Optimizing Existing Software with Genetic Programming

William B. Langdon and Mark Harman

***Abstract*—We show that the genetic improvement of programs (GIP) can scale by evolving increased performance in a widely-used and highly complex 50 000 line system. Genetic improvement of software for multiple objective exploration (GISMOE) found code that is 70 times faster (on average) and yet is at least as good functionally. Indeed, it even gives a small semantic gain.**

***Index Terms*—Automatic software reengineering, Bowtie2^{GP}, genetic programming (GP), multiple objective exploration, search based software engineering (SBSE).**

I. Introduction

GENETIC improvement [1]–[4] is the process of automatically improving a system's behavior using genetic programming. Starting from a human written system, genetic improvement tries to evolve it so that it is better with respect to given criteria. The criteria for improvement are typically non-functional properties of the system, such as execution time and power consumption, though many others are possible [1], [4]. The functional properties of the evolved system are usually required to mimic as faithfully as possible those of the original system. However, we show that it may also be possible to improve the program's outputs.

In order to check that the original system's semantics are not disturbed, the genetic improvement process relies on a set of test cases, obtained from running the original system. Notice we can always do this [5]. Even where the existing system lacks a formal specification, its existing behavior is its own de facto specification. The answer given by the new code can be compared with that given by the original code (which is assumed to be correct). Thus, the original code is a test oracle. The system may also have additional automated oracles, which are able to check an output's validity and/or quality. These can also be used to test the functional behavior of the genetically improved program (see Fig. 1).

Genetic improvement has many potential applications. An existing program can be ported from one platform and language to another [2], thereby helping to manage software multiplicity [6]. Genetic improvement also allows programs to be automatically sped up [4] or consume less power, while still performing the useful functions offered by the original.

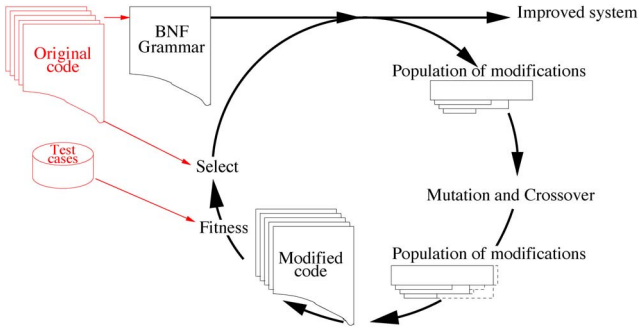

Fig. 1. Major components of GISMOE approach. (left) System to be improved and its test suite. (right) Genetic programming optimizes modifications, which originate from a grammar that describes the original system line by line. Each generation mutation and crossover create new modifications. Each modification's fitness is evaluated by applying it to the grammar and then reversing the grammar to get a new variant of the system. Each modified system is tested on a randomized subset of the test suite and its answers and resource consumption compared to that of the original system. Modifications responsible for better systems procreate into the next generation.

The goal of genetic improvement research is to automate as much of the improvement process as possible. Thus, new implementations can be discovered by an evolutionary process, rather than being hand-crafted by human programmers, in the currently familiar (yet time-consuming, tedious, and expensive) method. Ultimately, genetic improvement looks forward to a world in which our successors regard human programmers as a quaint anachronism of the past in much the same way that we now regard the human computers of our nineteenth and twentieth century forbearers.

Genetic programming provides a way to automate one of the most expensive and time-consuming aspects of the software engineering process: the production of the code itself. However, achieving genetic improvement for real world programs presents many challenges. The size and complexity of the programs to be evolved are considerably more demanding than those previously attempted.

We report the results of applying genetic improvement to a real-world system. Our genetic improvement of software for multiple objective exploration (GISMOE) approach reduces the search space for genetic improvement and manages the scalability of testing for functional and non-functional properties. We report the results of applying genetic improvement to Bowtie2 [7], a widely-used DNA sequencing

Manuscript received October 31, 2012; revised March 3, 2013 and June 4, 2013; accepted June 10, 2013. Date of publication February 6, 2014; date of current version January 28, 2015. This work was supported by the Engineering and Physical Sciences Research Council Grant EP/I033688/1.

The authors are with the Department of Computer Science, University College, London WC1E 6BT, U.K. (email: W. Langdon@cs.ucl.ac.uk; mark.harman@ucl.ac.uk).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TEVC.2013.2281544

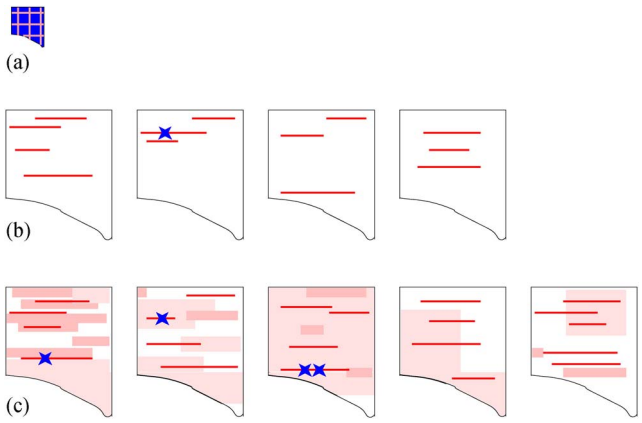

Fig. 2. (a) Initial approaches considered whole program equally (shaded). They update code (dark shading), which may be throughout the single source file. (b) Bug fixing. Genetic programming is directed to parts of code needing fixing (shaded) and the bugfix (star) is small. (c) GISMOE: Evolution is directed to used and heavily used code (light shaded, shaded, heavily shaded) several lines of code may be updated (stars).

system, consisting of 50 000 lines of C++ code for which we evolved 20 000 Line of code (LoC) (excluding headers and conditionally compiled debug code). In fact, by also excluding code that is not executed we focus the search on 2744 LoC. We used test cases from the 1000 genomes project [8]. In this case, the test cases are backed up by the Smith-Waterman score (as an automated test oracle, see Section IV-A).

Our primary finding is that genetic improvement can find new evolved versions of Bowtie2 that are, on an average, 70 times faster than the original (and produce on average slightly improved answers) when applied to DNA sequences from the 1000 genomes project. This is an important finding because genetic improvement (as opposed to automatic bugfixing) has previously only been applied to laboratory programs (of up to about 100 Lines of code). This previous work demonstrated proof of concept, but not the practical scalability required for realistic program improvements on real-world systems containing many thousands of LoC. Bridging this divide entails catering to all of the complexity and scale of real world systems.

Genetic programming (GP) has been used to fix bugs in real world programs of a similar scale to Bowtie2. However, this is the first improvement that has been applied to a real-world system. Though both genetic improvement and bug fixing have used GP as an underlying technique, the two applications of GP are different and pose different technical challenges as a result.

The difference in previous approaches to GP for software engineering is illustrated in Fig. 2, which consists of three lines. In each line, the icons denote the files that comprise a program or system. The first line depicts a program consisting of a single file containing a single procedure. The second and third lines depict entire systems (comprized of several files, each of which may have many procedures and functions).

Previous work on genetic improvement [2], [4] is depicted in the first line. This applies genetic operators to the entire program to improve it with respect to non-functional properties while maintaining [2] (or gracefully reducing [4]) functional properties. Initial foundational proof-of-concept work on GP for bug fixing [5] also applied genetic operators to small laboratory programs and so this initial work is also depicted in the first line of Fig. 2.

Subsequent work on bug fixing [9] extended this initial work to whole systems, using fault localization techniques to identify the parts of the system that might require changing. This demonstration of scalability of bug fixing is depicted in line two of Fig. 2; though the whole system is executed, the GP search is concentrated on only that small part to which the bug is localized. This localization is depicted by the horizontal shaded lines. Only a specific location (depicted by the star) is actually modified by the genetic operators to fix the bug.

Here, we extend genetic improvement [2], [4] from proof-of-concept to real-world applicability. In order to do this we apply GP to multiple points in a system (of multiple files), guided by a sensitivity analysis that identifies parts of the system that are most relevant to the non-functional property of interest. (In our case, the most frequently executed code.) This modus of operation is depicted in the last line of Fig. 2; the whole system is executed and the GP search is directed to multiple parts of the system (shaded). Although multiple parts of the system may be modified by GP to improve the performance of the overall system, the final number of lines changed may be modest (stars).

The primary contribution of this paper is to demonstrate that genetic improvement, previously only applied to laboratory programs, can scale to real-world systems of tens of thousands of LoC. We show that genetic improvement can produce dramatically faster versions of the program, for well-defined and useful subsets of the input domain and without loss of semantics. (Indeed, even with some modest improvement in semantics). In order to achieve this overall goal we introduce a number of techniques and approaches that may prove to be useful contributions for the future development of genetic improvement.

1) **Semantic Improvement**: We show that the presence of an automated test oracle opens up the possibility that genetic improvement might improve not only non-functional properties, but also a system's behavior (e.g., its accuracy), rather than merely seeking to maintain faithful semantics.
2) **Sensitivity Analysis**: We introduce a pre-analysis phase that tests the sensitivity of the program to the non-functional property we seek to improve (in this case execution time). As expected, improvements are most often found in the identified resource hungry code. We show how our grammar-based GP approach suits this sensitivity analysis, because it can identify the parts of the system to be evolved and those that are to remain untouched. This reduces the search space that genetic improvement has to consider.
3) **Output Bins**: We introduce an approach that caters to disparity between test cases by binning test cases according to the amount of output they produce. This allows a more uniform sampling, rather than merely sampling over the happenstance of test data availability.

Our output–bin approach is also used to assess the algorithmic complexity of the non-functional properties as they are empirically observed at each line of the program (see Section III-A).

4) **Operator Choice**: We demonstrate that our simple genetic operators, extending those used in automated bug fixing work, can also apply to genetic improvement.
5) **Grammatical Representation**: We introduce an adapted form of our grammar-based representation [10] to help guide the GP search.
6) **Local Search**: We show how a local search post processing phase can be used to address the potential (observed widely in many GP applications) for the solutions to become bloated. The local optimization is sufficient to reduce the modification required to a surprisingly small (and thus manageable) set of cut–and–paste operations (see Section III-F).

Section II presents an overview of the real-world system, Bowtie2, to which we apply genetic improvement, motivating our selection of this system. Section III outlines the GISMOE approach we use for genetic improvement, which is applied to Bowtie2 in Section IV using DNA data from the 1000 genomes project as test cases and the Smith-Waterman algorithm as an automated test oracle. The improvements we find using our approach are described (and investigated on held-out data sets) in Section V. Section VI describes related work and the relationship of our contributions to it. Section VII considers where else our approach might be successfully applied, while Section VIII concludes the paper.

II. Real World System Genetically Improved: Bowtie2

The exponential growth in DNA sequence data and the ever-changing analysis requirements for computer systems that operate on this data have led to many systems being created for DNA matching and analysis. Naturally, since genetic improvement techniques are in their infancy, these systems have been entirely hand-coded. In October 2012, Wikipedia alone listed more than 140 bioinformatics tools that perform some aspect of sequence analysis either on protein databases or DNA sequences. The production of so many tools requires a large amount of human effort, making this a natural target application domain upon which to evaluate an automated genetic improvement approach.

One of the most popular tools for querying next generation DNA sequences is the Bowtie system[1] Bowtie is very fast. However, its speed comes at the cost of some loss of functionality. Although derived from the Bowtie system, the Bowtie2 system [7] was written over 50 main system modules and 67 header files (plus documentation, scripts and support modules). These were downloaded from sourceforge. Although the system comprises more than one hundred source files, the final modification (see Fig. 15) changes only three of them. The Bowtie2 development effort is an attempt to emulate Bowtie, while retaining the speed of the original Bowtie system. However, though Bowtie2 is much faster than BLAST, it is, nevertheless, slower than Bowtie.

The 1000 genomes project [8] uses Solexa and other scanners to generate vast numbers of DNA sequences, in order to map human genetic variation. These data are publicly available and can be obtained via FTP [2]. For experimenting with genetic improvement applied to real world programs it is important to have a realistic pool of test data.

The properties of Bowtie2 and the test data make this an ideal target for the application and evaluation of our approach. More specifically:

1) The code is available, supporting full replication by subsequent authors.
2) There are realistic test cases available.
3) Test cases come from a non-trivial application (the analysis of human genetic variation, particularly with regard to disease factors and medical applications) that generate much interest. They are therefore more likely to involve real-world challenges than the artificially constructed code examples used so far.
4) Bowtie2 is much larger (being at least two orders of magnitude larger) than previously studied systems in work on genetic improvement.
5) Bowtie2 it is not merely larger, but also more complex than any previously studied systems for which results are reported for genetic improvement. Its scale crosses complexity boundaries not previously encountered. It includes many software engineering and programming features that any practical genetic improvement approach would need to address, yet which have been left unaddressed in previous work. Such features include modularization (functions and procedures), distinctions between main and support code (libraries, test harnesses etc.), separation into files, use of complex data structures, file access, preprocessing and macro calls.

Previous work on genetic improvement has demonstrated the possibility of using genetic programming to improve a program's non-functional properties and this has been very important. However, it is insufficient on its own. The development of techniques that apply to programs like Bowtie2 provide evidence that genetic improvement can be applied to programs used in demanding, complex, real-world applications.

III. GISMOE Approach

This section outlines our GISMOE framework [1] and how its principle components are instantiated to achieve genetic improvement for the Bowtie2 System.

A. O–Bins: Output Bins for Test Cases

We use output bins (O–Bins) to partition the available test cases. Our motivation for this is that testing practitioners intuitively have a concept of the difficulty of a test case. In many cases this is related to the amount of output that the test case causes the program to create. That is, tests that cause the generation of a lot of output are, in some sense, more difficult than those that cause comparatively little output to

[1] As of Oct 2012, according to Google Scholar, Bowtie had been cited 1706 times.

[2] See, for example, ftp.1000genomes.ebi.ac.uk

be produced. However, testers might have other appropriate measures. These might be easily measurable (e.g., run time) or require the code to be instrumented (e.g., length or complexity of execution path, such as number of branches). Alternatively, the testers may have their own subjective way of partitioning test cases to give a spread of difficulty.

O–Bins play a role in the assessment of both functional and non-functional properties of the code. For the functional properties, we use F different O–bins. Test cases (n per bin) are sampled uniformly from these F O–bins (rather than uniformly over all available test cases). The binning process ensures that we sample demanding test cases for fitness evaluation as well as less demanding ones, even though we only sample $n \times F$ test cases for fitness at each new generation. For the assessment of the non-functional properties, we also use O–bins to ensure that tests are sampled uniformly over their perceived difficulty (rather than merely over their availability). This use of O–bins is explained in more detail in Sections III-C and III-D.

B. Determining Functional Correctness

The functional properties of a system are typically assessed by GP using a test-based approach. However, testing suffers from the oracle problem. That is, we need an automated oracle that will determine whether a given output observed is correct. Fortunately, one of the advantages of genetic improvement is that the original program can serve as an oracle. That is, it can be used as an automated system that provides a reasonable output for a given input. This has been the basis of previous approaches to both genetic improvement and bug fixing [1], [2], [4], [9].

However, using the original program as an oracle has its drawbacks. The original program may be buggy, in which case the improved program may merely faithfully replicate buggy behavior. The original may also be either partially defined or non-deterministic, in which case it will not provide a reliable oracle for every possible input.

It is therefore always advisable to supplement the original program with an automated oracle (or partial oracle) if one is available. The use of partially automated oracles (other than the original program) also brings with it additional advantages: the genetically improved program may improve the functional properties of the system as well as its non-functional properties. Our approach to functional faithfulness is therefore to use the original program as one source of oracle information, but to additionally seek other partial oracles in order to check the output produced by the genetically improved system.

C. Sensitivity Analysis for Non-Functional Properties

In order to evolve systems to better meet non-functional requirements, we first apply a form of sensitivity analysis to determine the parts of the system that have the greatest effect on the non-functional property of interest. The parts of the system with greatest impact will have the highest priority during GISMOE's evolutionary phase.

Depending upon the requirements, there are a number of techniques that can be used to measure non-functional properties of software. Some of these can be fairly direct. On a server there is usually accounting information (e.g., number of page faults or number of pages of RAM in use), which can be harnessed as part of a fitness function. Similarly, the operating system might keep track of bytes sent/received via a wireless port. In other cases, the accounting information may not be available or may be too inaccurate and so the experimenter may have to devise their own measures. It is not common to keep track of the power consumed by individual software components. However, White *et al.* [4] demonstrated how simulators can do this, and that they can be incorporated into a GP fitness measure.

In this paper, the non-functional property of concern is the execution time of the system. As might be expected, typically, which lines are used and how many times they are executed varies a great deal. We use execution frequency as an indication of those lines of code that are likely to have the strongest influence on our non-functional property of interest. We weight lines of code both in proportion to how much they are used and also how this use scales with the difficulty of the problem.

We use a non-linear weighting in order to try to ensure that GP samples critical parts of the system to be improved more heavily. The determination of what makes parts of systems critical depends both on the domain and upon the non-functional properties to be improved. Hence, it must be defined for each kind of system to be improved. If such domain knowledge is unavailable or there is no meaningful characterization of difficulty of the non-functional properties, then this aspect of our weighting scheme can simply be ignored. However, where there is domain information, it makes sense to ensure that it plays a role in the determination of weights.

We are interested in assessing the way in which the non-functional values observed vary with test case difficulty. The test cases are therefore partitioned into N O–bins. Conceptually, we plot the variation of the non-functional property (on the vertical axis) against the test case bin-number (ordered by output size on the horizontal axis). Using this plot we determine the algorithmic complexity of the non-functional property for each line of code.

We assess the algorithmic complexity of this conceptual plot allocating a score of 10 for any complexity up to linear, 100 for quadratic, and 1000 for cubic and higher complexity. We combine this algorithmic complexity measurement with a scalar measurement that is simply a measure of the number of times that the line is executed on average by a set of test cases, sampled uniformly from the O–Bins. The overall measurement of the sensitivity of a line of code is the maximum of the scalar and algorithmic complexity measurement obtained for the line. To prevent search concentrating overmuch, during a GP run each line of code that is mutated has its weight reset to one. Thus, encouraging GP to move on to also consider other lines.

D. Testing for Fitness According to both Functional and Non-Functional Properties

1) *Compilation:* To reduce compilation time, an instrumented version of the system is compiled without optimization, with the gcc `-Wfatal-errors` option and using

precompiled header files. Initially, compilation time will be light because there will be few changes, but this will increase as more files are touched and recompilations requires a larger build. For example, in our experiments with Bowtie2, compilation time was observed to grow by an order of magnitude during the genetic improvement process (from below a second at the start, up to about 10 s by the end).

2) *Randomized Test Suite Sub-Sampling:* We give each evolved version several tests. To make them independent and so prevent an error on an earlier test affecting later tests, each evolved version is run on each test case separately (see Table II).

There are many test cases available for most programs we might wish to genetically improve. We therefore adapt our sampling approach [2], [11]. That is, at each generation, we select a single test case from each of F O–bins to form a test suite for that generation. At each generation the set of test cases to be used is reselected to ensure diversity of testing (To avoid retesting, we do not use elitism).

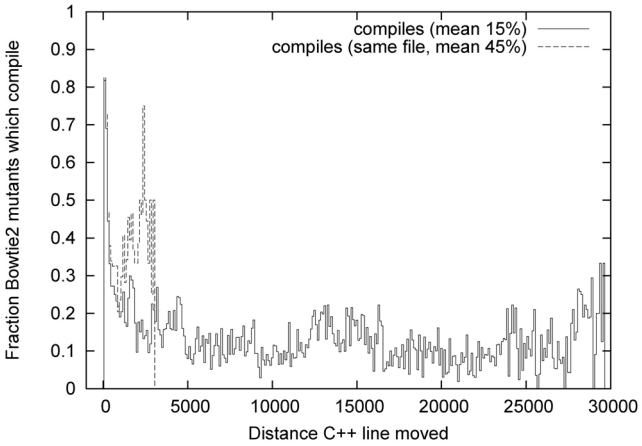

Fig. 3. Fraction of Bowtie2 variants which compile by distance replacement C++ line is moved. 82% of cases where code is moved $\leq$ 100 lines compile. When lines are only moved within the same source file (dashed line) on average three times as many variants compile since there are fewer out of scope errors. Data binned in units of 100 lines.

E. Handling Infinite Loops

Since we allow `for` and `while` loops to be changed by the genetic improvement process, it is quite possible that the modified code could enter an indefinite loop. We do not want non-termination of a genetically improved variant to lead to non-termination of our whole genetic improvement system. Several approaches have been suggested to handle this problem. For example, Maxwell [12] suggests a way to allow fitness comparison as programs run, while Teller [13] suggests using an anytime algorithm whereby answers, and hence fitness, can be extracted from an executing program, rather than waiting for it to terminate.

We use the operating system to time out and abort any evolved version that takes more than a predefined cut-off execution schedule. As the Bowtie2 documentation says "*Bowtie2 is not particularly designed with –all mode in mind, and when aligning reads to long, repetitive genomes this mode can be very, very slow.*" Hence, some test cases (especially those with more than 100 matches, see Section IV-B) need longer time outs than others. Pragmatically, we impose a CPU limit of twenty seconds on the first four test cases and one minute on the last test case.

F. Representation of the System to be Evolved

The existing program is used as the template for its own upgrade. The template (actually a special one-sentence BNF grammar) is created automatically from the program's source code. While evolution has great freedom to change the code, it is constrained by the template. For example, the template ensures classes, types, functions, and data structures are retained. Similarly, evolution cannot change the program's block structure. So, for example, in C++, opening and closing brackets have to remain in the same place but lines between them can be changed. Thus, for example, each function's name and arguments cannot be changed but their contents can be rewritten and indeed so too can the code that calls them. Similarly, variables retain their names and types but evolution can use them, change their use or indeed ignore them totally.

We use a specialized BNF grammar to ensure that the evolved code has no parse errors. However, GP can generate code with other language errors (e.g., referring to a nonexistent variable). These are trapped by the compiler, causing the modified code to fail at the compilation stage. Our experience is that almost all such compilation errors involve variables being out of scope. Earlier experimentation confirmed this to be the case.

Such scoping issues might be tackled by a detailed type analysis. However, increasing the fraction of shorter distance moves by restricting moves to be within the same source file has proved to be a simple and effective way of increasing the fraction of evolved versions that compile (see Fig. 3 and Section III-I).

Although we have not found them to be needed here, there are a number of sand boxing [10] and virtualization techniques to ensure C programs do not cause damage.

BNF rules that correspond to single lines of source code are modified so that they now invoke another rule with the same name but with a leading underscore inserted. (For example, for the Bowtie2 program, the original rule `<bowtie_main_46>` in Fig. 4 was modified so that it invokes the new rule `<_bowtie_main_46>`.) GP can replace this (the underscore rule) with another rule also starting with an underscore and the resulting program will be syntactically valid.

For example `<_bowtie_main_46> ("in. open (file);")` could be copied to replace `<_bowtie_main_51> ("args.push_back(string(argv[0]) );")`. This gives two calls to open `file` but now `args.push_back` is never called. The second call of `in.open` finds that stream `in` is already open and does nothing. The resultant code is syntactically valid and, in this case, compiles. In some test cases (e.g., where the first command line argument is not `"-A"`, see line 43, Fig. 4) the variant runs despite the missing call to `args.push_back` and generates identical output to the released code. Such test cases do not reach the site of the modified code and so it

```
<bowtie_main_42>        ::= "int main(int argc, const char **argv) {\n"
<bowtie_main_43>        ::= "{Log_count64++;/*29823*/} if" <IF_bowtie_main_43> " {\n"
#"if
<IF_bowtie_main_43>     ::= "(argc > 2 && strcmp(argv[1], \"-A\") == 0)"
<bowtie_main_44>        ::= "const char *file = argv[2];\n"
<bowtie_main_45>        ::= "ifstream in;\n"
<bowtie_main_46>        ::= "" <_bowtie_main_46> "{Log_count64++;/*29826*/}\n"
#other
<_bowtie_main_46>       ::= "in.open(file);"
<bowtie_main_47>        ::= "char buf[4096];\n"
<bowtie_main_48>        ::= "int lastret = -1;\n"
<bowtie_main_49>        ::= "while" <WHILE_bowtie_main_49> "  {\n"
#WHILE
<WHILE_bowtie_main_49> ::= "(in.getline(buf, 4095))"
<bowtie_main_50>        ::= "EList<string> args;\n"
<bowtie_main_51>        ::= "" <_bowtie_main_51> "{Log_count64++;/*29831*/}\n"
#other
<_bowtie_main_51>       ::= "args.push_back(string(argv[0]));"
<bowtie_main_52>        ::= "" <_bowtie_main_52> "{Log_count64++;/*29832*/}\n"
#other
<_bowtie_main_52>       ::= "tokenize(buf, \" \t\", args);"
```

Fig. 4. Fragment of BNF grammar used by GP. Most rules are fixed but `<IF_`, `<_`, `WHILE_` etc. can be manipulated using rules of the same type to produce variants of Bowtie2. `Log_count64++` etc. are automatically added to instrument Bowtie2. Lines beginning with # are comments.

cannot propagate its effects and so, in such cases, the variant is equivalent to the original code.

The conditional parts of `if`, `else`, and `while` as well as the initial, test, and increment parts of `for(;;)` loops are extracted into new rules (with rule names beginning `<IF_`, `<ELSE_`, `<WHILE_`, `<for1_`, `<for2_`, and `<for3_`) (see examples in Fig. 4). GP is free to exchange these with other rules of the same type, to generate a syntactically valid program.

We limit GP to evolving code in the main modules. That is, evolution cannot modify the include files. As in our previous work [10], we used the gcc compiler's `-E` option to strip comments, to ensure compile time configuration, (using the release configuration), and to perform macro expansion on the source code.

Human written code is highly repetitive; whole source lines of code occur more than once. For example, Gabel and Su [14] recently found that almost all small code fragments have been written before, somewhere by someone (i.e., not necessarily in the same application). Previous studies, see, [15], have reported Zipf's law in programmers' use of language tokens, e.g., `()` and `if` in Java, which are enforced by the compiler.

Excluding white space, Fig. 5 plots the number of times lines of C++ code in Bowtie2 that are exactly repeated. It is no surprise to discover lines composed of a single } or a single ; occur many times (actually 2310 and 1255 times). But many more interesting lines are also repeated. For example, the eighth most commonly repeated line is a non-trivial line of 56 characters including branches, variables, and constants. This is longer than most of the 5848 (29%) lines that are unique (their median length is 28 characters).

While Gabel and Su investigated code repetition across an entire suite of programs and systems, we present in Fig. 5, results for code repetition within one single C++ system, Bowtie2. We suspect that the results we observed and those reported for much larger corpuses [14] reflect a wider trend.

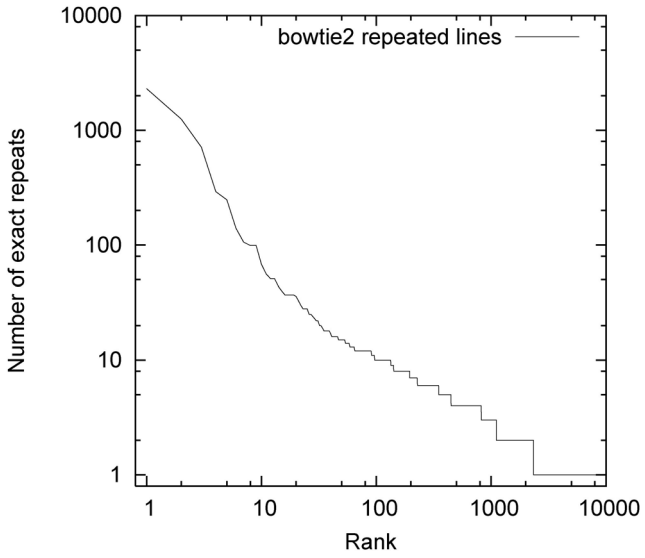

Fig. 5. Distribution of repeated Bowtie2 C++ code, after macro expansion, follows approximately Zipf's law, which predicts a straight line with slope of -1.

It is also well known that crossover can produce large amounts of repeats, both in natural DNA and in linear and tree genetic programming.

Genetic improvement should take these observations about code repetition into account. Therefore, instead of allowing GP complete freedom to invent any syntactically valid code, we insist it reuse code that has already been written by the

creator of the program to be genetically improved. Evolution thus proceeds by cut and paste. Cutting, i.e., removing lines of code, and pasting means to make a copy of a line of code in another place.

We were surprised that such a simple approach to modification could yield dramatic genetic improvements (but see also [9]). We believe that this is a potentially important finding of our work.

Our approach is similar to the plastic surgery approach of Weimer *et al.* [9] in which code is scavenged from other parts of the program under evolution. However, while Weimer *et al.* consider code at the statement level, we will deal with lines of C++ code. The creation of a grammar describing the existing source code (Fig. 4 contains an example grammar) identifies seven different types of source code fragment (see Section III-F). As long as we only cut and paste source code fragments of the same type, the new code will be syntactically correct at this lexical level (there are some examples in the next section).

G. Representation of Genetically Improved Variant

In earlier work on genetic improvement [2], [4], the entire program was evolved. This was feasible because there was only a small program [4] or part of a program [2] to be evolved. However, in order for genetic improvement to scale it must cater for programs of several orders of magnitude larger than have previously been considered. We therefore adapt an approach recently used to scale up bug fixing [9]. We represent a GP individual as an ordered list of changes [16] that are to be made to the BNF grammar. To delete a line of code, the GP individual gives the name of the line's BNF rule. To replace a line, the name of the corresponding BNF rule is given together with the name of the line of code which is to replace it. An insert operation is essentially the same, except we add + to the text, so we know to add a copy of the line of code and not to remove the original line of code.

Here are some examples of the application of this approach to the Bowtie2 system:

```
<for3_sa_rescomb_111><for3_sa_rescomb_69>
```

This GP individual causes the increment part of the `for` loop on line 111 of source file `sa_rescomb.cpp` to be replaced by the increment part from the for loop on line 69.

```
<_aligner_swsse_ee_u8_804>
```

This individual causes line 804 of `aligner_swsse_ee_u8.cpp` to be deleted.

```
<_aligner_result_47>+<_aligner_result_114>
```

This individual inserts a copy of line 114 in front of line 47 in file `aligner_result.cpp`.

The first generation is the initial random population. In it all GP individuals contain exactly one change. In the second generation we start to see individuals that make two or more changes. These individuals are simply one line of text with a space between each of their constituent mutations. Mutations are applied in order. However, we can readily spot mutations which replace the same line of code. In this case only the last one need be applied. In fact, we use genetic repair, so where conflict arises an individual's genome only contains the relevant, i.e., the last, mutation. Notice we can easily keep track of which source files have been changed and use a Unix make file to ensure only the modified files are recompiled.

All this manipulation is done in plain text, unlike other work, based on CIL, which operates on abstract syntax trees (AST) [9]. Our grammatical representation of the program to be improved makes this practical, even though it operates at a lexical level. Even for the largest set of genetic changes and even prior to the final local search bloat-removal phase, the time required to make the changes is typically less than that required to compile the resulting modified system.

H. Selection

Up to half the current population can be selected. Those below the cut point, as well as variants that failed to compile or that never exceeded the released code in any way, are not transmitted to be parents of the next generation. As will be mentioned in Section III-J, if fewer than half the population are selected, two new children per missing member are created from scratch. That is, they are effectively reinitialized. The new individuals are created in the same way as the initial population was created in generation zero.

I. Mutation

An individual is mutated by appending a new grammar modification to the list that denotes an individual (see Section III-G). The additional line to mutate is chosen from all lines executed at least once by the test cases selected from the O–Bins for sensitivity analysis. The line to be mutated is chosen with a probability that is defined by its sensitivity analysis weight.

One of the three types of mutation (deletion, replacement, and insertion) is chosen (with equal probability). Note that these are the only permitted changes, in particular, totally new code cannot be introduced. However, it makes no sense to delete one of the trivial lines (i.e., lines just containing a single disabled `assert` or `;`). Trivial lines can, with equal probability, be used either as the point to insert new code or be replaced with a non-trivial source line. The new code is chosen uniformly at random from non-trivial lines of the same type (captured by our grammatical representation approach) in the same source file that were executed at least once (the types were described in Section III-F above).

We avoid the generation of no operation and duplicate code. That is, a child is rejected if either it makes no change (i.e., replacing self with self) or where the corresponding sequence of changes already exists (both can be spotted efficiently since changes are essentially cut-and-paste operations). If a child is rejected in this way, then the parent is mutated again until a non-duplicate is created.

J. Crossover

We anticipate that many changes are somewhat independent; a genetically improved program can be found by combining multiple changes. This is the role of crossover. In our case,

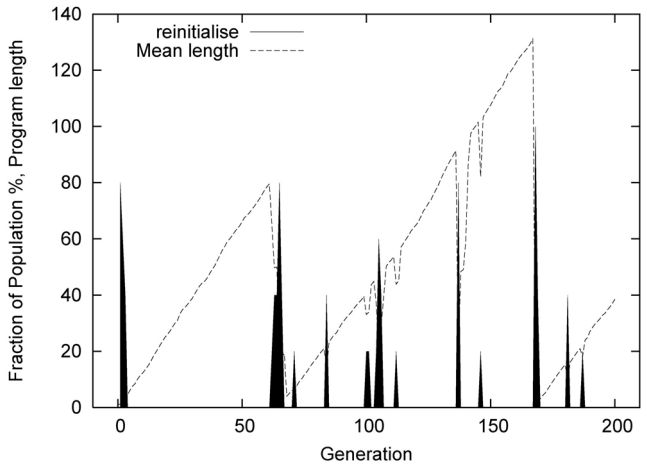

Fig. 6. Increase in mean number of mutations as evolution improves Bowtie2. Note many members of population (10) reinitialized near generations 65 and 167, causing long mutation lists to be replaced by new (much shorter) individuals.

crossover simply concatenates two individuals. The first parent is selected from the current population according to its fitness. The second is drawn uniformly from the members of the current population which compiled (i.e., have a fitness value). Naturally, such a crossover leads to rapid growth in chromosome length (bloat [17]). See Fig. 6 for an example of the increase in genotype length, which we observed in our experiments with Bowtie2.

As with mutation (see the previous section) each child's genotype is reduced to canonical form and a crossover will be rejected if it is already present in either the new or the previous generation. If, after a small number of retries, crossover cannot find a unique individual, the new child is created by mutating a fit member of the population.

Normally, half the new population is created by mutating the fittest parents and half by crossing over the fittest parents with other fit members of the population (to ease reproduction all the key parameters of our evolutionary system are given in Table I). However, if the number of fit parents in the current population falls below half, mutation and crossover will not create sufficient new children to fill the new population. In this case, to restore diversity to the population, the missing children are created at random (in the same way as the initial random population).

K. Post Processing Solution Cleanup

It is common for solution programs evolved using genetic programming to be bloated [17]. That is, some parts of the evolved changes make little or no difference. From the point of view of software engineering, maintenance, ease of integration of genetic changes into human written code, etc., it is easier to work with a minimal number of changes. It is possible for genetic programming to minimize evolved code, but previously we had used larger populations. Therefore, we decided to use a simple hill climbing strategy to minimize the size of the ordered list of changes after the end of the GP run.

Starting from the beginning of the best individual in the last generation, each of the changes are disabled one at a time. If removing the change makes the evolved version worse, then the evolved change is kept. Otherwise, it is removed. The hill climber then goes on to test the effect of removing the next evolved change and so on, until the whole evolved version has been so-processed.

IV. Applying the GISMOE Genetic Improvement Approach to Bowtie2

In this section, we explain how the GISMOE genetic improvement approach we introduced in the previous section is applied to the Bowtie2 program.

A. Determining Functional Correctness for Bowtie2

For the automated oracle, we use the Smith-Waterman algorithm to compare the answer given by Bowtie2 with the human genome. Unlike Bowtie2 itself (and related tools), Smith-Waterman performs a complete comparison, rather than using heuristics. However, Smith-Waterman can only allow us to check the reported sequence matches for correctness, it does not allow us to check for missing answers. Smith-Waterman thus provides a partial oracle, that can evaluate the answers given.

In order to use the Smith-Waterman score, we need to allow for partial matches (indels). To do this, the reference string against which matches are checked is extended by nine characters at either end. The genetically improved system's Smith-Waterman score for a test case is the mean of the Smith-Waterman scores over all matches it suggests for that test case. However, if the output from the modified system is a match that fails to lie exactly where Smith-Waterman locates the optimum match, then the match's score is reduced by 1.0 for each DNA string position by which its output disagrees (subject to the match's score not going below zero).

Bowtie2 reports many accountancy details about the matches it finds between the test Solexa DNA sequence and the human genome. We give credit only for the matches themselves. Potentially, evolution can make minor saving by mutating Bowtie2 so that it no longer generates this unwanted output.

1) *Training Data—Human Genome, Bowtie2 and the 1000 Genomes Project:* The complete official release of the reference human genome (Release 37, Patch 5) was downloaded from the National Center for Biotechnology Information (NCBI). The NCBI also maintains BLAST. The 64 bit Linux version of Blast was downloaded from its FTP site (version 2.2.25+) and this version was used in the experimental comparisons reported below. The C implementation of the Smith-Waterman local alignment algorithm was downloaded from Biological Physics Department, Cologne University. The Smith-Waterman algorithm does a complete search to find the optimum match between two strings.

The C++ sources for the 64 bit Linux version of Bowtie2 (version 2.0.0-beta2) were downloaded from sourceforge (50 745 lines). This version of Bowtie2 was used to create an ASM format database holding the reference human genome from the NCBI DNA sequences. We have evolved all new versions of Bowtie2 by fitness testing against the complete human genome (3.9 GB). Fitness testing might be sped up by

TABLE I
GENETIC PROGRAMMING PARAMETERS(INCLUDING, FOR REPLICATION PURPOSES, THE SPECIFIC PARAMETERS USED FOR IMPROVING BOWTIE2 ON 1000 GENOMES PROJECT SOLEXA SHORT DNA SEQUENCES)

Representation:	List of replacements, deletions and insertions into BNF grammar
Fitness:	Based on compiling modified code and testing it. See Sections III-D and III-H and Table II
Selection:	An evolved version cannot be selected to be a parent unless it does better on at least one test case than the original (instrumented) code. A new individual is said to better if the unmodified code failed on the test case and it doesn't or its mean Smith-Waterman score is higher than that of Bowtie2 or if it used fewer statements. However, normally it must also report at least one match. Scores are sorted by number of test cases where the evolved version returned an answer, mean Smith-Waterman, and finally by number of statements executed. The first five are selected to have 2 children in the next generation. One child is a mutant, the second is a crossover between a selected parent and another member of the current population. Children must be different from each other and from the current population. If crossover can not create a different child, the child is created by mutating the selected parent.
Population:	Panmictic, non-elitist, generational. 10 members. New training sample each generation.
Parameters:	Initial population of random single mutants weighted towards heavily used statements. 50% append crossover. The 3 types mutation (delete, replace, insert) are equally likely. No size limit. Stop after 200 generations.

using only part of the database or indeed a smaller genome from a non-human source (e.g., yeast or mycoplasma bacteria). However, our goal was to tailor Bowtie2 to the task of looking up human DNA sequences. Indeed, we wished to create a version specific to real DNA sequences generated by a particular sequencing technique, rather than synthetic data.

The 1000 genomes project [8] has sequenced, wholly or in part, DNA from more than one thousand individuals using a variety of next generation sequencers. Our goal is to show the automatic generation of improved software for a particular task, so we use data from a popular scanner make used in a laboratory for which we have copious training data. The data can be obtained via FTP from ftp.1000genomes.ebi.ac.uk. We selected homogeneous data (i.e., DNA sequences with exactly 36 bases) from one well studied CEU family and the Solexa data provided by the Broad Institute, Cambridge, MA, USA.

The Solexa scanner output includes an estimate of the quality of each base in the sequence. It also uses "N" to indicate any DNA base which it cannot decide which of the four bases (A, C, G, T) it really is. The data quality is highly variable. In one dataset, less than 1 in a 1000 sequences has an N. In the worst training dataset, every record had at least one (typically two or three).

2) *Preparing the Training Data:* The performance of Bowtie2 depends strongly upon the number of matches it finds between the query DNA sequence and the reference human genome. To get a good spread of Solexa DNA sequences for training, we started by using the released version of Bowtie2 to annotate a sample of DNA sequences with the number of times they occur in the human genome. We randomly selected 500 of the $\approx$8 million DNA sequences in each of 11 Solexa runs for a CEU female (NA12878). Then we ran the released version of Bowtie2 against the human genome on groups of 50 randomly chosen sequences and for each counted the number of matches it reported. (Total $11 \times 10 \times 50 = 5500$.) Even on a 32GB 8-core server, in five cases, Bowtie2 was aborted (after failing to respond), leaving us with 5250 DNA Solexa sequences. The distribution of the number of matches is plotted in Fig. 7.

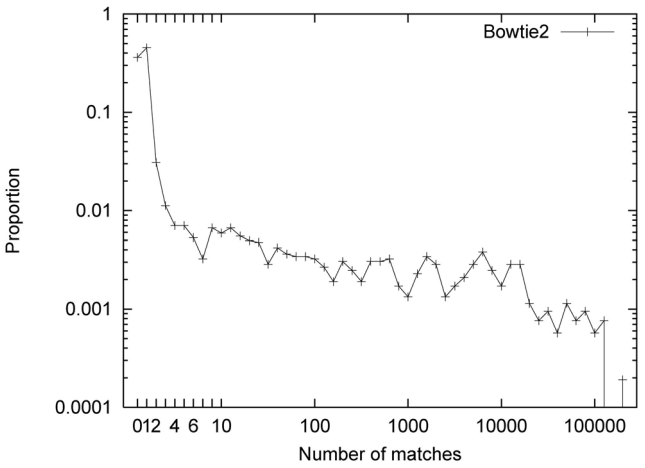

Fig. 7. Distribution of number of matches in human genome for Solexa DNA sequences found by Bowtie2. (Note non-linear scales. With more than ten matches decile bins are used.)

B. O–Bin Sampling as Applied to Bowtie2

We implemented the O-bin sampling described in Section III-A as follows. In each generation, five DNA sequences are automatically chosen uniformly at random from the 5250 Solexa sequences described in the previous section. They include the following:

1) A sequence which Bowtie2 cannot find in the human genome.
2) A sequence which it matches exactly once.
3) A sequence which matches between twice and ten times.
4) A sequence which matches between 11–99 times.
5) A sequence which matches between 100–200 times.

The first two cases can be viewed as positive tests to ensure the modified Bowtie2s still retain their essential ability to both report the absence of matches and find them. Test cases of type 3, 4 and 5 are designed to detect modified Bowtie2s that are faster. Cases 3 and 4 are intermediate. They seek to guard against chance playing too great a role in parenthood selection. Bowtie2 run time grows cubically with number of matches. Fig. 7 shows the number of matches (n) within the human genome that Bowtie2 can find is essentially unlimited. Given $O(n^3)$ run time, we cannot possibly use the sequences that match many times in fitness testing. Instead, we imposed an upper limit of 200 matches. Even so, the fifth test case (which is drawn from the O–Bin containing the most demanding test cases) frequently takes the most computational effort.

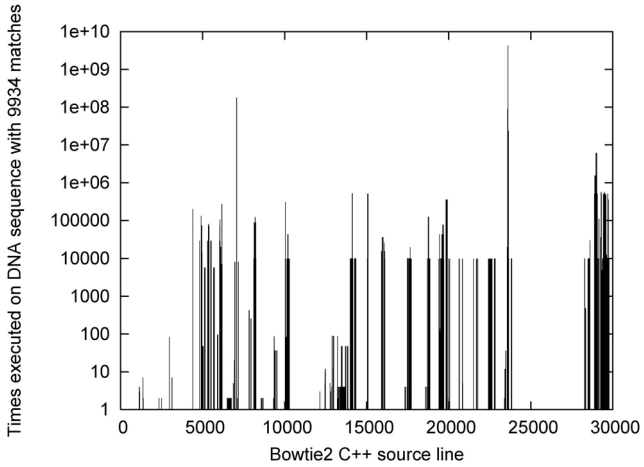

Fig. 8. Example when Bowtie2 finds many matches of the distribution of the number of times each Bowtie2 C++ source line is used. (All 39 source files but excluding all 67 header files.) 72% of lines are not used but 80 lines are run more than a million times. (Note log scale.)

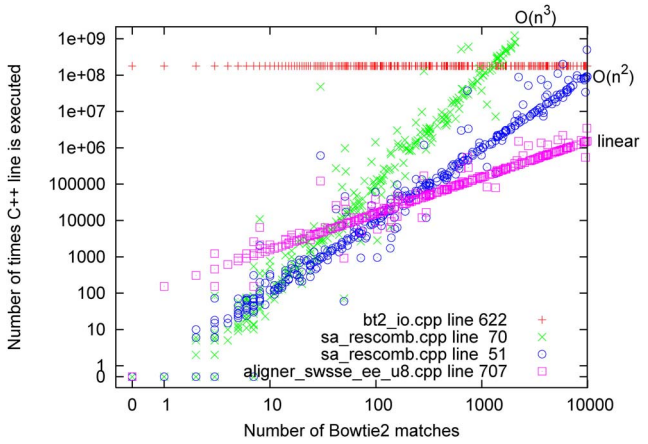

Fig. 9. Example heavily used lines in Bowtie2, which scale differently with number of matches found for the input Solexa DNA sequence in the human genome. Constant +, linear (□), quadratic (⊙), and cubic (×). (Note log scales.)

C. Representing Bowtie2 Source Code to be Evolved as Grammar

Following the general GISMOE approach outlined in Section III-F, we limit GP to evolving code in the main modules of Bowtie2. This yields 39 modules containing about 20 000 lines of code. These were automatically translated line for line into a BNF grammar of 19 949 rules. (See fragment in Fig. 4.)

D. Sensitivity Analysis for Non-Functional Properties Applied to Bowtie2

To illustrate the importance of our sensitivity analysis, consider Fig. 8, which shows the number of times individual lines of code were executed in an example run of Bowtie2. Of the 13 498 executable lines that were instrumented, 9 760 (72%) were never used, 1518 (11%) were used exactly once, 846 (6%) were used more than once but less than the number of matches (9934), 309 (2%) were used exactly 9934 times and 929 (7%) were used more than 9934 times. In fact, 80 lines were used more than a million times, with 2 being used more than 2 147 483 648 times.

Although Fig. 8 describes a single test case, the GISMOE weighting scheme (described in Section III-C) is based on a large number of test cases. In more detail, for Bowtie2, the 5250 DNA Solexa sequences, described in Section IV-A2, were sorted by number of matching strings into decile O–bins. That is, ten O–bins per order of magnitude. (Meaning ten O–bins for numbers between 10 and 100, ten for numbers between 100 to 1000, and so on.) That is, the O–bins are spread evenly on a logarithmic scale. If there were more than ten DNA sequences in an O–bin, ten were chosen at random from it. This yielded 362 DNA sequences with a wide variety of number of matches in the human genome.

An instrumented version of Bowtie2 was run on them all to give, for each of the 362 input test cases, which lines of code were used and how many times. Fig. 9 shows, for four example source lines, the relationship between the number of times the query string matches in the reference database and the number of times the source code is executed. The instrumented version allows us to not only know which lines of code are in use but also to estimate how their usage scales. We find: 1) lines that are never used; 2) lines that are used once (or a constant number of times); 3) lines whose use varies linearly with output size (n); 4) lines whose use varies as n^2, and 5) lines whose use varies in proportion to n^3. This gives us a crude assessment of the algorithmic complexity of each line of code.

For Bowtie2, the weights described in Section III-C are calculated as follows: if a C++ line is used in any of the 362 instrumented runs it will be given a weight between 1 and 1000 in proportion to the number of times it is used in that test. If it is not used at all, then GP ignores it and will not mutate it. If it is heavily used in any test, it is given the higher weight.

These weights are combined with the algorithmic complexity assessment (which allocates weights of 10, 100, and 1000 for O(n), O(n^2), and O(n^3) complexity, respectively, as described in Section III-C) by selecting the maximum of the two scoring systems. Fig. 10 (generation 0) shows the results of this non-functional sensitivity analysis as applied to Bowtie2. Fig. 10 shows 2111 lines have initial weight 1, 483 have initial weight 10, 103 have initial weight 100, and 47 have initial weight 723 or more (total 2744).

E. Combining Functional and Non-Functional Fitness to Create Overall Fitness for Bowtie2

Each genetically improved variant that compiles is compared to the instrumented original on the five test cases selected from the O–Bins (to be described in Section IV-F4, see also Table II). Two fitness criteria are used corresponding to the functional and non-functional criteria. Did the evolved version run faster (which, in our case, is measured in terms of the execution of fewer lines of code) and did it produce better answers on average (which is measured in a domain-specific

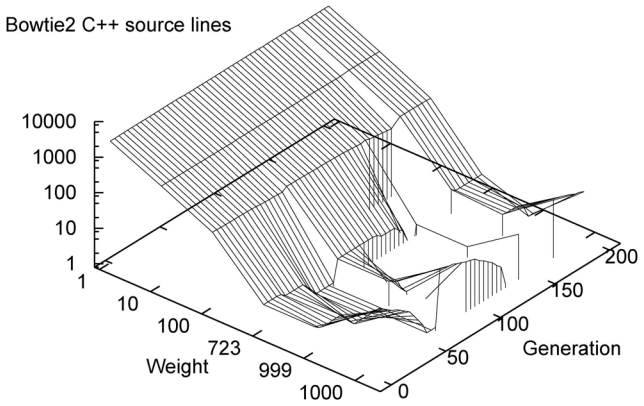

Fig. 10. Evolution of distribution of weights. Initial GP population (generation 0) at left. Note fall in proportion of higher weights as population evolves to mutate highly used C++ lines and then recovery as it (partially) resets in generations 68 and 168. (Note non-linear scales. To reduce clutter, data plotted only every 4^{th} generation.)

manner)? It need only do better on either criteria on any of the five test cases to be considered a parent of the next generation. (However, it is not considered better if it finds no answers at all, no matter how fast it goes.) Those variants that compiled and were judged better than the original code on the current five test cases are sorted according to three fitness criteria. The three fitness criteria (in order of precedence) are:

1) Number of test cases completed without run time error;
2) Mean Smith-Waterman score;
3) Lines of C++ code executed (minimized).

We order the criteria in this manner to favor functional faithfulness (to the original) highest, then the functional information from the oracle next highest, and finally the non-functional property we seek to improve (execution time). This is because we seek solutions that are better according to the non-functional criteria, but at least no worse according to the functional criteria. Other possibilities, that are more heretical in their approach to correctness, are discussed elsewhere [1].

F. Implementation Details

This section presents implementation details required to make the genetic improvement process practical and which may be required by other researchers for replication purposes.

1) *Aborts, Heap Errors, Segmentation Errors, Floating Point Exceptions, Assert Exceptions:* Almost all evolved programs that compile run all of their test cases and produce an answer on each. Only 6% fail. The most frequent causes of run time failure are segmentation faults (≈3%) and CPU time limit overruns (≈2%). The remaining 0.6% of runs either abort (e.g., due to heap corruption), report a floating point error (e.g., divide by zero) or fail one of Bowtie2's own `assert` exception checks.

2) *Zombies:* In Unix, a process can sometimes fail in such a way that the operating system has difficulty cleaning up after it and instead of terminating it Unix places it in a zombie state. Since such zombie processes do not terminate or timeout, a zombie could potentially hold up our GP indefinitely. We found that zombies occur infrequently. Indeed, none were created during the runs described in Section IV-F1. Nevertheless, in order to guard against it we used a background zombie killer.

If the machine is overladen then the failure of a process to respond might be due to its failure to receive any computation time. In our experiments we set this overladen threshold to 16. This means that the machine is considered overladen should there be 16 or more processes awaiting scheduling all of which are able to proceed. Sixteen seemed to be a suitably conservative value, given that our experiments ran on a machine with eight cores. Once per minute the zombie killer background process compares the CPU time taken by each fitness job with that taken by it in the previous minute. If they are identical the job is killed (provided the machine is not overladen), freeing the GP to continue to the next fitness test.

3) *Details of Compilation Failures and Aborted Runs:* Throughout the run (see Fig. 11) about 26% of compilations fail. (All but six compilation failures are caused by moving variables out of their scope).

4) *Comparing the Original Program:* As explained in Section III-H, the instrumented (but otherwise unchanged) Bowtie2 code is run on the test cases selected as the training set for the current generation. (See Section III-D). This took an average of 38 s.

TABLE II
FITNESS SUNCTION

- Each generation chose uniformly at random one test case from each O–Bin (Section III-A, page 3).
- **Fitness test** original (albeit instrumented) system on each of these test cases
- **Fitness test** each member of the population
  - Generate modified source code and then compile it.
  - If fails to compile, GP individual cannot be a parent so skip rest of fitness testing
  - For each of the selected test cases:
    - Run modified system (subject to time out, 20 or 60 secs)
    - For each reported match with the human genome
      - Calculate Smith-Waterman score for test case v. match
      - If optimal Smith-Waterman match is not where it was claimed to be, subtract the distance between claimed match and Smith-Waterman match from Smith-Waterman score (subject to score not being made negative).
    - calculate mean Smith-Waterman score for this test case and store number of instructions executed
- At the end of the generation, each member of the population which compiled is compared with the original code on each of the test cases.
  If a modified version of the system takes fewer instructions or has higher mean Smith-Waterman score (unless it never claims any matches) it is eligible to potentially become a parent.
- The eligible variants are sorted in order:
  1) Number of test cases where it did not fail or time out and was not aborted by the zombie killer (Section IV-F2)
  2) Mean Smith-Waterman score
  3) Number of instrumented lines of C++ run (total on all test cases). To be minimised.
- The top popsize/2 are selected to be parents of the next generation. If fewer than popsize/2 variants are eligible, two children per missing variant are created at random (in the same way that the initial population was created.)

G. Role of the Smith Waterman Score in the Post Evolution Clean

To avoid excessive run time, the hill climber used to minimize the evolved solution (see Section III-K) uses a

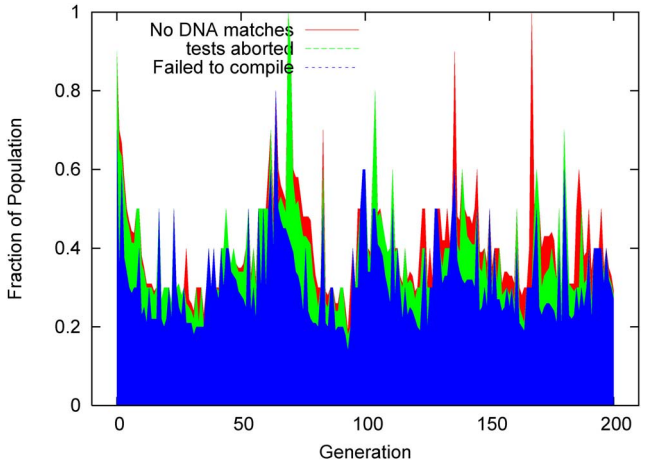

Fig. 11. Fraction of population which fails to compile (bottom), aborts on one or more test cases (light) or fails to find any match in the human genome (top). In generation 167 there are no suitable parents and the population is reset. Data smoothed (by averaging to right over ten generations).

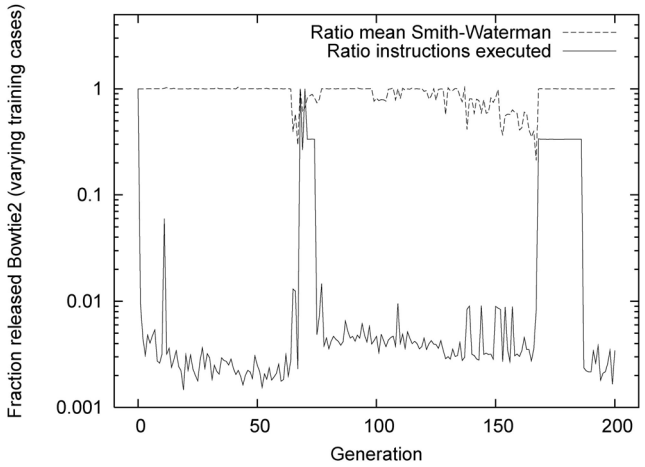

Fig. 12. Evolution of performance on randomly changing training cases. In the last generation, the best uses 290 times fewer C++ statements than the original code. In most generations, there is a small improvement in mean Smith-Waterman score (dashed line), which is obscured by the use of a log scale needed to clearly show instruction execution data.

(fixed) subset of all the training data. One hundred different DNA sequences were randomly chosen from each of the five classes described in Section III-D. (Only 41 of the 5250 Solexa training sequences match in the human genome reference sequence between 100–200 times. So they are all used.) This gives 441 training DNA sequences for the hill climber.

The Smith Waterman score is used to winnow the mutations listed in each individual evolved by generic improvement. A smaller mutation is considered worse if the following take place.

1) It does not compile.
2) It uses more than 1% more instructions than the evolved version.
3) If fails to find a match.
4) The Smith-Waterman score of all the matches it reports for a DNA sequence is on average more than 1.0 lower.
5) If ten or more of the matches it finds have on average lower Smith-Waterman scores than the evolved version.

If, according to this definition of worse, the removal of a change from an individual fails to make the resulting version of Bowtie2 worse, the change is considered unimportant and it is permanently removed from the individual.

V. Genetically Improved Bowtie2

Section V-A will describe the evolution of performance, both on the per generation training sets and on a fixed sample representing all the training data. Section V-B describes out-of-sample performance and then Section V-C describes out-of-sample performance of Bowtie2 when modified by the minimized genetic change. Section V-D tries to explain the minimized change's impact on Bowtie2.

A. Performance During Evolution

We evolved a population of 10 Bowtie2 variants for 200 generations. Fig. 12 shows the best in the population's fitness at each generation. There is a dramatic improvement in speed and a very small improvement in the mean Smith-Waterman score. Fig. 6 plots the evolution of genotype size. Notice that size increases (bloat) under the action of our crossover (see Section III-J). However, the population is reinitialized near generations 65 and 167. That is, twice during evolution the population contained very few good individuals and, as described in Section III-H, the poor ones were replaced by reinitializing them in the same way that the initial population is created.

To show training performance in general, every ten generations, we retested the best-of-generation program on a much bigger and fixed subset of the 5250 training Solexa DNA sequences described in Section IV-A2. Apart from the random number seeds, we used the same procedure as in Section IV-G to select 441 DNA sequences. In the last generation, the best individual used only $1/290^{th}$ of the instructions used by the instrumented Bowtie2 on the five DNA sequences used for training in generation 200. When both were tested on the 441 DNA sequences, one at a time, the ratio was 1 to 500.

The speed (as a fraction of the number of instructions used by the unmodified instrumented version of Bowtie2) of the best of each generation is plotted in Fig. 13. Fig. 13 shows the performance of the best in the population every tenth generation. Note that the versions of Bowtie2 are run once on the 441 test cases. (Rather than 441 times, once on each test case.) The best of generation 135 is described in the supplementary information.

As suggested in Section III-D, GP has been able to improve the initialization code. Therefore, when we use the 441 DNA sequences in a single file, improvements in the initialization code have less proportionate effect. Nevertheless, Fig. 13 shows that in most cases the best evolved version of each generation still uses only about a fifth of the instructions used by the unmodified Bowtie2. Also Fig. 13 shows that in most generations the best of the generation evolved version finds DNA matches in the human genome, which on average, are at least as good as the original unmodified version of Bowtie2.

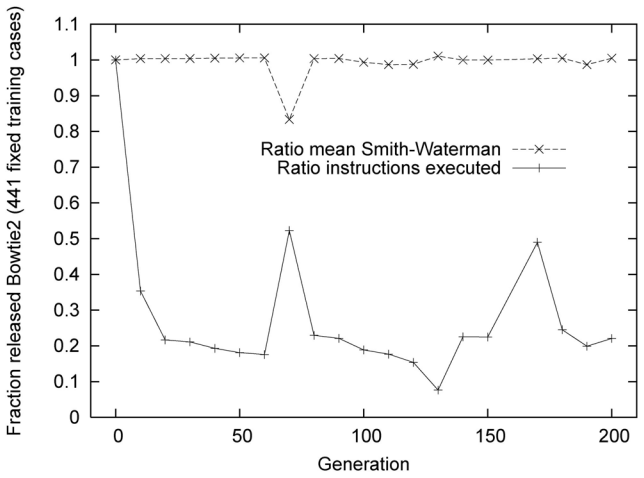

Fig. 13. Speed of best-in-population every ten generations on a fixed training set. (Fig. 12 gives GP fitness with training set, which is changed every generation.) Here, we run each genetically improved version of Bowtie2 on all 441 DNA sequences together.

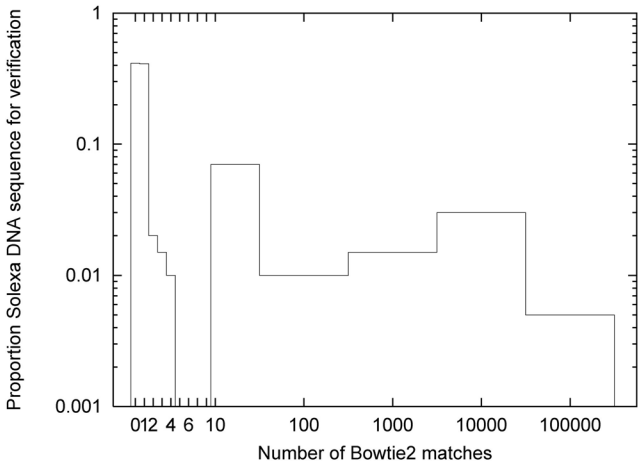

Fig. 14. Distribution of the number of matches in the human genome for verification Solexa DNA sequences. Note non-linear scales. With ten or more matches each bin contains data covering an order of magnitude. (see Fig. 7).

Again the improvement in Smith-Waterman score is small (e.g., 0.5% in generation 200.)

B. Performance Comparison on Hold Out DNA Sequences

For verification, 10 DNA scans were randomly chosen from 2 different individuals (one male, one female, NA12891 NA12892, being the parents of NA12878) giving 20 complete Solexa scans (a total of 176 893 951 DNA sequences). Ten DNA sequences were chosen from each Solexa scan. The distribution of the number of matches for these 200 DNA sequences, which is strongly related to run time, is shown in Fig. 14. Apart from sampling noise, it should be the same as the whole of the Solexa scans. Notice that it contains a few sequences, which match a very large number of times. As we described in Section IV-A2, such sequences were deliberately excluded from the training data, as they cause the released version of Bowtie2 to become very slow.

The released version of Bowtie2 and the evolved version were both tested on the 200 hold out DNA sequences. Neither was instrumented and both were compiled with the same compiler optimizations as are used in Bowtie2's installation kit (i.e., gcc -O3). The evolved version took 3.9 h. The released code took 12.2 days. Thus, we observe that, on average, the genetically improved program is 74 times faster on out-of-sample data.

In 178 cases (89%) the GP version of Bowtie2 produced identical results to the released code. In 17 of the remaining 22 cases the GP version found fewer matches (median reduction 0.8%, $p = 2\ 10^{-5}$ sign test). In 3 cases (1.5%), the reduction in the number of matches found was more than 40% but, in each case, the matches had a much better mean Smith-Waterman score. The GP version never reported more matches nor did it ever incorrectly report zero matches. Recall that multiple matches, particularly if low quality, are not normally useful to biologists. In 18 cases (9%), the GP version was better (i.e., the matches it reported had a mean Smith-Waterman score better than that of the released code). In 1 case, the Smith-Waterman score was identical and in 3 (1.5%) the scores were worse but differed only in the 4th and 6th significant decimal place ($p = 0.001$ sign test). The median improvement was 0.1 (max 6.32).

C. Minimizing the Final Evolved Variant

The best-in-generation 200 evolved individual (see supplementary information) makes 39 changes to the released version of Bowtie2. Using the clean up procedure described in Section IV-G, this was reduced to seven changes (shown in Fig. 15). Of course other techniques, such as diffx, might also be able to simplify it. The reduced version was compiled in the same way as the released code (i.e., gcc -O3) and tested on the 200 verification DNA sequences. It produced identical output to the evolved 39 changes version and was 4% faster, giving a speed up compared to the released code on the hold out DNA sequences of 77 times.

D. Optimizations Provided by Bowtie2^{GP}

The following sections try to explain the important optimizations found by GP (see Fig. 15).

1) *bt2_io.cpp line 622:* Source file bt2_io.cpp is concerned with reading the indexed human genome reference sequence from disk files (total 3.79 GB). In training cases with the original Bowtie2, line 622 (top of Fig. 15) is used 179 215 892 times at the start of each run (see Fig. 9). Line 622 is a for loop: `for(uint32_t i = 0;i < offsLenSampled;i++)` which genetic programming replaces with `for(uint32_t i = 0; i < this->_nPat; i++)`. Since `this->_nPat` typically has a value of 84 (rather than `offsLenSampled`'s 179215892) the whole loop is iterated far fewer times. However, the loop's body comprises only various assert statements. Eventually, in optimized non-instrumented production code, they are all removed by the compiler. Thus, while substituting `this->_nPat` for `offsLenSampled` reduces the instrumented number of lines used in a run by 179 215 808, it has no effect on (production) run time (we can view this as GP rediscovering a gcc -O3 optimization).

2) *sa_rescomb.cpp lines 50 and 69:* Bowtie2 starts from exact matches between the DNA sequence and the human

Weight	Mutation	Source file	line	type	Original Code	New Code
999	replaced	bt2_io.cpp	622	for2	`i < offsLenSampled`	`i < this->_nPat`
1000	replaced	sa_rescomb.cpp	50	for2	`i < satup_->offs.size()`	`0`
1000	disabled	sa_rescomb.cpp	69	for2	`j < satup_->offs.size()`	
100	replaced	aligner_swsse_ee_u8.cpp	707		`vh = _mm_max_epu8(vh, vf);`	`vmax = vlo;`
1000	deleted	aligner_swsse_ee_u8.cpp	766		`pvFStore += 4;`	
1000	replaced	aligner_swsse_ee_u8.cpp	772		`_mm_store_si128(pvHStore, vh);`	`vh = _mm_max_epu8(vh, vf);`
1000	deleted	aligner_swsse_ee_u8.cpp	778		`ve = _mm_max_epu8(ve, vh);`	

Fig. 15. Minimized evolved solution. After unneeded changes have been removed, we are left with 7 changed lines, in three C++ source files. This version of Bowtie2 is 77 times faster on average than the released version on short DNA sequences generated by the Broad Institute's Solexa next generation scanner.

genome, which are given by a hashing algorithm. (These are known as seeds). Since, in general, each seed covers only a part of the input DNA sequence, Bowtie2 uses C++ source file sa_rescomb.cpp to see if the matching region can be extended to cover the whole of the input sequence.

Lines 50 and 69 of sa_rescomb.cpp (3^{rd} and 4^{th} rows of Fig. 15) are both for loops in C++ method SAResolveCombiner::tryResolving(). In both cases, GP modifies the central loop control part and sets it to false. This means neither loop body is ever executed. Fig. 9 shows the unmodified scaling characteristics of these two for loop bodies. Line 51 in sa_rescomb.cpp $\odot$ is inside the line 50 for loop. It scales as O(n^2). Where n is the size of Bowtie2's output. Line 70 $\times$ is inside the line 69 for loop. Unmodified, it scales as O(n^3). The line 50 for loop counts how many elements are yet to be resolved in `satup_`

```
size_t needResolving = 0; for(size_t i=0;
i<satup_->
                  offs.size();i++){
    if(satup_->offs[i] == 0xffffffff) {
        needResolving++;
    }
}
```

Typically replacing `i < satup_->offs.size()` with `0` means instead of `needResolving` being set to up to 335, it remains as zero and the function immediately returns (so line 69 is never executed and the fact that it has also been disabled by GP never comes into play). Typically, disabling the line 50 for loop reduces the number of lines executed by 0–20% (the order plus the interaction between mutations to lines 69 and 50 mean the hill climbing simplification stage, (see Section III-K), could not spot that the change to line 69 was not needed as well as the change to line 50). Exiting the method early (i.e., just after the for loop on line 50) typically has no effect. This is because mostly `nfound` (set by for loop on line 69) is a lot smaller than `needResolving` so the condition `if(nfound == needResolving)` is false. (I.e., typically many elements in `refscan_` are also in `satup_`.) This means all the remaining code, which might update values used outside the method, is not executed. Also, the method's return value is always ignored. That is, GP avoids an expensive O(N^3) nested loop. The optimizing compiler cannot remove it because it does not know that typically it makes no difference to the final output.

3) *aligner_swsse_ee_u8.cpp line 707:* aligner_swsse_ee_u8.cpp lines 707, 766, 772 and 778 are in C++ method SwAligner::alignNucleotidesEnd2EndSseU8`()`. alignNucleotidesEnd2EndSseU8 uses Intel SSE instructions that operate on 16 unsigned 8-bit values packed into a single 128-bit register to solve the current alignment problem. All four modified lines are in a loop which processes each character in the reference text. This scales as O(n). However, lines 766, 772 and 778 are in a nested while loop which causes them to be executed about four times as many times as line 707 (still O(n) of course).

Starting with line 707, its replacement `vmax = vlo;` has no effect since `vmax` is already assigned to `vlo;`. Thus the change effectively deletes line 707. In the original code `vh = _mm_max_epu8(vh, vf);` `vh` is used as a scratch variable for 16 parallel SSE instructions. The instruction on line 706 `vh = _mm_max_epu8(vh, ve);` sets each element of `vh` to the maximum of the corresponding elements of `vh` and `ve`. Line 707 should have set them to the maximum of `vh`, `ve` and `vf`. Typically, only in 4% of cases is an element `vf` bigger than both `vh` and `ve`.

4) *aligner_swsse_ee_u8.cpp lines 766 and 772:* As with line 707, the replacement for line 772 makes no difference (since it too repeats exactly an existing calculation). Hence, line 772 `_mm_store_si128(pvHStore, vh);` is effectively deleted. `pvHStore` is a pointer (type `__m128i*`) to where the 16 results in `vh` should be stored next. It is incremented by `ROWSTRIDE` (4). As the store instruction on line 772 has been deleted, `pvHStore` should not be incremented. GP achieves this by deleting line 766 (probably a human programmer would have deleted the identical instruction on line 773, but deleting line 766 has the same effect).

It appears the immediate effect of not saving `vh` comes on lines 788 and 796 where the next value of `vh` should have been loaded but instead the old one is reloaded. It appears that this terminates the inner `while` loop after the next iteration. So typically reducing the number of times the `while` loop loop is used by a factor of about twelve, with almost no impact on the output. That is, it appears GP has discovered that later gap extensions are relatively unimportant, whereas originally Bowtie2 spent a long time searching for them.

5) *aligner_swsse_ee_u8.cpp line 778:* Line 778 `ve = _mm_max_epu8(ve, vh);` (see line 707 above) has the effect of updating elements in `ve` in the 25% of cases

where vh is bigger than the element in ve. Thus, in three quarters of cases removing line 778 has no effect.

The net combined effects of all seven modifications in Fig. 15 are somewhat subtle. They are somewhat like "loop perforation" [18]. However, loop perforation acts on complete loop iterations, e.g., to speed convergence, rather than manipulating program instructions which are repeated many times as they lie within nested loops. Indeed, the interactions of the modified program instructions may indirectly determine if the loop is repeated or not.

VI. Related Work

While genetic programming has been used many times in software engineering, e.g., in project management [19] and testing [20], we are particularly concerned with evolving code. GP has not demonstrated an ability to write large programs normally associated with programming. However, even modest amounts of evolved code can be useful.

A. Current and Existing Research

Martin [21] showed that genetic programming can build a telephone call rerouting service (home/office) from existing telephony components. Although clearly not a like-for-like comparison, Martin claimed GP elapsed times of about a minute, whereas a commercial study showed that for a complex service a team of engineers required 4.5 man years of effort to analyse, design, code, and test the service (note the commercial study also includes non-coding business activities whereas GP elapsed time covers only coding). More recently, Rodriguez-Mier *et al.* [22] showed that GP can create novel web services by combining human written existing web services using the web ontology language (OWL). Notice the power of web mashups comes from quite small amounts of glue logic. Glue logic might also be usefully evolved to combine other types of software: perhaps on the same server, perhaps written in the same or different languages (e.g., PHP and Cobol). Combinations that are rare or difficult for a human programmer or where skills are difficult to find may turn out to be no more difficult for a machine than a routine combination. Similarly, once objectives can be quantified, it may be as easy for a machine to juggle multiple objectives (perhaps a mixture of functional and non-functional requirements) which a human programmer would be hard pressed to meet simultaneously and would, in practice, tackle only one at a time.

Another approach to side-stepping the scaling problem is Yamamoto and Tschudin's [23] fraglets approach. This uses GP in an artificial chemistry-like approach, whereby small fragments of existing code are combined. Yamamoto and Tschudin [23] considered evolving network communications protocols. More recently, Weise and Tang [24] evolved distributed algorithms (election, critical selection for mutual exclusion distributed locking, and distributed greatest common divisor) using a GP rule based approach.

One of the earliest attempts to evolve software considered evolving hashing functions [25]. (See also [26], [27], and [28]). Hashing is often used to speed up search. A hash function takes an object (typically a string) and deterministically converts it to one of the legitimate indexes into a data store. while hashing typically takes constant time, the search in the data store typically grows with the number of items with the same hash index. An efficient hash function will ensure commonly used objects hash to (relatively) unique indexes. There are many good hash functions but how good a hash function is in practice depends upon the distribution of objects it has to deal with. Often this is not known in advance by the programmer. So a generic hash function may do poorly with specific examples. Human written hash functions may be tuned for a certain load. Tuning usually means the hash function is coded, tested, and recoded in response to the tests, tested again and so on. Notice this is effectively a manual version of the classic evolutionary algorithm: generate and test, then regenerate and retest and so on.

Memory management is another area where traditionally people try to write generic code by making assumptions about typical patterns of use but a specific implementation may turn out to be inefficient when used in unanticipated ways. Risco-Martin *et al.* [29] showed grammatical evolution [30] is able to evolve efficient heap managers for specific circumstances. They also considered non-functional requirements, like power consumption.

Another case where there is no universal optimal strategy is data caching. For caching between CPU and RAM to be effective it must be very fast. Hence, its algorithms are simple and implemented in hardware. While both word size and number of independent cache stores vary, even in the same product line, the use of cache lines is very common. Both Paterson and Livesey [31] and O'Neill and Ryan [32] evolved software that decides which data to expel from the cache (i.e., which cache line to flush). Both claim to do better than the common heuristic least recently used and their evolved code is small enough (1–3 lines typically) that it might be implemented in hardware. O'Neill and Ryan also suggest their solutions are more generic than those in [31].

Sipper *et al.* used GP to improve existing Java programs (symbolic regression, artificial ant on the Santa Fe trail, separating intertwined spirals, array sum, and tic-tac-toe [3]). Their initial population is seeded with Java byte code [33] from an existing poor program.

Weimer *et al.*'s work on using GP to automatically fix bugs [9] is increasingly well known. Most of this paper is at the level of C source code but his group has also shown bugs can be fixed in lower levels. They have also used GP to improve GPU shaders [34]. Rinard's group have also used non-evolutionary ways to make non-semantic preserving transformations to GPU kernels [18]. Other recent work on evolving fixes for bugs include [16], [35].

Arcuri and White [4] considered not only automatic bug fixing [5] but also have shown GP can improve programs. They considered several well-known software engineering benchmarks, including triangle, sort, factorial, remainder, switch 10 and select (select is the largest at 94 LOC). They showed GP can improve the source code and find optimizations, which the compiler was unable to find. For example, they showed GP reduced the number of instructions executed by factorial

(in Java) by 87.4%. They also showed GP can optimize non-functional properties. For example, they evolved pseudo random number generators (PRNG), which trade randomness (as measured by information entropy) against reduction in power consumption.

In previous work [2] we showed that non-trivial amounts of code can be automatically created with a test based fitness function and starting from a BNF grammar that constrains the code to be legal, compilable, executable and terminating. We chose the unix gzip compression utility and demonstrated the evolution of the longest_match routine within it. GP was asked to evolve a replacement for longest_match which ran on different hardware (an nVidia graphics card) and slightly different programming language (nVidia's CUDA).

GP offers a potential way of moving existing applications to mobile platforms [36] where, not only is the hardware radically different, but so too are user interfaces and expectations. Compiler based reoptimization is not sufficient but evolutionary computation may be able to provide the more radical changes to the program sources needed.

The grammar approach we have described here is based on our earlier work on mutation testing [10] where a grammar is used to describe a source code and the mutations (variations) that can be applied to it. Grammars have been used widely in genetic programming however, except for work on bug fixing [9] and our own, the grammars tend to be general and try to represent any solution, rather than to represent a substantial existing human written program and variations from it. Grammatical evolution (GE) [30] is the most widespread grammar based GP approach, with several hundred papers. As yet, GE has been used only a few times in software engineering (for compiler optimization [37] and, as mentioned above, for memory heap management [29]). Non-GE grammar based genetic programming approaches to software engineering include project effort estimation [38] and web services composition [22].

B. Less Explored areas for Evolving Software

Another potential software engineering application of GP is to use evolutionary computation to produce diverse variants of programs [39]. While multiplicity computing [6] currently considers a few different versions of programs, GP can already produce populations of variants. While some of the population are not suitable for use, if evolution is allowed to continue after it finds the first solution, in subsequent generations, the population can contain hundreds of solutions [2]. Obviously, the number of program variants could be increased still further. Additional non-functional criteria might be introduced into fitness selection. These criteria might ensure a certain minimum level of variation [40] or ensure evolved version are not too extreme. In some cases it might even be possible to ensure every user had their own variant of the program. Uses of diverse software might include resilience, both to attack and faults, and for watermarking.

Software product lines present a slightly different need. Instead of wanting different versions of the same program, software product lines represents a way to manage functionally different versions of a program to meet different customer needs. This might be for customers who speak different languages (internationalization) but also covers things like embedded controllers in similar but not identical hardware. e.g., a deluxe microwave oven may have many features not available in the economy version; both have the same controller chip but need different software. Current approaches mostly consider only enabling and disabling parts of the source code but in the future these parts might form the components of more flexible systems glued together by evolutionary computing.

VII. Discussion

In the future we intend to use GISMOE to optimize the non-functional properties of a number of diverse programs. We hope that GISMOE, perhaps incorporating hyperheuristics, will form the basis for dynamically adapting computing.

We have demonstrated the GISMOE approach on C/C++ and related languages, i.e., CUDA, nevertheless GISMOE should be applicable to improving software written in other high level languages. Indeed similar approaches could well work on assembler programs and even at the binary level. Each new programming language would need a set of mutation operators, which change the target program and yet retain a reasonable chance that the modified code will compile and run. We expect that initially these will be based on our cut-and-paste operators (see Section III).

Although there has been interesting work on interactive evolution, it appears that user fatigue will limit subjective fitness functions to applications, which can effectively use crowd sourcing. Therefore, GISMOE really needs a cheap automatic way of rating the relative effectiveness of individual members of a population of program modifications during evolution. There may be a connection with automatic test case generation whereby tests can be cheaply created which target modifications created by GP. Once the population has evolved one or more potential solutions, more expensive testing, analysis, and proving techniques might be applied to the final outcome. To be cheap enough during evolution, testing will probably not use all of the available test cases.

We feel we have been fortunate in that the evolved program gives an immediate and large payback for about one CPU core day computation. Other applications may yield less. Still larger applications might increase the cost, but this might be offset by other improvements. We have used make and precompiled headers to reduce compilation cost. These and similar incremental compilation approaches are widely available.

In many cases it will be straightforward to estimate the benefit by multiplying the improved performance by the number of users and the number of times they run the program. However, a future benefit may be more valuable: where the optimization enables the program to be used where initially it would have been impracticable. For example, optimizing existing code for mobile or embedded systems or other resource constrained platforms. It is interesting that modern just in time compilers find the cost of monitoring/optimizing code can be offset against the improved performance of the optimized code, even for a single user. The GP approach is currently more expensive than an optimizing compiler. However, it might be

run over night while the user is sleeping (dream optimization), and it can readily be parallelized. Also, the optimizations GP can consider are much wider, so we may one day see embedded JIT evolutionary optimization.

VIII. Conclusion

Considerable manual effort is needed to create programs. Even identifying new operating points for tools that are suitable for new circumstances or new user requirements is labor intensive and few can afford to even explore more than one possibility by hand. Automated software production offers the prospect of exploring complete Pareto trade-off surfaces, for example, between functionality and speed.

With this in mind, for the first time, we have evolved specific improvements to substantial multifile C++ code using a fitness function, which compares the output of new code with that of the old to ensure a principled trade-off between existing functionality and improved non-functional requirements. In this example, we find a trade-off that improves both functional and non-functional performance. Starting from 50 000 lines of code, the search is progressively concentrated. It first focuses in on 20 000 lines, then 2744 lines (see Section IV-D), then using weights (see Section III-C) GP finds a solution of 39 changes, which can be reduced to just seven changes in three source files (see Fig. 15). While we may not be so fortunate with other programs, on held out test cases, the evolved version of Bowtie2, on average, yields slightly better answers and is more than 70 times faster.

Acknowledgment

The authors would like to thank B. Langmead, S. Forrest, M. Gabel, P. Devanbu, J. Dolado, A. Arcuri, M. O'Neill, D. R. White, G. Wilson, and W. Weimer.

References

[1] M. Harman, W. B. Langdon, Y. Jia, D. R. White, A. Arcuri, and J. A. Clark, "The GISMOE challenge: Constructing the Pareto program surface using genetic programming to find better programs," in *Proc. 27th IEEE/ACM Int. Conf. ASE*, Sep. 2012, pp. 1–14.

[2] W. B. Langdon and M. Harman, "Evolving a CUDA kernel from an nVidia template," in *Proc. IEEE World Congr. Comput. Intell.*, Jul. 2010, pp. 2376–2383.

[3] M. Orlov and M. Sipper, "Flight of the FINCH through the Java wilderness," *IEEE Trans. Evol. Comput.*, vol. 15, no. 2, pp. 166–182, Apr. 2011.

[4] D. R. White, A. Arcuri, and J. A. Clark, "Evolutionary improvement of programs," *IEEE Trans. Evol. Comput.*, vol. 15, no. 4, pp. 515–538, Aug. 2011.

[5] A. Arcuri and X. Yao, "A novel co-evolutionary approach to automatic software bug fixing," in *Proc. IEEE World Congr. Comput. Intell.*, Jun. 2008, pp. 162–168.

[6] C. Cadar, P. Pietzuch, and A. L. Wolf, "Multiplicity computing: A vision of software engineering for next-generation computing platform applications," in *Proc. FSE/SDP Workshop Future Software Eng. Res.*, Nov. 2010, pp. 81–86.

[7] B. Langmead and S. L Salzberg, "Fast gapped-read alignment with Bowtie 2," *Nature Methods*, vol. 9, no. 4, pp. 357–359, 2012.

[8] D. Altshuler, R. M. Durbin, G. R. Abecasis, D. R. Bentley, A. Chakravarti, A. G. Clark, *et al.*, "A map of human genome variation from population-scale sequencing," *Nature*, vol. 467, no. 7319, pp. 1061–1073, Oct. 2010.

[9] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, "GenProg: A generic method for automatic software repair," *IEEE Trans. Softw. Eng.*, vol. 38, no. 1, pp. 54–72, Jan.-Feb. 2012.

[10] W. B. Langdon, M. Harman, and Y. Jia, "Efficient multi-objective higher order mutation testing with genetic programming," *J. Syst. Softw.*, vol. 83, no. 12, pp. 2416–2430, Dec. 2010.

[11] W. B. Langdon, "A many threaded CUDA interpreter for genetic programming," in *Proc. 13th EuroGP*, Apr. 2010, pp. 146–158.

[12] S. R. Maxwell, III, "Experiments with a coroutine model for genetic programming," in *Proc. IEEE World Congr. Comput. Intell.*, Jun. 1994, pp. 413–417.

[13] A. Teller, "Genetic programming, indexed memory, the halting problem, and other curiosities," in *Proc. 7th Annu. FL Artif. Intell. Res. Symp.*, May 1994, pp. 270–274.

[14] M. Gabel and Z. Su, "A study of the uniqueness of source code," in *Proc. 18th ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, Nov. 2010, pp. 147–156.

[15] H. Zhang, "Exploring regularity in source code: Software science and Zipf's law," in *Proc. 15th WCRE*, Oct. 2008, pp. 101–110.

[16] T. Ackling, B. Alexander, and I. Grunert, "Evolving patches for software repair," in *Proc. 13th Annu. Conf. Genetic Evol. Comput.*, Jul. 2011, pp. 1427–1434.

[17] W. B. Langdon, T. Soule, R. Poli, and J. A. Foster, "The evolution of size and shape," in *Advances in Genetic Programming 3*. Cambridge, MA, USA: MIT Press, Jun. 1999.

[18] S. S.-Douskos, S. Misailovic, H. Hoffmann, and M. C. Rinard, "Managing performance vs. accuracy trade-offs with loop perforation," in *Proc. SIGSOFT FSE*, Sep. 2011, pp. 124–134.

[19] J. J. Dolado, "A validation of the component-based method for software size estimation," *IEEE Trans. Softw. Eng.*, vol. 26, no. 10, pp. 1006–1021, Oct. 2000.

[20] M. C. F. P. Emer and S. R. Vergilio, "GPTesT: A testing tool based on genetic programming," in *Proc. GECCO*, Jul. 2002, pp. 1343–1350.

[21] P. Martin, "Genetic programming for service creation in intelligent networks," in *Proc. EuroGP*, Apr. 2000, pp. 106–120.

[22] P. R.-Mier, M. Mucientes, M. Lama, and M. I. Couto, "Composition of web services through genetic programming," *Evol. Intell.*, vol. 3, nos. 3–4, pp. 171–186, 2010.

[23] L. Yamamoto and C. F. Tschudin, "Experiments on the automatic evolution of protocols using genetic programming," in *Proc. Autonom. Commun. 2nd Int. IFIP Workshop WAC*, Oct. 2005, pp. 13–28.

[24] T. Weise and K. Tang, "Evolving distributed algorithms with genetic programming," *IEEE Trans. Evol. Comput.*, vol. 16, no. 2, pp. 242–265, Apr. 2012.

[25] D. Hussain and S. Malliaris, "Evolutionary techniques applied to hashing: An efficient data retrieval method," in *Proc. GECCO*, Jul. 2000, p. 760.

[26] P. Berarducci, D. Jordan, D. Martin, and J. Seitzer, "GEVOSH: Using grammatical evolution to generate hashing functions," in *Proc. 15th MAICS*, Apr. 2004, pp. 31–39.

[27] C. Estebanez, J. C. H.-Castro, A. Ribagorda, and P. Isasi, "Evolving hash functions by means of genetic programming," in *Proc. GECCO*, vol. 2, Jul. 2006, pp. 1861–1862.

[28] J. Karasek, R. Burget, and O. Morsky, "Towards an automatic design of non-cryptographic hash function," in *Proc. 34th Int. Conf. TSP*, Aug. 2011, pp. 19–23.

[29] J. L. R.-Martin, D. Atienza, J. M. Colmenar, and O. Garnica, "A parallel evolutionary algorithm to optimize dynamic memory managers in embedded systems," *Parallel Comput.*, vol. 36, nos. 10-11, pp. 572–590, 2010.

[30] M. O'Neill and C. Ryan, "Grammatical evolution," *IEEE Trans. Evol. Comput.*, vol. 5, no. 4, pp. 349–358, Aug. 2001.

[31] N. Paterson and M. Livesey, "Evolving caching algorithms in C by genetic programming," in *Proc. 2nd Annu. Conf. Genetic Program.*, Jul. 1997, pp. 262–267.

[32] M. O'Neill and C. Ryan, "Automatic generation of caching algorithms," in *Proc. Evol. Algorithms Eng. Comput. Sci.*, Jun. 1999, pp. 127–134.

[33] E. Lukschandl, M. Holmlund, and E. Moden, "Automatic evolution of Java bytecode: First experience with the Java virtual machine," in *Proc. 1st Eur. Workshop Genetic Programming*, Apr. 1998, pp. 14–16.

[34] P. S.-Amorn, N. Modly, W. Weimer, and J. Lawrence, "Genetic programming for shader simplification," *ACM Trans. Graph.* , vol. 30, no. 6, article no. 152, Dec. 2011.

[35] J. L. Wilkerson, D. R. Tauritz, and J. M. Bridges, "Multi-objective coevolutionary automated software correction," in *Proc. 14th Int. Conf. Genetic Evol. Comput. Conf.*, Jul. 2012, pp. 1229–1236.

[36] A. Cotillon, P. Valencia, and R. Jurdak, "Android genetic programming framework," in *Proc. 15th EuroGP*, Apr. 2012, pp. 13–24.

[37] H. Leather, E. Bonilla, and M. O'Boyle, "Automatic feature generation for machine learning based optimizing compilation," in *Proc. Int. Symp. Code Generat. Optimiz.*, Mar. 2009, pp. 81–91.
[38] Y. Shan, R. I. McKay, C. J. Lokan, and D. L. Essam, "Software project effort estimation using genetic programming," in *Proc. IEEE Int. Conf. Commun. Circuits Syst. West Sino Exposit.*, vol. 2, 2002, pp. 1108–1112.
[39] R. Feldt, "Generating diverse software versions with genetic programming: An experimental study," *IEE Proc. Softw. Eng.*, vol. 145, no. 6, pp. 228–236, Dec. 1998.
[40] J. R. Koza, F. H. Bennett, III, and O. Stiffelman, "Genetic programming as a Darwinian invention machine," in *Proc. EuroGP*, May. 1999, pp. 93–108.

William B. Langdon received the Ph.D. degree in genetic programming from University College London (UCL), London, U.K., which was sponsored by National Grid plc., London, U.K.

He is currently a Senior Research Fellow with UCL.

Mark Harman is currently a Professor of software engineering with the Department of Computer Science, University College London, London, U.K., where he directs the CREST centre. He is widely known for his work on source code analysis and testing, and was instrumental in the founding of the field of search based software engineering (SBSE), an area of research to which this paper seeks to make a contribution. He is collaborating with W. B. Langdon in the Engineering and Physical Sciences Research Council-funded GISMO and Dynamic Adaptive Automated Software Engineering (DAASE) projects, which partly supported the work presented in this paper.