-017-0149-1
24. Langdon, W.B., Lorenz, R.: Evolving AVX512 parallel C code using GP. In: Sekanina, L., Ting Hu, Lourenco, N. (eds.) EuroGP. LNCS, vol. 11451, pp. 245–261 (2019), http://dx.doi.org/10.1007/978-3-030-16670-0_16
25. Lorenz, R., Bernhart, S.H., Höner zu Siederdissen, C., Tafer, H., Flamm, C., Stadler, P.F., Hofacker, I.L.: ViennaRNA package 2.0. Algorithms for Molecular Biology 6(1) (2011), http://dx.doi.org/10.1186/1748-7188-6-26
26. Andrews, R.J., et al.: A map of the SARS-CoV-2 RNA structurome. NAR Genomics and Bioinformatics 3(2), lqab043 (2021), http://dx.doi.org/10.1093/nargab/lqab043
27. Lewis, T.E., Magoulas, G.D.: TMBL kernels for CUDA GPUs compile faster using PTX. In: Harding, S., et al. (eds.) GECCO. pp. 455–462 (2011), http://dx.doi.org/10.1145/2001858.2002033
28. Liou, J.Y., Forrest, S., Carole-Jean Wu: Genetic improvement of GPU code. In: Petke, J., Shin Hwei Tan, Langdon, W.B., Weimer, W. (eds.) GI-2019, ICSE workshops proceedings. pp. 20–27 (2019), http://dx.doi.org/10.1109/GI.2019.00014
29. Liou, J.Y., Xiaodong Wang, Forrest, S., Wu, C.J.: GEVO: GPU code optimization using evolutionary computation. ACM Transactions on Architecture and Code Optimization 17(4), Article 33 (2020), http://dx.doi.org/10.1145/3418055
30. Liou, J.Y., et al.: Understanding the power of evolutionary computation for GPU code optimization. arXiv (2022), http://dx.doi.org/10.48550/ARXIV.2208.12350
31. Ryan, C., Collins, J.J., O'Neill, M.: Grammatical evolution: Evolving programs for an arbitrary language. In: Banzhaf, W., Poli, R., Schoenauer, M., Fogarty, T.C. (eds.) Proceedings of the First European Workshop on Genetic Programming. LNCS, vol. 1391, pp. 83–96 (1998), http://dx.doi.org/10.1007/BFb0055930

32. Shuyue Stella Li, et al.: Genetic improvement in the Shackleton framework for optimizing LLVM pass sequences. In: Bruce, B.R., et al. (eds.) GECCO. pp. 1938–1939. Association for Computing Machinery (2022), http://dx.doi.org/10.1145/3520304.3534000
33. Peeler, H., et al.: Optimizing LLVM pass sequences with Shackleton: A linear genetic programming framework. In: Trautmann, H., et al. (eds.) GECCO Comp. pp. 578–581. GECCO '22, Association for Computing Machinery (2022), http://dx.doi.org/10.1145/3520304.3528945
34. Peeler, H., et al.: Optimizing LLVM pass sequences with Shackleton: A linear genetic programming framework. Arxiv (2022), https://arxiv.org/abs/2201.13305
35. Brameier, M., Banzhaf, W.: Linear Genetic Programming. No. XVI in Genetic and Evolutionary Computation (2007), http://dx.doi.org/10.1007/978-0-387-31030-5
36. Petke, J., et al.: A survey of genetic improvement search spaces. In: Alexander, B., Haraldsson, S.O., Wagner, M., Woodward, J.R. (eds.) GECCO. pp. 1715–1721 (2019), http://dx.doi.org/10.1145/3319619.3326870
37. Rainford, P., Porter, B.: Using phylogenetic analysis to enhance genetic improvement. In: Rahat, A., et al. (eds.) GECCO. pp. 849–857. GECCO '22, Association for Computing Machinery (2022), http://dx.doi.org/10.1145/3512290.3528789
38. Langdon, W.B.: Genetic improvement of genetic programming. In: Brownlee, A.S., Haraldsson, S.O., Petke, J., Woodward, J.R. (eds.) GI @ CEC 2020 Special Session (2020), http://dx.doi.org/10.1109/CEC48606.2020.9185771
39. Papadakis, M., Yue Jia, Harman, M., Le Traon, Y.: Trivial compiler equivalence: A large scale empirical study of a simple fast and effective equivalent mutant detection technique. In: ICSE (2015), http://pages.cs.aueb.gr/~mpapad/papers/ICSE15B.pdf
40. Langdon, W.B., Petke, J., Bruce, B.R.: Optimising quantisation noise in energy measurement. In: Handl, J., et al. (eds.) PPSN. LNCS, vol. 9921, pp. 249–259 (2016), http://dx.doi.org/10.1007/978-3-319-45823-6_23
41. Blot, A., Petke, J.: Using genetic improvement to optimise optimisation algorithm implementations. In: Hadj-Hamou, K. (ed.) ROADEF'2022. INSA Lyon (2022), http://www.cs.ucl.ac.uk/staff/a.blot/files/blot_roadef_2022.pdf
42. Harman, M., Yue Jia, Langdon, W.B.: A manifesto for higher order mutation testing. In: du Bousquet, L., Bradbury, J., Fraser, G. (eds.) Mutation 2010. pp. 80–89 (2010), http://dx.doi.org/10.1109/ICSTW.2010.13
43. Sitthi-amorn, P., Modly, N., Weimer, W., Lawrence, J.: Genetic programming for shader simplification. ACM Transactions on Graphics 30(6), article:152 (2011), http://dx.doi.org/10.1145/2070781.2024186
44. Langdon, W.B., Veerapen, N., Ochoa, G.: Visualising the search landscape of the Triangle program. In: Castelli, M., McDermott, J., Sekanina, L. (eds.) EuroGP 2017. LNCS, vol. 10196, pp. 96–113 (2017), http://dx.doi.org/10.1007/978-3-319-55696-3_7