Evolutionary Improvement of Programs

David R. White, Andrea Arcuri, and John A. Clark

***Abstract*—Most applications of genetic programming (GP) involve the creation of an entirely new function, program or expression to solve a specific problem. In this paper, we propose a new approach that applies GP to improve existing software by optimizing its non-functional properties such as execution time, memory usage, or power consumption. In general, satisfying non-functional requirements is a difficult task and often achieved in part by optimizing compilers. However, modern compilers are in general not always able to produce semantically equivalent alternatives that optimize non-functional properties, even if such alternatives are known to exist: this is usually due to the limited local nature of such optimizations. In this paper, we discuss how best to combine and extend the existing evolutionary methods of GP, multiobjective optimization, and coevolution in order to improve existing software. Given as input the implementation of a function, we attempt to evolve a semantically equivalent version, in this case optimized to reduce execution time subject to a given probability distribution of inputs. We demonstrate that our framework is able to produce non-obvious optimizations that compilers are not yet able to generate on eight example functions. We employ a coevolved population of test cases to encourage the preservation of the function's semantics. We exploit the original program both through seeding of the population in order to focus the search, and as an oracle for testing purposes. As well as discussing the issues that arise when attempting to improve software, we employ rigorous experimental method to provide interesting and practical insights to suggest how to address these issues.**

***Index Terms*—Coevolution, embedded systems, execution time, genetic programming, multiobjective optimization, non-functional criteria, search based software engineering.**

I. Introduction

INCREASINGLY, modern software must meet both functional and non-functional requirements. For example, non-functional requirements such as execution time, memory usage, and power consumption are of particular importance in designing software for low-cost, low-power embedded systems. Even where non-functional requirements are not specified explicitly, it is often implicitly desirable to reduce resource consumption.

Producing software to meet non-functional requirements is a difficult task for the programmers, because their primary concerns usually lie elsewhere, and the programming language, compiler and target platform constitute a complicated system. Small changes at the source level can lead to starkly different behaviors at execution time. Also, non-functional properties are often highly interdependent and present tradeoffs to the programmer.

Compilers may aid the programmer by attempting to optimize non-functional properties. However, compilers must apply a fixed set of transformations that guarantee to preserve the semantics of the original code [1]. They usually apply local transformations such as peep-hole optimizations, and they are heavily constrained by the decisions a programmer makes.

In this paper, we propose the use of evolutionary computation to improve the non-functional properties of existing software, in cooperation with the compiler and simulation of the target platform. We employ several evolutionary techniques: genetic programming (GP) for program manipulation, coevolution for effective testing, and multiobjective optimization (MOO) methods [2] to consider both functional and non-functional objectives.

Given an existing C function to be optimized as input, we describe a framework for optimizing this function. We use the original program to seed the population and then allow GP to carry out arbitrary manipulation of the program. The candidate solutions are tested separately for non-functional and functional fitness, taking into account an expected distribution of inputs, i.e., an operational profile [3]. The test cases are executed via interpretation for functional fitness, and we use a simulator to estimate the non-functional properties of the software.

Preserving semantic equivalence is the most challenging part of this optimization process. We cannot guarantee that the framework as it stands will produce output that is semantically equivalent to the input program: results must be verified manually. To gain confidence in the correctness of output we used coevolved test cases to test the semantics of the program, which we find to be an effective method.

In this paper, we compare our results against the GNU GCC compiler using the –O2 flag, and discover that GP can find improved optimizations that the compiler cannot. Perhaps a more sophisticated machine-learning compiler will be able to find the optimizations reported here, but we emphasize that our approach is not directly competing with compiler design. The proposed method can operate alongside a compiler as a sophisticated preprocessor, or to find novel optimization methods that can subsequently be incorporated into compiler design. To the best of our knowledge, we do not know of any other system that is able to automatically suggest such optimizations.

This paper develops our previous preliminary results [4] in several important directions: first, by validating this novel methodology on a variety of example program functions, and

Manuscript received October 5, 2009; revised February 19, 2010 and September 2, 2010; accepted September 9, 2010. Date of publication January 17, 2011; date of current version July 29, 2011. This work was supported by SEBASE (EP/D050618/1) and by the Norwegian Research Council.

D. R. White and J. A. Clark are with the Department of Computer Science, University of York, York, YO10 5GH, U.K. (e-mail: drw@cs.york.ac.uk; jac@cs.york.ac.uk).

A. Arcuri is with the Simula Research Laboratory, Lysaker 325, Norway (e-mail: arcuri@simula.no).

Digital Object Identifier 10.1109/TEVC.2010.2083669

second by carrying out more detailed empirical analysis of the method and reporting new results. Third, we investigate new ways of seeding a population. Thus, the emphasis is on further developing and testing this approach to non-functional optimization.

This paper is organized as follows. Section II describes related work. We then give a high-level background of evolutionary optimization in Section III. This is followed by a problem definition in Section IV. Section V describes our framework, and Section VI presents our case studies. Implementation details are discussed in Section VII. Sections VIII and IX describe our results. The limitations of our method are listed in Section X. Section XI suggests further work. Finally, Section XII concludes the paper.

II. Related Work

Whilst we are not aware of any work that has proposed a general approach to solving the problem we outline in this paper, there are examples of related work in the literature that apply evolutionary methods within the context of efficiency.

In this section, we discuss compiler optimization, almost exclusively the current method employed to optimize software in this way, we look at previous work involving GP that we build upon, and finally review the important topic of seeding that is crucial in exploiting the input program.

A. Improving Compiler Performance

Compilers employ a range of techniques to optimize non-functional properties of code [5], albeit mostly through localized transformations. Most previous work has therefore focused upon the use of evolutionary techniques at the compiler interface, to find the most effective combination of such optimizations. For example, evolutionary algorithms have been used to optimize solution methods for NP problems. Stephenson *et al.* [6] used GP for solving hyperblock formation, register allocation, and data prefetching. Leventhal *et al.* [7] used evolutionary algorithms for offset assignment in digital signal processors. Kri and Feeley [8] used genetic algorithms for register allocation and instruction scheduling problems.

Compilers use sequences of code optimization transformations, and these transformations are highly correlated with each other. In particular, the order in which they are applied can have a dramatic impact on the final outcome. The combination and order of selected transformations can be optimized using evolutionary algorithms: for example, the use of genetic algorithms to search for sequences that reduce code size has been studied by Cooper *et al.* [9]. Similar work with genetic algorithms has been done by Kulkarni *et al.* [10], and Fursin *et al.* [11] used machine learning techniques to decide which sequence of code optimization transformations to employ when compiling new programs.

Compilers such as GCC give the user a choice of different optimization parameters, and to simplify their selection, predefined subsets of possible optimizations (e.g., –O1, –O2, and –O3 for optimizing execution time and –Os to reduce size). The benefits of a particular set of optimizations over another are dependent on the specific code undergoing optimization. Hoste and Eeckhout [12] therefore investigated the use of a multiobjective evolutionary algorithm to optimize parameter configurations for GCC.

B. High-Level Optimization

Much less work has been published in optimization of program characteristics through direct manipulation of the software itself, rather than the actions of the compiler.

Optimization of a program must assume, explicitly or implicitly, an expected distribution of input, and partitioning the input space may aid conventional optimization methods. Li *et al.* [13] investigated the use of genetic algorithms to evolve a hierarchical sorting algorithm that analyzes the input to choose which sorting routine to use at each intermediate sorting step. This partitioning of the input space is conceptually similar to the development of *portfolio* algorithms [14].

A more involved technique is to allow arbitrary manipulation of software code through automated programming methods. By far the most popular evolutionary method of creating new software is GP, thanks to its applicability and versatility. The most immediate example of considering non-functional properties of software with GP is the control of bloat (see Section III-A), although we do not treat bloat itself as a program characteristic: it is simply an artefact of the search algorithm.

Perhaps the only prior work with explicit goals similar to our own is that on program compression by Langdon and Nordin [15], where the authors attempted to reduce the size of existing programs using GP. They used a multiobjective approach to control the size of programs, having started with existing solutions. They applied this approach to classification and image compression problems. They were particularly interested in the impact such a method would have on the ability of final solutions to generalize. Interestingly, they too used seeding, as discussed in the next section.

During the software life-cycle, code changes may decrease the quality of software. There are different metrics to describe the quality of software [16], which is a non-functional property. Re-factoring aims to reverse this decline in software quality by using sequences of semantics preserving transformations on the software to improve those quality measures. An example of transformation is to move one static method from one class to another one (in the case of object-oriented software). Harman and Tratt [17] analyzed the performance of a Pareto-based MOO to address this task.

Automatic parallelization of code is important to improve the performance of sequential software when it is run on parallel hardware. Ryan *et al.* [18] investigated the use of GP to evolve sequences of semantics preserving transformations to automatically parallelize sequential code.

C. Seeding

It is well-known that the starting point of a search within the solution space can have a large impact on its outcome. It may be considered surprising, then, that more research has not focused on the best methods to sample the search space when creating the initial generation within GP. The crucial importance of domain-specific knowledge in solving optimization problems is also clear: yet little sound advice can be given on how best to incorporate existing information, such as low-quality or partially complete solutions to a problem, into an evolutionary run.

There are, however, examples of previous applications that employ some kind of seeding, by incorporating solutions generated manually or through other machine learning methods. Langdon [19] initialized a GP population based on the results of a genetic algorithm, whereas Westerberg and Levine [20] used advanced seeding methods based on heuristics and search strategies such as depth first and best first search. In both cases, these seeding strategies obtained better results than random sampling. Marek *et al.* [21] seeded the initial population based on solutions generated manually.

Our approach here is similar to our previous work on *Automatic Bug Fixing* [22], in which all the individuals of the first generation were copies of the original incorrect software, and the goal is to evolve a bug-free version. Similarly, subsequent work by Forrest *et al.* [23] shares similarities with our original work [4], in that they used seeding and attempt to improve an existing program using GP, although their focus is on functional correctness alone.

The most relevant application of seeding in the literature is from Langdon and Nordin [15], who employed a seeding strategy in order to improve one aspect of a solution's functional behavior: its ability to generalize. The initial population was created based on perfect individuals, where the goal of the evolutionary run was to produce solutions that were more parsimonious and had an improved ability to generalize.

Recently, Schmidt and Lipson [24] investigated six different seeding strategies for GP. Their experiments were carried out on 1000 randomly generated symbolic regression problems. In their empirical analysis, they found that seeding strategies that use only parts of the input program give better results than using exact copies.

III. Background

In this section, we give an overview of the three key areas of evolutionary computation used: GP, coevolution, and MOO. A reader well-versed in these areas should be able to omit reading this section without any loss of clarity.

A. Genetic Programming

GP [25] is a paradigm for evolving programs and can be regarded as a branch of *machine learning* [26]. Although first applications of evolutionary techniques to produce software can be traced back to at least as early as 1985 with Cramer [27], it was not until Koza [28] popularized the technique in 1992 that the method became widely adopted and subsequent successful applications in many fields followed (e.g., [29]).

It is commonly the case that solutions within GP are evaluated based on a set of test cases, which are pairs of input and corresponding desired behavior (x_i, y_i). A *training set* T is a subset of all possible pairs, and the goal is to *evolve* a program p that can correctly pass these test cases, i.e., $\forall(x_i, y_i) \in T,\ p(x_i) = y_i$. The fitness function provides a measure of the distance an individual resides from the desired behavior y_i, usually based on the difference between evaluations of $p(x_i)$ and y_i.

Issues such as *generalization* of the resulting programs and *noise* in the training data are common to all machine learning algorithms [26]. A program that learns how to pass those test cases in T will not necessarily perform well on those not in T. In some applications, the available data are noisy such that y_i is not a perfect representation of desired behavior for the input x_i. In these cases, a program that completely fits the training data would have also learnt the error and therefore it is likely that it will not perform well on unseen data outside of T.

In GP, a tree-based representation is the predominant choice. Each node in the tree is a function whose inputs are the children of that node. A population of programs is maintained at each generation, where individuals are chosen to fill the next population according to their performance as evaluated by a problem-specific fitness function. Often the fitness function rewards the minimization of the error of the programs when run on a training set.

The programs are modified at each generation by evolutionary-inspired search operators, principally a form of crossover and mutation. When programs are evolved with GP, the search operators can break the syntactic constraints of the domain language. To avoid this problem, strongly typed genetic programming [30] employs a set-based type system along with enforced constraints on the search operators. We employ strongly typed GP in this paper, which avoids creating syntactically invalid children to a large extent (see Section VII-C).

One of the main issues of tree-based GP is *bloat* [31], [25], where an increasing growth of the sizes of individual trees is observed with no significant improvement of the fitness values of individuals. Large programs are not only more computationally expensive to evaluate, but are also less able to correctly classify new unseen data (i.e., are vulnerable to over-fitting). Common bloat control techniques are to limit the maximum depth of a tree or to penalize larger trees through a parsimony component in the fitness function [28].

B. Coevolution

In coevolutionary algorithms, one or more populations coevolve, such that the fitness values of individuals within the separate populations are interdependent. There are two types of such relationships: *cooperative coevolution* in which the populations work together to accomplish the same task, and *competitive coevolution* as predators and prey in nature. In the framework presented in this paper we employ competitive coevolution.

Coevolutionary algorithms are subject to the *Red Queen* effect [32]: the fitness value of an individual depends on the interactions with other individuals and therefore the fitness function is not static. For example, exactly the same individual can obtain different fitness values in different generations. One consequence is that it is difficult to assess whether a population is actually "improving" or not [33]–[35]. In fact, there could be *mediocre stable states* [36] in which the coevolution begins a circular behavior in which the fitness values are high at each generation. To try to avoid this problem, *archives* [37]–[39] can be used to store old individuals. The fitness values of the current generations are also dependent upon the old individuals in the archive.

Another issue in competitive coevolution is the *loss of gradient* [34], [40]. If the individuals in one population are either too difficult or too easy to "kill," then the individuals in the other population (assuming for simplicity just two opposing populations) may all be assigned the same fitness value. This can preclude the reward of individuals that are technically better, as interaction with the other population may not be sufficient to expose their superiority.

One of the first applications of competitive coevolutionary algorithms was the work of Hillis on generating sorting networks [41], where he attempted to find a correct sorting network that employed as few comparisons of the elements as possible. He used evolutionary techniques to search the space of sorting networks, where the fitness function was based on a finite set of tests (i.e., sequences of elements to sort): the more tests a network was able to correctly pass, the higher fitness value it received. For the first time, Hillis investigated the idea of coevolving the tests themselves alongside the networks. Coevolving tests was superior to random test case generation as they forced generalization. The experiments of Hillis showed that shorter networks could be found when coevolution was used—enabling him to produce human-competitive solutions.

Ronge and Nordahl [42] used coevolution of genetic programs and test cases to evolve controllers for a simple "robot-like" simulated vehicle. In such work, the test cases are instances of the environment in which the robot moves. Similar work has been successively done by Ashlock *et al.* [43]. In such work, the test cases are instances of the environment in which the robot moves. Given as input a formal specification of a program, Arcuri and Yao [44] used a coevolution of GP and test cases to evolve programs that satisfy the input specification.

C. Multiobjective Optimization

MOO is the process of optimizing more than one objective (fitness) function. A formal definition based on [2] is

$$\begin{array}{lll} \text{Minimize} & f_m(p) & m = 1,2\ldots M \\ \text{subject to} & g_j(p) \geq 0 & j = 1,2\ldots J \\ & h_k(p) = 0 & k = 1,2\ldots K. \end{array}$$

That is, to find a solution p (a program function, in the context of this paper) that optimizes M objective functions and also meets J inequality and K equality constraints. For more details on MOO, please see [2] and [45].

A straightforward method of satisfying multiple objectives is to combine them in a weighted fitness function

$$f(p) = \sum_{m=1}^{M} w_m \cdot f_m(p).$$

However, if objectives are conflicting then we must decide upon the values of the weights $w_1 \ldots w_m$. We must establish the relative importance of particular objectives, and this may not be possible.

In contrast to this weighted sum method, traditional *Pareto-based optimization* seeks to discover a set P of individuals offering differing levels of tradeoffs between objectives, such that they are *Pareto non-dominated*

$$P = \{p \mid \neg\exists p' : (\forall m,\ f_m(p') \leq f_m(p) \wedge \exists n,\ f_n(p') < f_n(p))\}.$$

Thus we now intend to find a set of individuals, where each individual cannot be matched in all objectives and improve upon in at least one. This set represents the possible tradeoffs between objectives.

IV. Problem Formulation

Given an existing program function as input, p_0, and an expected distribution over input values, find an improved function p^* such that

$$\begin{array}{lll} & f_e(p^*) = 0 & \text{for the functional objective } e \\ \text{Minimize} & f_l(p^*) & \text{for each non-functional objective } l. \end{array}$$

Where achieving $f_e(p^*) = 0$ ensures semantic equivalence between p^* and p_0 with respect to a test set T. In this paper, we consider only two objectives: the functional objective f_e and f_{inst}, the instruction count over a fixed set of test cases, but the work can be extended to satisfy multiple non-functional objectives.

V. Framework

We now propose a framework to solve the problem using a combination of evolutionary optimization methods. An overview of the framework is given in Fig. 1. The framework takes as input the code of a function, expressed in the C programming language, along with an expected input distribution. The general nature of the approach means it can easily be applied to other target languages and platforms.

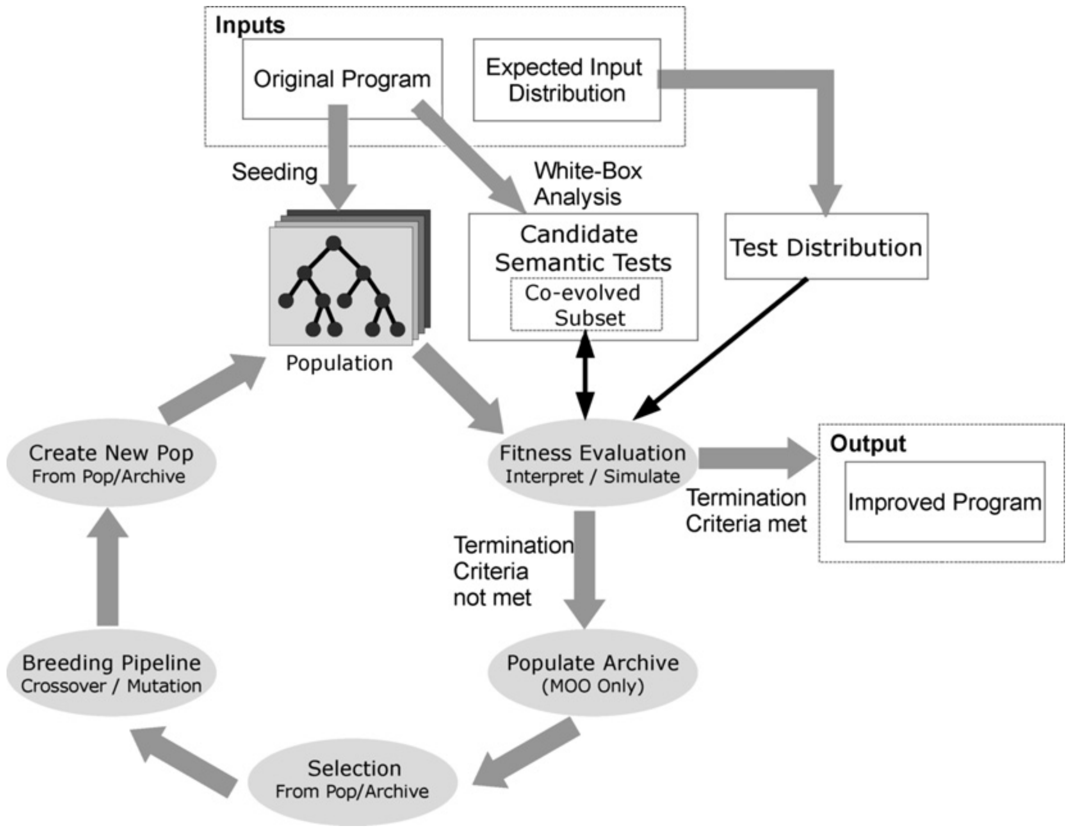
If we take into account the probability distribution of the *usage* of the software (i.e., the operational profile [3]), we can allow for more sophisticated improvements. For example, if a function takes an integer input and if we know that this input will usually be positive, this information could be exploited by optimizing the software for positive input values. Here, we sample integer values according to a simple uniform distribution in $\{-127\ldots128\}$ and array sizes in $\{1\ldots16\}$.

The framework applies GP to optimize one or more non-functional criteria, whilst maintaining semantic equivalence with the original program. This framework is a prototype, and part of the purpose of our experimentation is to assess whether the proposed use of MOO and coevolution is beneficial, in terms of their impact on the ability of the framework to optimize non-functional properties of the software.

The main differences from previous applications of GP are: how the population at the first generation is initialized (seeded), how the training set is generated and subsequently used, the particular rationale for adopting MOO, and the employment of simulation and models in estimating non-functional properties of individuals.

A. Seeding

Evolving faultless software from scratch with GP is a difficult task [46], but part of our interest in this problem is that



Inputs
Original Program
Expected Input Distribution
Seeding
White-Box Analysis
Candidate Semantic Tests
Co-evolved Subset
Test Distribution
Population
Create New Pop From Pop/Archive
Fitness Evaluation Interpret / Simulate
Output
Improved Program
Termination Criteria met
Termination Criteria not met
Breeding Pipeline Crossover / Mutation
Populate Archive (MOO Only)
Selection From Pop/Archive

Fig. 1. Evolutionary framework.

we have as input the source code of the function that we want to optimize, and we can exploit this information when creating our initial population. Invariably, GP systems create an initial population using Koza's established ramped half-and-half method [28], but here we investigate three alternatives based on the concept of using the input program as a starting point.

In designing a different seeding strategy we immediately encounter a classic "exploration versus exploitation" tradeoff that is so often an issue in heuristic search, and in particular evolutionary computation. On one hand, if we over-exploit the original program we might constrain the search to a particular sub-optimal area of the search space, i.e., the resulting programs will be very similar to the input one. On the other hand, ignoring the input genetic material would potentially make the search too difficult.

Although we do not want a final program that is identical to the input one, its genetic material can be used as *building blocks* in evolving a better program.

There are further dangers, as discussed in Poli *et al.* [25]: including many highly fit individuals along with those generated by other means may lead to a lack of diversity in the following generations, and including too few fit individuals may result in the loss of this initial guidance.

We therefore make the seeding strategy the subject of experimental investigation, and in this paper we consider the following types of seeding.

1) *Standard:* Koza's ramped half-and-half initialization method.
2) *Cloning:* a fraction of the initial population will be replaced by a copy of the input function. The remaining individuals are then generated using the *standard* initialization method.
3) *Delta:* a fraction of the initial population will be created by making a copy of the input program and by applying n random mutations to each individual using one of the mutation methods to be employed during the rest of the evolutionary run. If we consider a mutation as a step away from the original program in the search space, then we could use the parameter n to tradeoff exploration and exploitation through selection of an appropriate value for n. The remaining individuals are generated using the *standard* initialization method.
4) *Sub-tree:* a fraction of the initial population will be composed of copies chosen at random from those sub-trees within the input program that have a root node type-compatible with the root node of the input program (recalling that we are using strongly typed GP). The remaining individuals are generated using the *standard* initialization method. This method may be most effective if the building block hypothesis applies to GP: allowing recombination of these subtrees in ways that are more efficient than the original program in terms of the non-functional properties of the final solution.

B. Preserving Semantic Equivalence

Maintaining semantic equivalence is the biggest challenge facing any attempt to solve the problem outlined in Section IV.

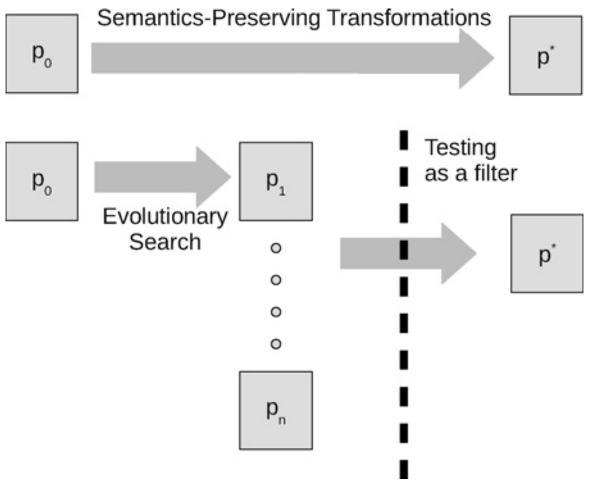


Fig. 2. Optimizing Software by Search and Filter, rather than semantics-preserving transformations.

At this stage, we cannot guarantee semantic equivalence and thus can only recommend the proposed technique as a method of gaining insight into potential optimizations, rather than a fully automated approach to optimization. For example, the output from the framework could be verified by manual inspection. However, the raw optimized output function may be immediately useful in applications that do not have a Boolean measure of acceptable functionality, such as lossy compression or pseudorandom number generation.

In this paper, we recast the problem of software optimization. Rather than assuming we will make only semantics-preserving operations, and then attempting to find the best such operations in order to achieve maximum improvement of a quality metric, we take a dramatically different approach. We try to optimize the software for the quality metric, which is easily done (e.g., if it is execution time then we can simply remove an instruction), and turn the problem into one of mutation testing [47]: can we tell the difference between the two programs? If we fail to find test cases that can discern between the original program and the optimized one, despite a good deal of effort, we have a good degree of confidence that we found a semantically equivalent yet optimized version. We can describe the latter as "Search and Filter" optimization. This is illustrated in Fig. 2.

Within the framework, the "filter" employed is our testing method. It is therefore important that our evaluation of individuals is effective in testing the semantics of new programs against the original. Exhaustive testing is usually impossible, and any testing strategy is open to exploitation by an evolutionary algorithm through over-fitting.

A program P is semantically equivalent to another program P' if and only if for each input that satisfies the precondition of the program they have the same behavior. For example, we can assume that a Merge Sort is semantically equivalent to a Bubble Sort [48], although their implementation is very different. If P has a fault (i.e., it does not confirm to the expected behavior), then a semantically equivalent program P' should manifest the same type of faulty behavior.

Coevolution is employed to improve the effectiveness of program fitness evaluation, such that the set of tests used to evaluate the program population changes at each generation. This application of coevolution is similar to the approach used by Hillis [41]. Prior to evolution, we first generate a large set of test cases that satisfies a branch coverage criterion [49]. Any automated test-case generation technique may be employed, for example [50], [51]. We produce a set of test cases for each branch within the input program, in an attempt to ensure that our testing always encapsulates the semantic notions effectively encoded in the input program.

Fig. 3 illustrates the relationships between the program and test set populations. The group labelled "Pool of Test Cases" are those test cases generated prior to evolution. During evolution, we select only a subset of these test cases at each generation, labelled the "Test Case Population" in Fig. 3. As with the original pool, this coevolved test population is partitioned by the branches of the original program that each test case exercises. This coevolutionary approach has many advantages (see Section III-B).

The computational effort directly depends on how many test cases are used for the fitness evaluation of the GP individuals. We can generate as many test cases as we want, but we cannot run all of them due to time constraints. Once this number of test cases is fixed (e.g., 200 test cases), we still want to have diversity at each generation of the GP process. This is the reason why we sample a larger set of test cases (for example, ten times larger), and at each generation we choose only a subset to use. On one hand, using other strategies such as choosing only a fixed set of "difficult" test cases for the original input program would lead to problems of over-fitting. What is difficult for the input program is not necessarily difficult for the evolving programs, and vice-versa. Furthermore, if we use a seeding strategy in which copies of the input program are introduced in the GP population, then for them all the test cases would pass. On the other hand, using a coevolutionary approach to choose these subsets of test cases has several benefits (see Section III-B), therefore it was a natural first choice to follow. However, other techniques could be considered.

Note that we generate only test cases with valid inputs. Inputs for which the pre-condition of the program is not satisfied are not tested, because for these inputs any output would be valid. For example, for functions that take as input a pointer to an array and a variable representing its length, we only test valid values for the length variable.

At each generation, the GP individuals are tested with the test cases in the training set. The sum of the errors from the expected results is referred to as the *semantic score* and is one component of the fitness of a GP individual. These semantic scores are also used for the fitness value of the test cases. Each program tries to minimize the semantic score on each test case, whereas each test case tries to maximize this score on each program. Fig. 4 shows the relations in the coevolution between evolving programs and test cases.

At each generation, for each subset of test cases we retain only the best half (based on their fitness values). The other half is randomly replaced by test cases in the large pool produced prior to evolution. When we perform these replacements, we ensure each test case in the training set is unique. For

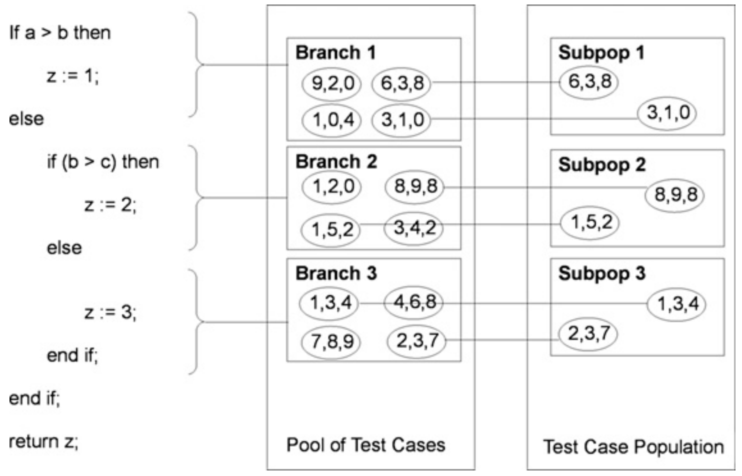


Fig. 3. Relationship between a program and the semantic test set population.

each subset, we store the best individual in a *Hall of Fame* archive [37]. The fitness of the programs is also based on their execution on these old test cases.

When in our empirical study coevolution is not employed, then the test cases are chosen at random at each generation. Their probability distribution is the same used for choosing test cases for evaluating the non-functional properties (i.e., they are randomly chosen based on the operational profile).

C. Evaluating Non-Functional Criteria

In order to search for individuals with some improved level of a non-functional property, we must be able to quantify that property. To do so, we must execute the program on a set of test cases, and here a separate training set from that used to evaluate the semantic score is employed. The set is drawn from the expected input distribution provided to the framework, which could for example be based on probe measurement of software.

The set of non-functional tests is resampled from the expected input distribution at each generation, to prevent overfitting of non-functional fitness for a particular set of inputs.

The final fitness of a potential solution (a GP individual) will be composed of both its semantic scores and this measurement of its non-functional property.

We must be able to reliably (in a repeatable and relatively accurate manner) estimate or measure the property concerned, and in this case we use the number of instructions executed. This is a fairly reliable high-level estimate of execution time. It also reflects the predominantly one-instruction-per-cycle nature of many modern embedded systems such as those using ARM and Alpha ISAs. Embedded systems dominate the computing landscape and place much greater emphasis on non-functional requirements than traditional platforms, hence they are a prime target for the fine-grained optimization we are investigating [52]. More detailed simulation may be desirable when the computation time can be afforded.

In this paper we use both simulation and modeling of instruction count. We use modeling because the goals of this paper require large-scale experimentation that would not be feasible if we used simulation. Similarly, the case studies chosen are fairly small in size to make the large-scale experimentation possible. As a consequence of the small code sizes, native execution using hardware cycle counters would be too noisy to be reliable, as well as too slow. When trying to achieve the best possible optimizations, native execution or simulation would be employed.

Although we are concerned here with execution time, it is worth noting that any property that can be estimated in a similar manner may be used.

1) *Simulation:* The instruction count of an individual can be estimated using a processor simulator and here we have used the M5 Simulator [53], targeted for an Alpha Microprocessor. The parameters of the simulator were left unchanged from their default values, although a few small code changes for convenience and efficiency were applied. The choice of this particular target micro-processor was motivated by the goals of replicability, efficiency, and accuracy. The free availability of the simulator allows others to replicate our paper, and its source code and implicit processor model are available for anyone to review. The Alpha is the most mature target platforms of the M5 simulator, and reflects the type of embedded architectures that we may hope to target with such optimization.

When employing simulation, individuals are written out by the framework as C code and compiled with an Alpha-Targeted GCC cross-compiler. A single function is linked with test code that executes the given test cases, and a total instruction usage estimate provided by a trace file.

We use total instruction count, rather than fine-grained measures such as cycle-count, because it is faster to estimate and a more repeatable measure than cycle-count. Cycle usage is heavily dependent on machine-state, and repeating tests to average out cycle consumption leads to a longer evaluation time. For similar reasons, no operating system is loaded into the simulator prior to testing.

Whilst simulation does not perfectly reflect a physical system, it is worth noting that we are concerned only with *relative accuracy* between individuals, i.e., we wish to *improve* efficiency rather than precisely determine it. Incorrect

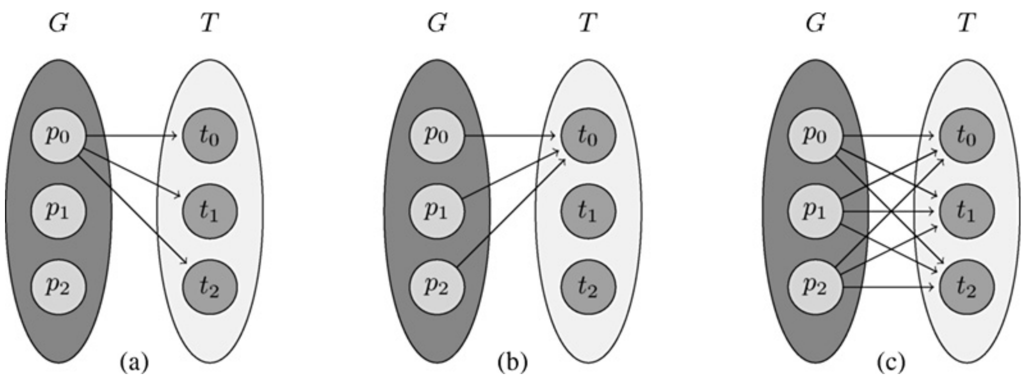


Fig. 4. G is the population of programs, whereas T is the population of test cases. (a) shows the test cases used to calculate the fitness of the first program p_0. (b) shows the programs used to calculate the fitness of the first test case t_0. Note the common arc between the first program and the first test case. Finally, (c) presents all possible $|G| \cdot |T|$ connections.

relative evaluation of two individuals will add noise to the fitness function. The difficulties and intricacies of accurately simulating complex hardware platforms are an issue beyond the scope of our paper: we can easily incorporate alternatives or improvements in both the simulator and compiler.

2) *Model Construction:* In this paper, we have carried out a large number of experiments as part of the analysis of the problem, and compiling and simulating each individual is very computationally expensive in the context of such large-scale experimentation. We therefore opted to study the approach of modeling the instruction usage as a linear model of the high-level primitives executed. This technique could be used as part of a hybrid approach using both modeling and simulation in the future.

To achieve this, we make the assumption that a program's execution time can be estimated by the number of instructions it executes, and that such a predictive model can be expressed linearly

$$Y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \ldots \beta_n x_n + \epsilon$$

where Y is the estimated instruction count of a program, $x_1 \ldots x_n$ are the frequencies with which each of the n instructions are executed within a program, and the coefficients $\beta_1 \ldots \beta_n$ are an estimate of the number of machine-level instructions each higher-level instruction will create. The intercept is given by β_0 and ϵ is the noise term, introduced by factors not considered by the other components of the model. We acknowledge that this is a simplification, because subsequent compiler optimizations will be dependent on the program structure and there is not a simple mapping between high level source code and low level instruction execution.

To use such a model, the coefficients must be estimated. We achieved this by executing one large evolutionary run of the framework for each case study, and separately evaluating each program through interpretation to record the frequencies with which each high-level primitive (`if`, `loop`, array access, and so on) is executed and logging the corresponding instruction count of an individual as measured by the simulator. Least squares linear regression is then used to fit this model. It is possible to verify the relative accuracy of this model for the data points used in constructing it, as detailed in the results on this part of our paper in Section IX-C.

D. Multiobjective Optimization

In this paper, we consider two objectives and use both weighted sum and Pareto-based MOO. The two objectives are:

1) minimize error across test cases, as in Section III-A;
2) minimize the number of instructions executed.

We are concerned primarily with minimization of error, and second with reducing execution count. Ultimately, we are not concerned with individuals that do not meet their functional specification as estimated by the test set, and so it is possible to combine our objectives with a weighted sum method, as described in Section III-C.

By normalizing the fitness values and setting the weighting of the functional fitness to be greater than the weighting given to the instruction count, we can create an arbitrarily sized bias toward favoring improvement in functional performance over the instruction count of a solution.

As well as employing weighted sum methods, we also aim to use MOO methods in an unusual application. The difference between the goal of our optimization against a traditional Pareto-based method is illustrated by Fig. 5. Given the input program at the bottom-right of the figure, we are ultimately interested only in locating the desired output program that lies on the x-axis as close to the origin as possible.

However, we still employ traditional MOO techniques within this paper: this is because we are interested in the balance between exploration and exploitation. A MOO approach explicitly forces the search to discover and retain solutions in Fig. 5 that are *not on the x*-axis, that is those solutions we may actually regard as *inferior*. It is part of our goal here to identify if employing MOO allows the search to locate smaller, instruction-efficient subtrees of a program that may be reassembled by the search algorithm into improved solutions later in the execution of an evolutionary run. This proposal and its effectiveness are linked to the building block hypothesis and GP Schema theory: readers are encouraged to consult [54] for further details.

We therefore adopted two approaches to combine objectives in fitness evaluation. The first was to use a simple linear combination of the functional and non-functional fitness measures. The second is to use the strength Pareto evolutionary algorithm two (SPEA2) [55], a popular Pareto-based method that

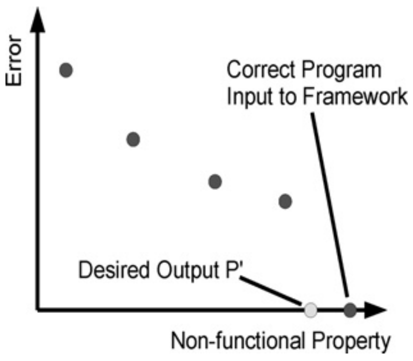


Fig. 5. Pareto front composed of five programs in objective space.

attempts to approximate the Pareto-front in objective space. The choice of which to use is the subject of experimental evaluation in Section VIII-A.

Any advantage we may find by using a Pareto-based method has interesting implications for understanding how GP achieves its goal: *can building blocks be recombined in different ways to improve performance?*

Note that in some experiments we will therefore be using both MOO and coevolution of test-cases. Coevolution effectively creates a "moving target" for optimization. Thus, the dominance of an individual, its relation to other solutions in a tradeoff space, will change with this moving target. Thus, an individual considered Pareto non-dominated in one generation may not be considered so in the next. This is not an unusual situation, as it may occur when employing other types of test case sampling alongside MOO, or when using a noisy fitness function. This raises an important question as to the tolerance of multiobjective optimization algorithms to variability in Pareto non-dominance, such that it may be the case that MOO without coevolution is actually more effective.

E. Fitness Function

Given a population of programs G, we use two sets of test cases T_0 and T_1. The fitness function of the programs in G is based on their execution on these test case sets. The set T_0 is used to evaluate the semantic score of the programs. It is this set that is coevolved at each generation. Note that although T_0 is partitioned in different subsets and an archive, we are considering their union. The set T_1 is used to assess the non-functional value of the programs, i.e., the instruction count in our case.

Given $s(p,t)$ a function to calculate the semantic score of a program $p \in G$ run on a test $t \in T_0$, and given $c(p,t)$ another function to calculate its instruction count, the fitness function $f(p,T_0,T_1)$ for the programs to minimize is

$$f(p,T_0,T_1) = \alpha \mathbf{n} \left(\sum_{t \in T_0} s(p,t) \right) + \beta \mathbf{n} \left(\sum_{t \in T_1} c(p,t) \right)$$

where $\mathbf{n}$ is any normalizing function in [0,1], in particular we used $\mathbf{n}(x) = x/(x+1)$, as suggested in [56]. α and β are constants to give different emphasis on each objective. We use the arbitrary values $\alpha = 128$ and $\beta = 1$ to give more emphasis on the semantic score, see [31]. Note that we are not penalizing bloat directly, because the instruction count already does it. In the cases in which a program is either not compilable or it has runtime errors (see Section VII), then we apply a death penalty, i.e., its fitness value would be $\alpha + \beta$.

When MOO is used, these two objectives (normalized sum of the semantic score and cycle score) are treated separately.

The fitness function $f(t,G)$ for the test cases in T_0 to maximize is

$$f(t,G) = \sum_{p \in G} s(p,t) \ .$$

There can be different ways to define the function $s(p,t)$. A simple way would be

$$s(p,t) = \begin{cases} 0, & \text{if } t \text{ is passed} \\ 1, & \text{otherwise.} \end{cases}$$

However, we can try to use some heuristics to give more gradient to that binary function $s(g,t)$. For example, in this paper when the programs return an integer value y_g and that is compared in the test case against the expected value y_t^*, then we can use

$$s(p,t) = |y_p - y_t^*| \ .$$

This however would not be particularly useful if the integer output is actually encoding an enumerated type, as for example the classification of the triangles in the program in Fig. 11 in Section IX-A.

In more complex cases (e.g., comparisons of arrays as for example in a sorting routine), the heuristic depends on the state of the memory after the computation. For example, in our case study for the sorting algorithms we use test cases that have assertions on the equality of each compared cell of the arrays. In other words, each value in the modified array A' (after the computation is finished) is compared against the value in the same location in the input array A. The heuristic we use is simply adding up these errors

$$s(p,t) = \sum_{i=0}^{length-1} |A'[i] - A[i]|.$$

Depending on the type of outputs, different heuristics can be designed. However, a complete analysis of this problem is not in the scope of this paper. We simply define heuristics for the types of outputs we have in our case study.

VI. Case Studies

In our experiments, we consider eight different C functions. Table VI summarizes their properties and their source code listings are given in Section IX-A.

When selecting case studies, we were interested in finding functions that had been studied for their execution time, represented a variety of program structures, used simple data types in order to greatly simplify our coevolutionary testing, component, and from a source readily available and well-documented.

We use two different implementations of the triangle classification program published in [50] and [57], and two different implementations of a Bubble-Sort algorithm [48].

TABLE I
SUMMARY OF THE PROGRAMS USED IN THE CASE STUDY

Name	LOC	GP Nodes	Input	LV
Triangle1	35	107	int, int, int	1
Triangle2	38	175	int, int, int	1
Sort1	11	63	int[], int	3
Sort2	18	69	int[], int	4
Factorial	7	16	int	0
Remainder	40	208	int, int	3
Swi10	22	68	int	1
Select	94	392	int[], int, int	9

The lines of code (LOC) and the number of GP nodes in their tree representation are displayed. It is also specified what type of input these functions take and the number of local variables (LV).

We also consider a recursive implementation of the Factorial function [48]. From [58] we use the Remainder routine. Finally, from a library of worst-case execution benchmarks [59], we consider the Swi10 and Select (returning the *k*th order statistic) functions.

The programs include a variety of structures: branching and nesting, loops over arrays, internal state altered multiple times within a function, switch and case statements, use of both temporary variables and arrays, and recursive calls. Two pairs of functions also solve the same problem, which provides for an interesting comparison of their optimization.

The program functions used in our case study are fairly small, with the largest containing only around 90 lines of code, which is advantageous in allowing us to perform large-scale experimentation. The solving of industrial-scale problems can require a lot of computational effort [29] and a study of the scalability of our approach is important, a matter of future investigation and likely to reflect the scalability of the underlying search algorithm, GP.

There could be several approaches to address larger programs. If a program is composed of two or more functions, then each function can be independently optimized by our framework, one at a time. If a function is very long, then it could be divided into subdivisions of code of manageable size. Our framework would try to optimize only these subdivisions, one at the time. It will be matter of future work to analyze whether it would still be possible in this context to obtain types of optimizations that current compilers cannot produce. Note that, in our empirical study, for each problem we only used 50 000 fitness evaluations. For larger software, more computational resources (i.e., larger GP populations and more generations) would be necessary.

VII. IMPLEMENTATION

A. Introduction

In this section, we describe the implementation of the framework, including parameter values that were not the subject of experimentation. We also discuss potential pitfalls in evaluating programs in a language such as C, which are often hidden from the practitioner when manipulating programs in an abstract language and executing them through interpretation. For example, consider a protected-division primitive commonly used in the GP literature: implementing this in C would require a macro to check for a division-by-zero, which can be inefficient. One advantage of using a simulator is that the evolved programs are demonstrably compilable and can be declared free of run-time errors for those inputs tested.

```
int Factorial(int a)
{
  if (a <= 0)
     return 1;
  else
     return (a * Factorial(a-1));
}
```

Fig. 6. Source code of factorial.

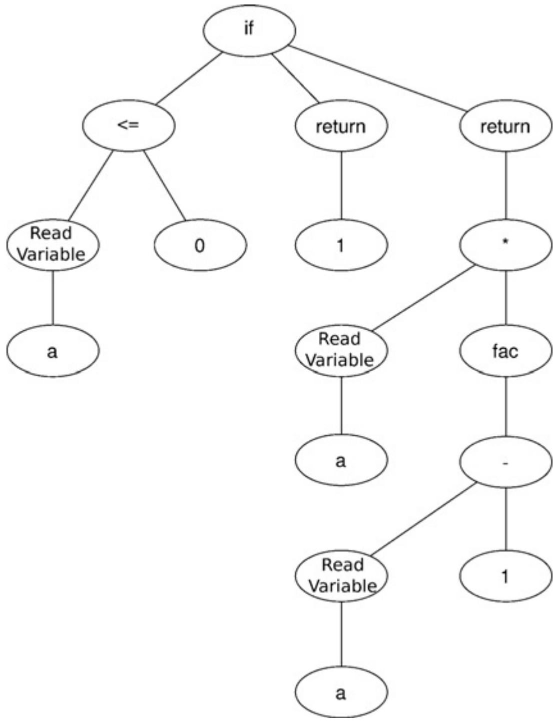


Fig. 7. GP representation of factorial.

B. Framework Implementation

The framework was implemented in Java, and we used ECJ 18 [60] for the GP system. We use strongly typed GP [30] to assign return and argument types to each primitive in the function set. This assures that syntactically invalid trees are not created when applying crossover and mutation.

The primitives used within the GP algorithm are listed in Table II. This is the superset of all primitives required to represent each of the chosen case studies as an S-Expression. Consequently, we must search a large space of possible trees, and the extensive (necessary) use of strongly typed GP results in a representation that does not easily lead to valid programs when carrying out crossover or mutation.

Fig. 6 shows the implementation of a Factorial function, whereas its GP representation is depicted in Fig. 7. The translation from Java code to the GP representation is done automatically. To parse the Java code, we use the program JavaCC [61].

ECJ parameters not detailed here or specified in Section VIII were left to the defaults as inherited from `koza.params`,

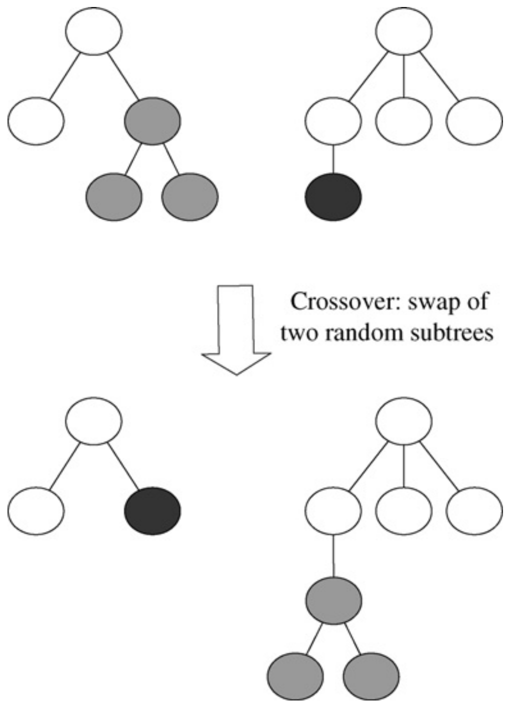


Fig. 8. Example of crossover.

provided with the distribution. The default method of initializing the population is to use Koza's ramped half-and-half method, with a minimum depth of 2 and a maximum depth of 6 for the half method, minimum and maximum 5 for the grow method.

We use three methods to populate the next generation: mutation, crossover and reproduction. The probability of reproduction is fixed at 0.1, whereas the balance between the remaining operators is the subject of experimentation. We use all six methods of mutation ECJ provides with equal probability. Each of the three ways of generating the next population selects individuals from the current population using standard tournament selection. When selecting a terminal or non-terminal for crossover or mutation, we do so with probability 0.1 and 0.9, respectively.

Mutation is used to copy the parents with a single change in their trees to generate slightly different offspring. In case of a mutation event, one of the following different mutations provided by ECJ is randomly chosen with a uniform probability, where n is a randomly chosen node in the program tree. Note that the following is just a brief description, for more details please see [60].

1) Point mutation: the sub-tree rooted at n is replaced with a new random sub-tree with depth 5.
2) OneNode mutation: n is replaced by a random node with the same constraints and arity.
3) AllNodes mutation: each node of the sub-tree of n is randomly replaced with a new node, but with the same type constraints and arity.
4) Demote mutation: a new node m is inserted between n and the parent of n. Hence, n becomes a child of m. The other children of m will be random terminals.

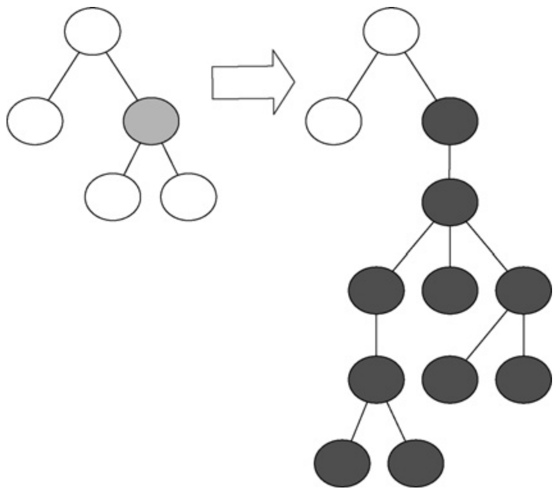


Fig. 9. Example of point mutation.

5) Promote mutation: the sub-tree rooted at the parent of n will be replaced by the sub-tree rooted in n.
6) Swap mutation: two children of n are randomly chosen. The sub-trees rooted at these two nodes are swapped.

Crossover of two individuals consists of choosing one random subtree in each of them (with the constraint that their roots should be compatible), and then these two subtrees are swapped. Note that in our case neither crossover nor mutation generates syntactically incorrect offspring.

These types of crossover and mutations are the typical ones used in the literature, for more details see [25]. For the sake of clarity, Fig. 8 shows an example of the application of the employed crossover, whereas Fig. 9 depicts the application of a point mutation.

For each problem, we generated a set of potential tests containing 2000 individual test cases. The set fulfils the branch coverage criterion for the input function and as such could be generated with any automated software testing technique prior to the execution of our framework, although in this paper we manually wrote specific scripts to generate these test cases. The test case population size is 200, with a further archive of 50 elements. The test case population is partitioned in a number of subsets that depends on the number of branches in the original input program.

The cycle score is evaluated on 100 test cases that are randomly sampled at each generation. Integer variables are chosen according to a uniform distribution of values in $\{-127, \ldots, 128\}$. The length of the arrays is uniformly chosen in $\{1, \ldots, 16\}$. We are interested in improving programs when they are called with valid inputs, and so we only assess its non-functional properties using valid input data. Thus, where a length is supplied to a particular case study function, we give the correct value corresponding to the current input array.

With the inclusion of loops and recursion in the primitive set, we must limit the number of iterations that may occur: we set this figure to an arbitrary value of 1000, which is large enough for the case study functions to produce the correct output on all possible inputs without reaching the limit.

TABLE II
PRIMITIVES GROUPED BY TYPE

Type	Name	Number	Description
Arithmetic	+, −, ∗, /, %	5	Typical arithmetic operators. % is the module operator.
Unary modification	++, −−	2	++ is the post unary increment, whereas −− is the post unary decrement.
Boolean	&&, ||, !, >, ≥, ==, <, ≤	10	Typical operators to handle Boolean predicates.
Constant	`true`, `false`, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9	12	Boolean and 10 integer constants.
Statement	`for`, `while`, `if`, `switch`, `case`, `return`, `skip`, `statement_sequence`, `case_sequence`	9	Typical statements. `skip` is the empty statement. `statement_sequence` is used to hold two child statements to create statement sequences, and `case_sequence` is used in a similar manner to link the cases of a switch statement.
Variable	`ReadVariable`, `WriteVariable`, `VariableWrapper`, `V_tmp`	≥ 4	Primitives to read and write variables. `VariableWrapper` is a single child node used to simplify the GP breeding operators. `V_tmp` is an integer variable. We also supply primitives representing the inputs and the local variables specific to each individual case study (see Table VI).
Array	`ReadArray`, `WriteArray`, `ArrayWrapper`	3	Primitives to read and write array variables. `ArrayWrapper` is used to simplify the GP breeding operators. These three primitives are used only if the program under test manipulates an array.
Special	`fac`	1	Miscellaneous primitives. *fac* is only used in the Factorial problem to call the function recursively.

C. Non-Compilable GP Individuals

The type constraints on the GP primitives are of a local nature, specifying the type of each node and the type of parent and children they can have. This is sufficient for most GP applications, but it is not sufficient when modeling a higher level language such as C. In order to evaluate individuals through simulation, we must first translate them to C. It is possible that valid GP trees would generate compilation errors once translated. For example, we encountered two such problems during our experimentation, both with the `switch` statement.

1) The indexes of a case within a `switch` statement are not constant, i.e., they cannot be calculated at compile time.
2) The indexes of the cases within a `switch` statement are not all unique, i.e., at least one index is used more than once.

Instead of complicating the constraint system to forbid the generation of non-compilable individuals, we allow them and punish any such individuals. Through checking for these two cases, we are able to identify invalid individuals and penalize their fitness by setting it to the worst possible value. Compilation and simulation of such an individual is subsequently suppressed.

D. Run-Time Errors

Using such a rich subset of the C language in our representation can lead to program errors discovered at run-time. For example, an array may be accessed out of bounds, corrupting memory. This can lead to unpredictable behavior and our system must be robust with respect to such eventualities. We elected to follow a similar policy to that of uncompilable programs: the individual is flagged as not having completed every test case and punished via their fitness value.

E. Exceeding the Iteration Limit

Long or infinite iteration is inefficient, costly or impossible to evaluate. The M5 simulator provides support for an upper limit on simulated cycles, and we use this feature to enforce an execution "timeout" on each individual. The timeout is set to a value twice that of the original input program's instruction count on a representative sample of fitness cases.

VIII. Experimental Method

A. Overview

Given our proposed framework, there were several issues we wished to investigate: first, was it possible to optimize our case studies using the framework? What kinds of optimization is it capable of producing? These questions are potentially answerable with a single execution of the framework for each case study, or a set of repetitions to ensure repeatability. The remaining questions required extensive experimentation. These were as follows.

1) How well can we model the instruction count of an individual without executing the simulator?
2) Which of the components and parameters of our framework are important in determining the level of improvement that can be achieved?
3) What is the impact of using a model rather than the simulator to estimate instruction count?
4) How does the use of MOO affect the amount of exploration?

B. Method

Our experimentation consists of the following steps, each one designed to answer the corresponding numbered question in the previous section.

1) Produce a model estimating instruction usage for each problem and evaluate those models using resampling to simulate tournament selection.

TABLE III
EXPERIMENTAL PARAMETERS

Parameter	Description	Low	High
x_1	Probability of crossover	0.1	0.8
x_2	Population size	50	1000
x_3	Tournament size	2	7
x_4	Seeding proportion	0.1	0.9
x_5	Coevolution enabled?	FALSE	TRUE
x_6	SPEA2 enabled	FALSE	TRUE
x_7	SPEA2 archive size	$0.1x_2$	$0.9x_2$
x_8	Seeding method	Clone, mutation, or subtree	
Dependent Parameters			
x_9	Probability of mutation	$0.9 - x_1$	
x_{10}	Generations	$50\,000 / x_2$	

2) Carry out a factorial experiment and further comparisons to evaluate the importance of components of our framework.
3) Select parameter settings that performed well in the factorial experiments, and use them to compare simulation to using a model of instruction usage.
4) Collect and analyze data on the impact of using MOO on the exploration of the search space.

C. Factorial Experimentation

In order to determine which components were important in affecting the performance of our framework, we carried out a full factorial experiment, a robust approach that has previously been demonstrated to be effective when using GP [62]. After initial experimentation we identified key parameters that appeared to affect the performance of the algorithm most, and these were chosen for experimentation. Their levels are given in Table III. As well as components of our framework, we also included general GP parameters we considered to be of large importance, so we could allow for their impact on the behavior of other parameter settings.

With 7 parameters at 2 levels and one at 3 levels, we had $2^7 * 3 = 384$ design points per problem. We carried out 30 repetitions at each design point to allow for variation in the response caused by the random seed. Thus a total of 11 520 design points per problem, 92 160 experimental runs in total for this part of our experimentation.

D. Response Measure

At the end of an evolutionary run, we select the best individual from the final population as defined by the weighted sum given in Section III-C as our response. This individual is then tested on new data to evaluate its functional correctness, and is run through the simulator to estimate its non-functional fitness. We use a separate set of 1000 test cases that have not been used in the coevolution to validate the final output.

This does not guarantee that a program is truly semantically equivalent, but it does test the individual against tests independent of, for example, any coevolved test set.

IX. RESULTS

A. Example Optimizations

It is of immediate interest as to *how* optimizations were successfully made by the framework, and here we report a

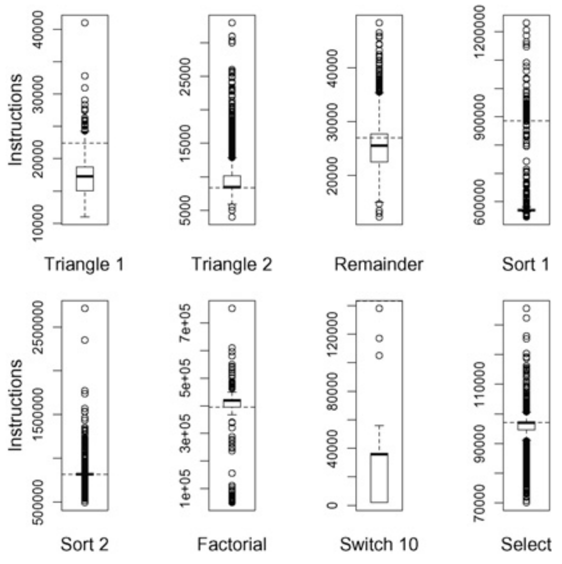


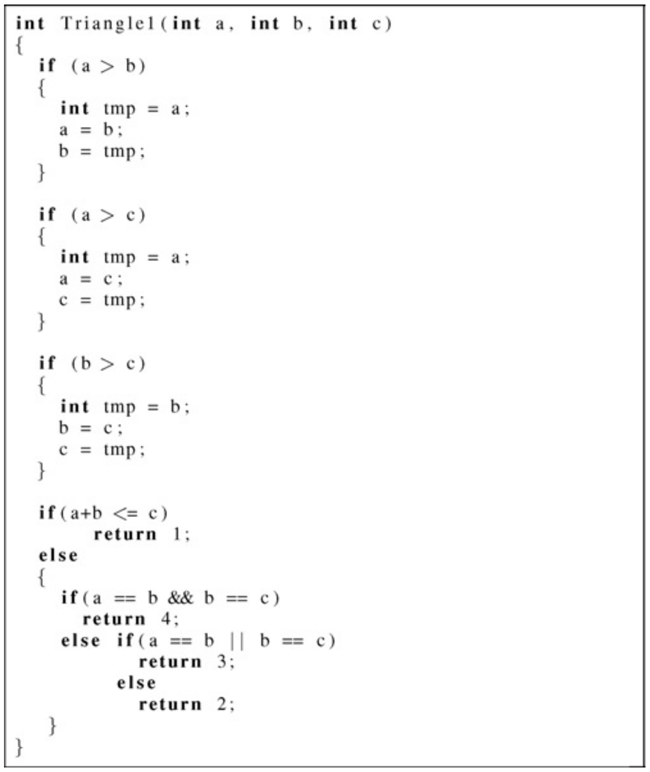
Fig. 10. Boxplots illustrating the distribution of instruction counts for valid program outputs. The dashed lines indicate the performance of the original programs.

selection of optimizations that we came across and found interesting. Note that there are thousands of unique output programs, and those we have looked at are only a small selection. Perhaps some form of data mining could be used to extract commonly produced optimizations.

Boxplots of the instruction counts across design points that generated solutions passing the final test set for each problem are given in Fig. 10. The variance of the instruction count of optimized programs is quite low for each problem, which demonstrates how robust the technique is to parameter changes. In some cases it is necessary to examine the outliers to see real improvements in instruction count.

1) *Triangle1:* In the implementation of Triangle1 (see Fig. 11), the three inputs are ordered in the three variables (a,b,c). There are three `if` statements controlling three swap operations to achieve this ordering. In a valid triangle, the sum of the two shortest edges should be greater than the length of the third edge. To see whether these edge lengths represent a valid triangle, there is the check $a + b \leq c$. But it is not important to guarantee $a \leq b$, we just want that a and b represent the shortest edges. Therefore, a true order of the three variables is not necessary. GP is able to learn this property. In the evolved programs, most of the times GP removes the first `if` statement with its swap block. This does not change the semantics of the program. Notice that this is a very easy optimization that compilers such as GCC are not able to make.

In a swap operation, a temporary variable is employed and there are three assignments. After the three `if` statements with the three swaps at the beginning of the code, that temporary variable is not used anymore. GP learns to use only two assignments for the last swap and then to use the temporary variable in the remaining of the code instead of the variable c. In other words, in the last swap operation the command $c = tmp$; is removed, and each successive occurrence of the variable c is replaced by tmp.



```
int Triangle1(int a, int b, int c)
{
  if (a > b)
  {
    int tmp = a;
    a = b;
    b = tmp;
  }

  if (a > c)
  {
    int tmp = a;
    a = c;
    c = tmp;
  }

  if (b > c)
  {
    int tmp = b;
    b = c;
    c = tmp;
  }

  if(a+b <= c)
       return 1;
  else
  {
    if(a == b && b == c)
      return 4;
    else if(a == b || b == c)
           return 3;
         else
           return 2;
   }
}
```

Fig. 11. Source code of Triangle1.

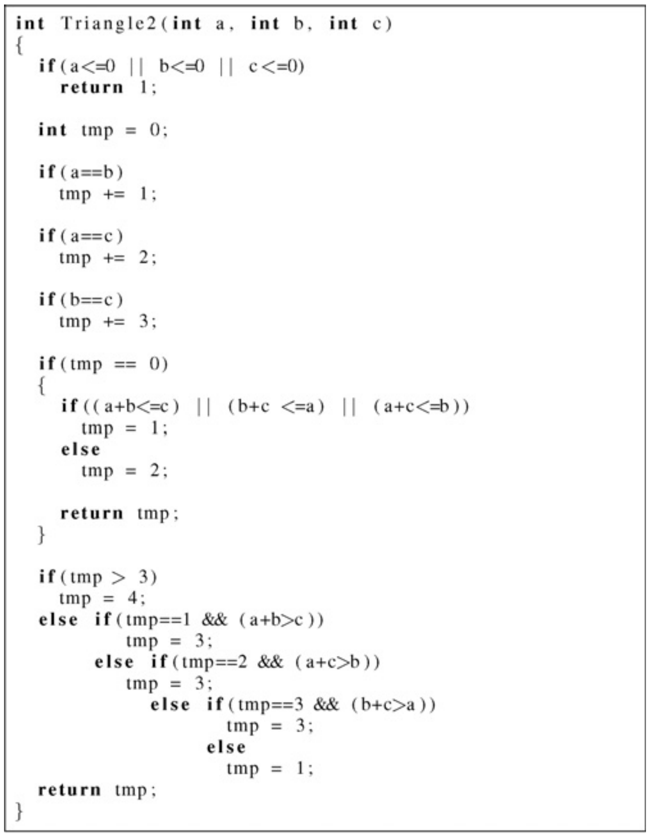
The last optimization we found that GP is able to do is quite surprising. The following code structure has been evolved:

```
if(a > c) {/* swap a and c */}

if(b > c) {/* swap b and c,
                   and use tmp instead of c */}
else {/* check type of triangle as in the
              original code and return */}

if(a+b <= tmp) {return 1;}
else if (a == b) { return 3;}
else {return 2;}
```

To make the last swap, the predicate $b > c$ needs to be evaluated. If that is true, than the triangle cannot be equilateral. Hence, GP learns to move the code that checks the type of triangle directly in the `else` branch of that swap. Then it replicates that code after that `if` statement to handle the case $b > c$. But in this latter case, the triangle not only cannot be equilateral, but also to check whether it is isosceles we only need to evaluate $a == b$. So the code is significantly reduced. This is a non-trivial optimization because code is replicated and reduced based on exploitation of the properties resulting on the evaluation of the predicates in the branching statements.

2) *Triangle2:* In our experiments, GP was able to evolve faster versions for the program Triangle2, given in Fig. 12. Our analysis of the resulting GP trees did not find any interesting optimization. This because GP learnt a *undesired pattern* in the test script we used to generate the test cases. This pattern is easy to exploit at the beginning of the code. So, any further optimization of the code cannot be rewarded because it does not provide any improvement.

We found this property only once all the experiments were finished and we analyzed the results. Considering the large amount of computational time required for the empirical analysis, we decided to not change the script generator and re-run all the experiments. Nevertheless, this case study is still important to stress the role that the test cases play in the coevolution. If there is an easy way to cheat, then very likely GP is going to find this way (as it has done in this case study).



```
int Triangle2(int a, int b, int c)
{
  if(a<=0 || b<=0 || c<=0)
    return 1;

  int tmp = 0;

  if(a==b)
    tmp += 1;

  if(a==c)
    tmp += 2;

  if(b==c)
    tmp += 3;

  if(tmp == 0)
  {
    if((a+b<=c) || (b+c <=a) || (a+c<=b))
      tmp = 1;
    else
      tmp = 2;

    return tmp;
  }

  if(tmp > 3)
    tmp = 4;
  else if(tmp==1 && (a+b>c))
        tmp = 3;
      else if(tmp==2 && (a+c>b))
        tmp = 3;
          else if(tmp==3 && (b+c>a))
                tmp = 3;
              else
                tmp = 1;
  return tmp;
}
```

Fig. 12. Source code of Triangle2.

3) *Sort1:* Sort1 implements a naive bubblesort. The best solutions found by the framework improved on the original by omitting a single iteration, and using an input variable as a loop counter, rather than using a new variable.

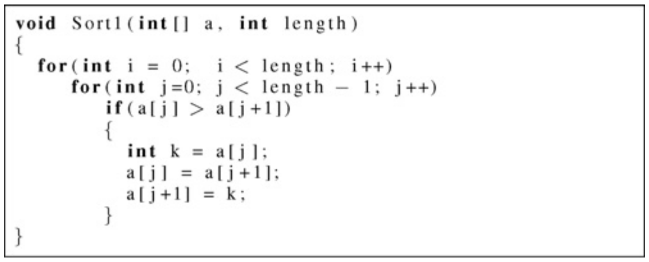
The original code was as given in Fig. 13.

The optimized output was

```
void sort(int* a, int length) {
    for (; 0 < (length - 1); length--) {
        for (int j = 0; j < (length - 1); j++) {
            if (a[j] > a[1 + j]) {
                k = a[j];
                a[j] = a[j + 1];
                a[1 + j] = k;
            }
        }
    }
}
```

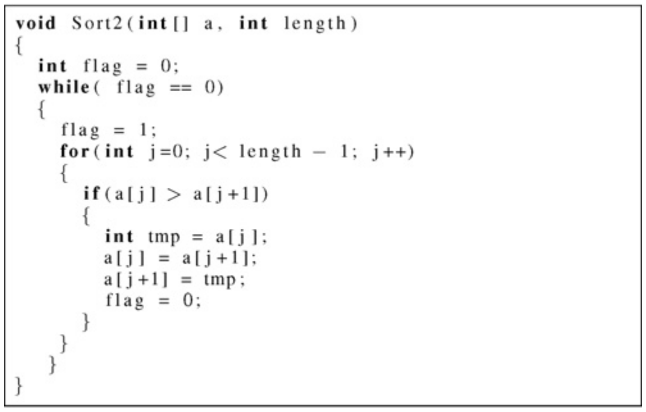
These are sensible optimizations, but perhaps it is a little disappointing that further optimizations were not discovered, such as the use of a *sorted* flag to halt the algorithm once a pass has been completed with a swap. Such an optimization was given as input to the next problem.

4) *Sort2:* Some of the optimizations were similar to those found in Sort1, such as removing the initialization of a loop



```
void Sort1(int[] a, int length)
{
  for(int i = 0;  i < length; i++)
     for(int j=0; j < length - 1; j++)
        if(a[j] > a[j+1])
        {
          int k = a[j];
          a[j] = a[j+1];
          a[j+1] = k;
        }
}
```

Fig. 13. Source code of Sort1.



```
void Sort2(int[] a, int length)
{
  int flag = 0;
  while( flag == 0)
  {
    flag = 1;
    for(int j=0; j< length - 1; j++)
    {
      if(a[j] > a[j+1])
      {
        int tmp = a[j];
        a[j] = a[j+1];
        a[j+1] = tmp;
        flag = 0;
      }
    }
  }
}
```

Fig. 14. Source code of Sort2.

counter and using input variables as counters rather than separate individual variables.

The original input is given in Fig. 14. One simple optimization the system found was to "inline" the first pass of the sort algorithm

```
for (V_j = 0; V_j < (V_length - 1); V_j++) {
  <main body>
}

while (V_flag == 0) {
  for (V_j = 0; V_j < (V_length - 1); V_j++) {
    <main body>
  }
}
```

This saves on a single comparison-and-branch, which over 1000 test cases is certainly worthwhile. The system also discovered multiple variants of

```
void sort(int* a, int length) {
    int flag=false;
    int tmp = 0;
    int i=0;
    while (0 == flag) {
        flag = 1;
        for (j = 0; j < (length - 1); j++) {
            if (a[V_j] > a[j + 1]) {
                tmp = a[j];
                a[j] = a[j + 1];
                a[j + 1] = tmp;
                flag = 0;
            }
        length--;
     }
}
```

Note the decrement of *length*. Experienced human programmers may also make this optimization.

5) *Fac:* Fac is a small function (see Fig. 6), but one optimization made by the GP system is to change

```
if (V_a <= 0) {
```

to

```
if (V_a <= 1) {
```

This saves one recursive call for test cases containing positive inputs.

A more interesting optimization is the exploitation of overflow behavior in Java. We tested individuals in Java for the most part, because we were interested in large-scale experimentation and interpretation rather than simulation of individuals is more efficient. Thus, the system ensured semantic equivalence with the Java version, whilst targeting a model of execution time on an embedded processor. This can lead to unforeseen issues if the Java interpreter does not match the semantics of compiled C. We eliminated nearly all such cases, but we were not able to prevent the system from exploiting one such behavior when optimizing the Factorial function.

Observing some of the most efficient individuals, it quickly became apparent that evolution had "decided" it was beneficial to add conditional statements such as the following:

```
if (!((2 * (9 + 8)) <= V_a))
while (V_a < (7 * (1 + 4))) {
if (V_a <= (8 + (8 + (8 + 9)))) {
```

Following this statement would be the usual Factorial function. Each constant being compared to the input is 33, 34, or 35. Clearly, not calculating fac(33) or fac(34) was an optimization in terms of timing—but surely this would break the semantics of the program?

However, fac(34) = 0 according to the original program. An overflow results in a zero return. Therefore, for any value $n \geq 34$, we have fac(n) = 0 because a multiplication by fac(34) = 0 will occur during the computation of fac(n). The GP system correctly discovered that cycles were being wasted in these cases, and by adding this `if` statement, it enabled the code to fall through to the default "return 0" at the end of our test harness.

The factorial of numbers increases very rapidly, and already for small input values their factorial cannot be stored in a 32 bit integer variable. When we ran our experiments, we did not consider this case, so we did not specify any precondition for the factorial function. Wrong outputs due to arithmetic errors were hence considered as valid output. Therefore, the evolving GP individuals try to behave on these inputs in the same way as the original program depicted in Fig. 6. Although the value 0 is not the right factorial of the input 34, the *actual* semantics of the input program in Fig. 6 are preserved. Our framework takes as input only the code of the program we want to optimize. Without any further information, it is impossible to evolve a program that satisfies the *intended* behavior of the input program if this latter one is faulty (or if its precondition is not specified). Notice that optimizations made by compilers would still result in programs that give 0 as output if the input is bigger or equal than 34.

6) *Remainder:* The source of the Remainder function is given in Fig. 15. Most of the optimizations of this (somewhat

```
int Remainder(int a, int b)
{
  int r  = -1;
  int cy = 0;
  int ny = 0;

  if (a==0);
  else if (b==0);
       else if(a>0)
              if(b>0)
                 while((a-ny)>=b)
                 {
                   ny=ny+b;
                   r=a-ny;
                   cy=cy+1;
                 }
              else // b<0
                 while((a+ny)>= ((b>=0) ? b : -b))
                 {
                   ny=ny+b;
                   r=a+ny;
                   cy=cy-1;
                 }
            else  // a<0
              if(b>0)
                while( ((a+ny)>=0 ? (a+ny) : -(a+ny)) >= b)
                {
                  ny=ny+b;
                  r=a+ny;
                  cy=cy-1;
                }
              else
                while(b>=(a-ny))
                {
                  ny=ny+b;
                  r= ((a-ny)>=0 ? (a-ny) : -(a-ny));
                  cy=cy+1;
                }
  return r;
}
```

Fig. 15. Source code of Remainder.

inefficient) code simply use the % modulo operator, as we would expect. One interesting optimization is the removal of the first `if` statement, which checks for a zero value of *a*, and directly returns 0 for that value. But removing that check does not change the semantics (i.e., the return value will still be 0). This is an interesting optimization, because it exploits the expected input distribution. This code is only useful when the input *a* is zero, which is not a common occurrence as its input domain is [−127, 128]. Had our input domain been (for example) [−1, 2], then the optimization would not have been worthwhile. We can imagine that this simple method of optimization might be applied to many other programs.

7) *Switch 10:* Switch 10 (see Fig. 16) was chosen deliberately to see if our system could optimize it in the obvious way. Indeed, we found that the system was able to find the following:

```
return 10 + V_a;
```

A minimal solution, containing just this code, was found multiple times by the system. This is a non-trivial optimization that GCC –O2 was unable to achieve, a satisfying result.

8) *Select:* Select, given in Fig. 17 is the largest and arguably the most complicated function. Many of the output optimized programs were large, and this made it time-consuming to analyze them. Perhaps the introduction of a sophisticated parsimony method such as that proposed by Poli *et al.* [63] may improve the readability of the outputs.

Nevertheless, there are some common trends to the optimizations. The first optimization is a check for negative values

```
int Swi10(int a)
{
  for (int i=0; i<10; i++)
  {
    switch (i)
    {
      case 0: a++; break;
      case 1: a++; break;
      case 2: a++; break;
      case 3: a++; break;
      case 4: a++; break;
      case 5: a++; break;
      case 6: a++; break;
      case 7: a++; break;
      case 8: a++; break;
      case 9: a++; break;
      default: a--; break;
    };
  }

  return a;
}
```

Fig. 16. Source code of Swi10.

```
if (V_a >= V_k) {
  V_flag = 1;
}
```

By placing this at the start of the program, no iterations are made and the function quickly returns.

The second optimization is that the conditional statement in the `else` part of the main `if` statement can be removed

```
else if ( flag == 0) {
```

This is replaced with

```
else {
```

This can be done because `flag` is tested in the `while` statement previously.

The original solution is also slightly inefficient in one of its exchanges, the system found this issue and removed the inefficiency. Observe

```
temp=(arr[mid]);(arr[mid])=(arr[l+1]);(arr[l+1])=temp;

if (arr[l+1] > arr[ir]) {
temp=(arr[l+1]);(arr[l+1])=(arr[ir]);(arr[ir])=temp;
}
```

The first assignment of the second exchange is unnecessary, as `temp` already contains the value in `arr[l+1]`. This was exploited by many output programs.

B. Overview of Improvements

It is interesting to consider the range of improvements the framework achieved. Table IV gives an overview of the results that were achieved during our factorial experimentation described in Section IX-D.

The original programs were compiled using GCC's –O2 optimization level (as it was the most sophisticated level we were able to profile using the simulator), and run through the 1000 validation tests used by the factorial runs to test output individuals. The number of instructions for each of the original programs are listed in Table IV. This is the baseline by which improvements must be measured.

TABLE IV
SUMMARY OF FACTORIAL EXPERIMENTS

	Output Programs				Instruction Counts			
Problem	Successful	Valid	Improvement	Unique	Original	Minimum	% Improvement	Median
Triangle1	11 507	2203	2112	1444	22 402	10 996	50.9	17214.5
Triangle2	11 433	1981	524	187	8380	4000	52.3	8000
Remainder	11 070	4805	3184	526	27 025	12 318	54.4	23 318
Sort1	11 502	3131	2708	128	884 842	545 878	38.3	569 517
Sort2	11 496	5271	1831	348	815 882	492 217	39.7	790 062
Factorial	11 514	3374	142	51	395 189	49 664	87.4	278 700
Switch 10	11 520	4763	4763	34	143 000	2000	98.6	36 000
Select	11 506	3653	1206	507	97 077	70 085	27.8	90201.5

The table shows the number of individuals classified as "valid" after checking them on a validation set (this does not guarantee semantic correctness in general), and those passing the tests that also constituted an improvement on the original program's performance.

Note that these figures simply summarize the experimentation, which is over a range of parameter values and we would not expect good performance at each of the 384 design points (parameter settings). As an indicator of the diversity of solutions produced for each problem, we list the number of unique instruction counts.

Most of the improvements were due to the elimination of redundant and not useful code. Other common types of algorithmic optimization were modifications of the control flow graph (for example, altering the `if` statements) to avoid the computation of non-necessary data and of conditions that are necessarily true (a clear example is Triangle1, see Section IX-A1). Given the used set of primitives, our system is technically able to make *any* type of modification to the source code. Currently, our system does not handle the introduction of new (local) variables. We use a temporary local variable (see `V_tmp` in Table II), but others could be introduced as well. Whether it is better to have fix number of extra variables, or rather having a GP system that can introduce any arbitrary number of new local variables, is a matter of further investigation. Both approaches have advantages and drawbacks.

C. Modeling Instruction Count

In order to construct a model, it was necessary to gather data from an evolutionary run, and thus we decided to execute one evolutionary run for each case study, with simulation enabled. We logged the number of instructions an individual used within the simulator and also the count of the high level primitives evaluated when running through the ECJ interpreter. We set parameters based on ECJ defaults for the run, and manually selected settings where defaults were not available or appropriate—the parameter settings themselves being of less importance in model-building than when actually optimizing software.

Our experiments used a population size and generation limit both set to 100, which we later increased to 250, effectively sampling 62 500 points in the search space. We carried out an evolutionary run as opposed to systematically sampling the space in order to ensure our sample was representative of the individuals likely to be encountered during a run.

TABLE V
INDIVIDUALS SAMPLED WHEN CONSTRUCTING MODELS

Problem	Total	Unique	Free of Run-Time Errors	Completed Successfully
Triangle1	62 500	13 847	13 835	13 833
Triangle2	62 500	12 960	12 959	12 959
Remainder	62 500	14 065	14 061	14 030
Sort1	62 500	8220	8220	8220
Sort2	62 500	7972	7972	7972
Factorial	62 500	9465	9463	9455
Switch	62 500	3047	3037	3034
Select	62 500	20 521	20 520	20 519

Our sample is restricted in its generality by two factors: first duplicate data where the same program is sampled more than once and second programs where full evaluation was not possible. The former is to be expected, particularly with seeding methods that heavily favored the introduction of programs identical or similar to the original input function. The latter was caused by problems such as run-time errors and iteration limit timeouts. A summary of data at the larger sample size illustrate the extent of this issue, as given by Table V.

This greatly reduces the effective sample size, and an exploratory comparison with the model produced by runs of 250 generations with a population size of 250 led us to use this increased sample size. Increasing the sample size any further would have taken individual evolutionary runs beyond the scope of weeks of compute time, which quickly became impractical.

Applying standard least squares linear regression to construct the model, it was observed that negative coefficients emerged. It is intuitive that this may cause problems: the evolutionary framework might exploit this by adding extra instructions with negative coefficients and this would improve the fitness of an individual provided that it did not interfere with its functional behavior. In other words, we were in danger of actually encouraging the emergence of bloat! Exploratory runs confirmed that negative coefficients quickly lead to a great deal of program bloat. We therefore repeated the regression and constrained coefficients to positive values only, using the R `nnls` library [64].

Some of the coefficients were zero, usually because the instructions concerned had not been sampled sufficiently to accurately estimate their cost. This is a limitation of the modeling approach, given a limited availability of compute power, and we accept these inaccuracies here as part of the expense of approximation.

```
int Select(int[] arr, int k, int n)
{
  int i=0,j=0,mid=0,a=0,temp=0;
  int flag = 0, flag2 = 0;
  int l=1;
  int ir=n;

  while (flag == 0)
  {
    if (ir <= l+1)
    {
      if (ir == l+1)
        if (arr[ir] < arr[l])
        {
          temp=(arr[l]);
          (arr[l])=(arr[ir]);
          (arr[ir])=temp;
        }

      flag = 1;
    }
   else if ( flag == 0)
        {
          mid=(l+ir) / 2;
          temp=(arr[mid]);
          (arr[mid])=(arr[l+1]);
          (arr[l+1])=temp;

          if (arr[l+1] > arr[ir])
          {
            temp=(arr[l+1]);
            (arr[l+1])=(arr[ir]);
            (arr[ir])=temp;
          }

          if (arr[l] > arr[ir])
          {
            temp=(arr[l]);
            (arr[l])=(arr[ir]);
            (arr[ir])=temp;
          }

          if (arr[l+l]> arr[l])
          {
            temp=(arr[l+1]);
            (arr[l+1])=(arr[l]);
            (arr[l])=temp;
          }

          i=l+1;
          j=ir;
          a=arr[l];

          while ( flag2 == 0)
          {
            i++;
            while (arr[i] < a)
              i++;
            j--;
            while (arr[j] > a)
              j--;

            if (j < i)
              flag2 = 1;

            if (flag2 == 0)
            {
              temp=(arr[i]);
              (arr[i])=(arr[j]);
              (arr[j])=temp;
            }

            arr[l]=arr[j]; arr[j]=a;

            if (j >= k)
              ir=j-1;
            if (j <= k)
              l=i;
          }
        }
  return arr[k];
}
```

Fig. 17. Source code of Select.

TABLE VI
ACCURACY OF MODELING

Problem	Tournament Size 2	Size 5	Size 7
Triangle1	66.1%	43.4%	41.4%
Triangle2	55.3%	29.0%	23.9%
Remainder	75.6%	64.6%	62.1%
Sort1	75.7%	73.8%	74.5%
Sort2	79.0%	73.4%	74.4%
Factorial	80.8%	69.3%	63.0%
Switch	74.1%	61.8%	57.1%
Select	72.0%	53.4%	49.8%

We can now evaluate the accuracy of modeling. By sampling a selection of individuals of tournament size n, then carrying out tournament selection on that group using first the instruction count from the simulator, and second the estimate from the model, we can measure the accuracy of a model by comparing how many times the two choose the same individual, and how many times they differ. The results are in Table VI, based on a sampling of 10 000 simulated tournament selections in each case. These generally compare well to the figure of 65% given in our previous work though it depends on the tournament size. We have in effect added noise to our fitness function.

Our model proved to be sufficient to achieve the optimizations previously described, and a full comparison to simulation is given in Section IX-E.

D. Important Components and Parameters

We used the results of the factorial experimentation to analyze the behavior of the framework.

The priority of a practitioner must be to produce error-free software first, and solutions that are time-efficient as a secondary objective. Therefore, for each problem we selected the ten treatments with the greatest probability of producing error-free solutions as measured by the proportion of zero error outputs in our repetitions. A new set of 30 runs for each of these ten treatments for each problem were then recorded by repeatedly running the framework with each treatment and recording the outcome of successful runs (nearly all runs were successful). The performance of these settings in our repeated results had similarly low error rates and produced completely valid programs in all but six cases.

A boxplot of the 30 repetitions for the ten best treatments chosen for Triangle1 is given in Fig. 18. There is little variety in the ability of the treatments to find good optimizations. These boxplots are representative of other problems: the top ten treatments generally give similar results (with a few treatments having much higher variance). In these case studies, it is observed that the framework is therefore very *robust* to parameter values. This is important because the highly parametrized nature of search algorithms can be a barrier to adoption if such robustness is not present.

We then chose the best parameter settings based on their medians, for use in Section IX-E. These settings cannot be regarded as optimal settings, only that they performed well in the full factorial on our problems. These settings are listed in Table VII, as levels denoted in Table III.

Taking these top ten treatments, we may examine them to assess the impact of specific parameter settings. Table VIII

TABLE VII
BEST PARAMETER SETTINGS FOR EACH PROBLEM

Parameters	Triangle1	Triangle2	Remainder	Sort1	Sort2	Factorial	Switch 10	Select
Probability of crossover	1	0	1	0	0	0	0	1
Population size	1	1	0	0	0	0	0	1
Tournament size	0	0	1	1	1	1	1	1
Seeding proportion	1	1	0	0	0	0	0	1
Coevolution enabled?	1	1	1	0	0	0	0	1
SPEA2 enabled?	0	0	0	0	0	0	0	0
SPEA2 archive size	0	1	0	0	0	0	0	0
Seeding method	1	0	0	0	0	0	0	0

lists the number of times the components were set to the high value in the best treatments for each problem. Note that the seeding method had three levels, hence the frequency of the low/medium/high values are given. The main patterns we can observe from this table are as follows.

1) Most parameters are heavily problem-dependent, that is sometimes the features of our framework are helpful and sometimes they are not. The problem-specific nature of the parameters is underlined by the similarity between the settings for the two problems optimizing sort functions.
2) Apart from Select (the largest of the problems), the best settings did not use SPEA2. We conjecture that this is because SPEA2 encourages exploration that is unnecessary to achieve good improvements in execution time for functions of this size. We investigate this behavior in more detail in Section IX-F.
3) The third type of seeding method, subtree seeding, is rarely used. Thus, it does not appear that "building block reassembly" is one of the mechanisms our framework uses to find optimizations.
4) Generally, a high value of seeding proportion is preferred. This confirms our expectations that seeding would be important: that GP is not creating solutions from scratch.
5) Coevolution is favored, particularly in those problems that have boundary conditions. It is likely that coevolution is able to find key discriminating test cases more efficiently than random testing.

It may be possible to separate our problems into those most suitable for optimization using mutation, and those more susceptible to crossover. The former involve smaller, localized optimizations that may be applied stepwise, i.e., their beneficial effect is additive. The latter require more radical restructuring. This may explain the modal pattern of best settings that we see between Triangle, Remainder, and Select as one group and the Sort, Factorial and Switch problems as another.

E. *Comparing modeling to Simulation*

To analyze the impact of using full simulation rather than modeling of a program's instruction usage, we used the parameter settings given in Table VII and ran 20 repetitions of each, first retaining the model as the method of evaluation, and second using the full simulator. The latter runs took days as full simulation is computationally expensive. Note that this does not necessarily prohibit the application of this method, as

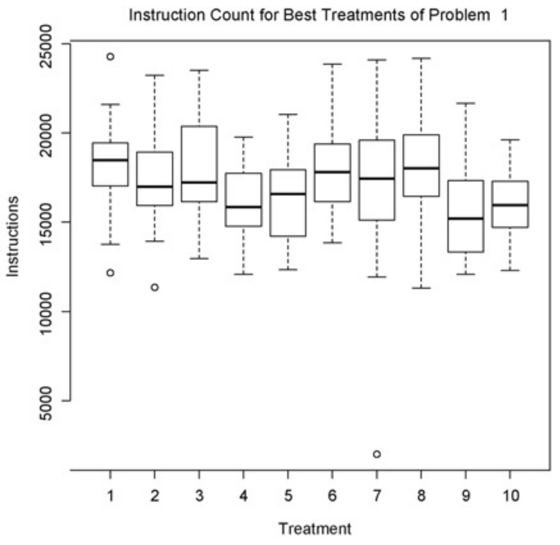


Fig. 18. Instruction counts for the ten best treatments of Triangle1.

simulation may be used alongside modeling, any application would not require so many runs, and the outputs found by automated optimization may be generalized into optimization strategies in a static manner.

Boxplots comparing the resulting distributions of instruction count are shown in Fig. 19. We then performed a nonparameteric Mann-Whitney rank-sum test for significance between the distributions for each problem. In cases where a significant difference was found at the 0.05% level, we carried out a further test of effect size to determine the *scientific significance* or importance of the difference, and all figures for these tests are given in Table IX.

Surprisingly, there are only four problems where there exists a statistically significant difference at the 0.05% level. Taking significant effect size for the Vargha-Delaney A statistic [65] at less than 0.36 or greater than 0.64, we can state that these four differences are also *scientifically significant*, i.e., of a magnitude that we should be interested in. As expected, those significant differences all represented cases where the use of a simulator improved performance.

Whether or not to employ simulation therefore depends on the goal of the practitioner. For further research along the lines of this paper, modeling is an efficient alternative to simulation that still provides the opportunity to achieve large optimizations. Similarly, incorporating large-scale search into software design in the medium term would appear to

TABLE VIII
NUMBER OF TOP TEN TREATMENTS CONTAINING HIGH VALUES

Parameters	Triangle1	Triangle2	Remainder	Sort1	Sort2	Factorial	Switch 10	Select
Probability of crossover	10	7	10	0	0	0	0	4
Population size	4	10	4	0	0	0	0	4
Tournament size	7	4	5	10	10	10	10	4
Seeding proportion	6	8	7	6	6	6	6	8
Coevolution enabled?	10	10	8	4	3	4	4	4
SPEA2 enabled?	0	0	2	0	0	0	0	6
SPEA2 archive size	4	5	5	5	5	5	5	5
Seeding method	4/6/0	5/5/0	5/5/0	7/3/0	6/2/1	7/3/0	7/3/0	5/2/3

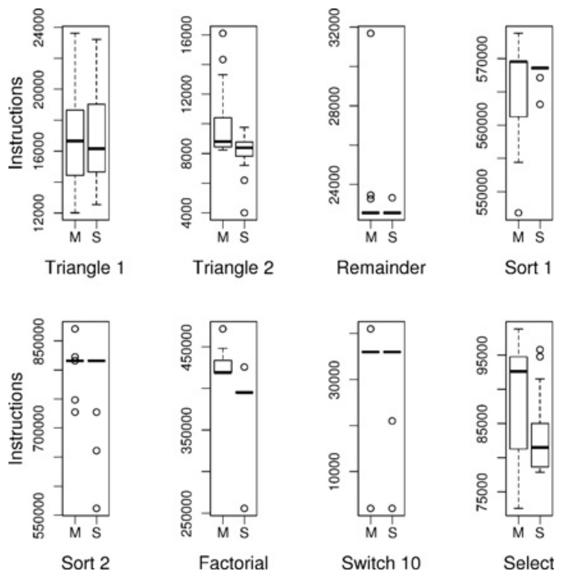


Fig. 19. Comparing instruction counts using modeling versus simulation.

TABLE IX
COMPARING INSTRUCTION COUNT DISTRIBUTIONS USING MODELING VERSUS SIMULATION

Problem	Rank-Sum P-Value	Vargha-Delaney A Statistic
Triangle1	0.8924	0.5137
Triangle2	0.0060	0.2450
Remainder	0.2988	0.4488
Sort1	0.0203	0.2950
Sort2	0.4233	0.4425
Factorial	6.89e-08	0.0375
Switch 10	0.3496	0.4537
Select	0.0206	0.2850

favor the option of modeling over simulation due to the computational requirements of the simulator. Certainly, manually designed or more sophisticated automated modeling methods may improve on the simple linear model used in this paper.

When optimizing more complex programs, or taking into account detailed machine-level behavior such as cache access and misses, it is recommended that a simulator is employed. Whilst a simple approach to modeling can achieve impressive results, it is unlikely to be able to do so as the relationship between high-level source code and run-time behavior becomes more complex.

F. Exploration Using MOO

We previously conjectured that taking a Pareto-based approach using the SPEA2 algorithm would increase the exploration of the search space, such that small highly fit subtrees may be recombined efficiently to create improved solutions. To assess the impact of using SPEA2, we took the best parameter settings from the factorial experiments (all of which did not use MOO) and compared them to the same settings with MOO enabled, over 30 repetitions. During the runs, we logged the pair of values (error, instructions) for each individual evaluated.

Fig. 20 compares the exploration of the *objective* space between the two pairs of settings, for a single repetition of Triangle1 (other repetitions produced very similar results). Surprisingly, the difference is not great. The SPEA2 method explores more points closer to the y-axis, but the difference is limited and the error values on those extra points explored are quite large. Thus it appears that SPEA2 is not exploring small, highly-fit programs in this case. Most other problems have similar plots.

The exception to the rule is Select. The same type of plot is given for this problem in Fig. 21. This is the largest program and arguably the most complex. We can see in Table VIII that six out of ten of the best treatments used SPEA2 in this case, which is exceptional compared to the other problems. The same plot for this problem tells a very different story: SPEA2 has extensively explored the objective space, whereas the weighted sum method has not.

To summarize the data, Table X lists the number of unique points in the objective space sampled across the thirty repetitions for each problem, both for the weighted approach (SPEA2 disabled) and with SPEA2 enabled. SPEA2 samples far more unique points in general, although Triangle2 is an exception.

We conjecture at this stage that SPEA2 or other Pareto-based methods are most likely to be useful in optimizing larger programs, or at least that diversity-maintaining functions should be used when trying to achieve a scalable optimization method. For smaller programs, of the size we have examined here, it is unlikely to be of use.

X. LIMITATIONS

Our novel framework has the following limitations.

1) Software testing cannot prove that a program is free of errors [49]. Because the modifications we apply to the

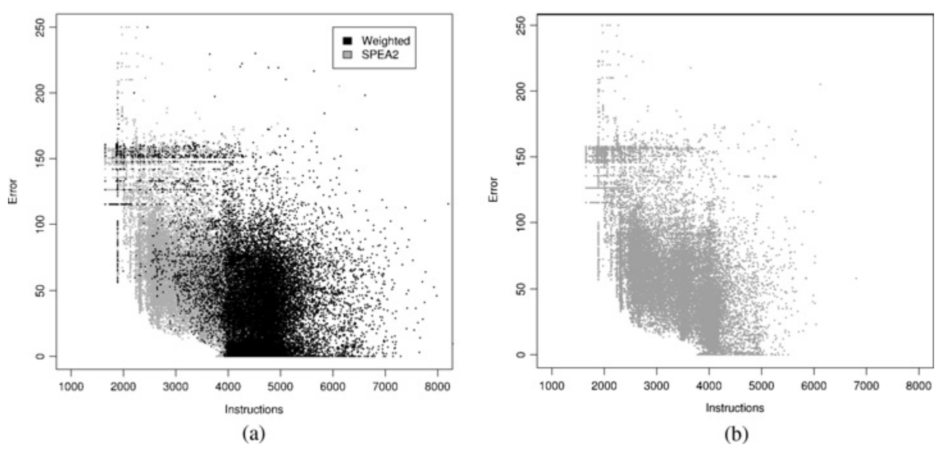


Fig. 20. Exploration of the objective space for Triangle1. (a) Weighted fitness. (b) Pareto-based fitness.

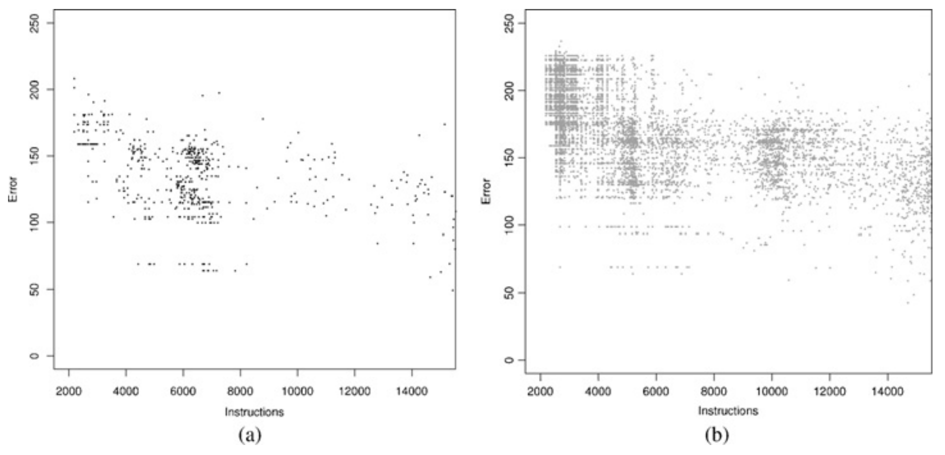


Fig. 21. Exploration of the objective space for Select. (a) Weighted fitness. (b) Pareto-based fitness.

TABLE X

UNIQUE OBJECTIVE VALUES SAMPLED BY WEIGHTED AND SPEA2 METHODS

	Triangle1	Triangle2	Remainder	Sort1	Sort2	Factorial	Switch 10	Select
Sampled	1 500 000	1 500 000	1 500 000	1 500 000	1 500 000	1 500 000	1 500 000	1 500 000
Unique weighted	692 170	755 688	166 782	241 206	194 215	155 401	37 150	435 618
Unique SPEA2	720 198	580 429	912 945	521 717	626 644	637 414	256 402	682 031

programs are not semantics-preserving, we cannot guarantee that the output of our framework is semantically equivalent to the input program. Therefore, output must be manually verified.

2) In our current prototype, the space of all test cases used to validate the program's semantics is constructed before the search. They are chosen based on structural criteria (e.g., branch coverage) of the input program. Evolving programs can have different control flow and different boundary conditions. A test set designed for the input program may not be appropriate for the evolving programs. An alternative would be to generate new test cases at each generation. Several different heuristics could be designed to choose for example how many new test cases to create, when to create them, which program to use for the testing (e.g., the best in the current population), how to choose the old test cases to discard, and so on.

XI. Future Work

The framework we present in this paper is quite complex and the approach we employ is novel. There are many directions future research could take.

A. Semantic Correctness

The problem of preserving semantic correctness is an outstanding issue in applying optimization in this manner. We hope that by publishing these results, we may generate debate as to ways of addressing this issue. The way we have approached this problem here is to verify results manually, which did not prove too difficult albeit with small example program sizes.

A more philosophical approach is to accept the fallibility of evolutionary search in the same way that we accept the fallibility of human programmers, and rely on the testing performed to increase our confidence in its correctness. Another approach is to tackle problems without a Boolean value of acceptability, where a quality of service can be given as a quantifiable, continuous, measure. For example, a sort algorithm is correct or it is not, but an image filter may not have such a strict definition of correctness.

However, we originally selected programs with Boolean levels of functionality because we were interested in addressing this issue using coevolution. Coevolution has proved effective for these case studies, but a more formal approach to validation would be preferable. Therefore, two approaches may be of interest: first using semantics-preserving transformations to either carry out the original optimization, or subsequently verify the coevolved optimization. The second approach is to employ a technique such as model checking to formally verify optimized programs. We are currently working on combining GP and model-checking to achieve this goal, but it is limited by the scalability of contemporary model-checkers.

B. Scalability

Scalability is an important factor that needs to be studied in more detail. Will this approach be effective only on relatively small functions, or can it scale up to larger systems? Even if its scalability is limited, it may still be useful because it can obtain types of optimizations that current techniques are not able to obtain.

In this paper, we were limited to optimizing small example programs because we wanted to run large numbers of experiments to analyze the usefulness of individual framework components. This is reflected in the amount of computational effort applied for each GP run: 50 000 fitness evaluations is small by the standards and much modern GP work (e.g. [29]). Thus there is much scope for an increased number of fitness evaluations. The wall clock time of a single run depends on multiple factors: whether simulation is employed, the size of the code, the types of primitives used (chiefly, the loop structures involved) and the average tree size of the GP solutions. Thus it is difficult to give precise estimates of execution time corresponding to code size alone. It would intuitively make more sense to set a wall clock time-limit, and optimize the framework for that time-limit, when considered the scalability of a deployable optimization tool.

The scalability of the method is related to how the tradeoff between exploration and exploitation is set. On one hand, we do not believe at the moment that entire software systems can be re-arranged in new algorithmic ways (e.g., given as input a bubble sort we do not expect as output a merge sort). On the other hand, we conjecture that local modifications that improve the performance could be obtained even for larger software (see for example the very easy type of improvement that can be obtained for the case study Triangle1).

C. Seeding

Seeding strategies are important to help the search process to focus on promising regions of the search space by exploiting useful building blocks from the input program. However, seeding strategies can have a drastic impact on the *diversity* of the program population. Search operators that have been designed for randomly initialized populations (e.g., Koza's single point crossover) are likely not the best option for greedy seeding strategies. We hence conjecture that for each type of strategy we want to use we also need to define tailored search operators to improve the final results. Different further types of strategies and relative operators can be defined. Large empirical studies need to be carried out to validate their performance.

D. Non-Functional Properties

Execution time is only one possible non-functional criterion that can be optimized. Other criteria need to be investigated as well, for example power consumption. It will be important to study how many criteria can be optimized at the same time, and assess the impact on the search process when more than one criterion is addressed.

E. Analyzing Optimizations

When analyzing the results we were faced with a daunting task: *there were literally thousands of different optimized programs produced*, and these optimizations contain potentially useful information. Perhaps by applying data-mining methods, common optimizations could be isolated and examples presented to an engineer. Furthermore, mining results from a variety of problems could lead to generic templates of optimization methods that could be incorporated into a conventional compiler.

XII. Conclusion

In this paper, we presented a novel framework to improve non-functional properties of software. In particular, we concentrated on execution time, although other properties could be considered as well.

We used transformations of the programs that do not guarantee the preservation of the original semantics. This enables us to produce new versions of the programs that could not be obtained with semantic preserving operators.

To address the problem of preserving the original semantics, we intensively tested the evolving programs. Because the original input program we want to optimize can be used as an oracle, we can generate as many test cases as we wish (dependent upon the computational resources and time constraints).

At each generation, we evolved the programs to pass all the given test cases, but at the same time we also evolved the test cases to find new unknown faults in the programs.

We demonstrated our novel approach on a set of case studies. For each, we obtained new improved versions. To our best knowledge, the types of algorithmic improvements we obtained cannot be obtained with current compilers.

Acknowledgment

The authors are grateful to P. Cairns and S. Poulding for insightful discussions, and to S. Sen and I. Bate for helpful feedback.

REFERENCES

[1] J. Collard, *Reasoning About Program Transformations*, 1st ed. Berlin, Germany: Springer, 2002.

[2] K. Deb, *Multi-Objective Optimization Using Evolutionary Algorithms*. New York: Wiley, 2001.

[3] B. Beizer, *Software Testing Techniques*. New York: Van Nostrand Rheinhold, 1990.

[4] A. Arcuri, D. R. White, J. Clark, and X. Yao, "Multiobjective improvement of software using co-evolution and smart seeding," in *Proc. Int. Conf. SEAL*, 2008, pp. 61–70.

[5] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*, 2nd ed. Reading, MA: Addison-Wesley, 1986.

[6] M. Stephenson, S. Amarasinghe, M. Martin, and U. M. O'Reilly, "Meta optimization: Improving compiler heuristics with machine learning," *SIGPLAN Not.*, vol. 38, no. 5, pp. 77–90, 2003.

[7] S. Leventhal, L. Yuan, N. K. Bambha, S. S. Bhattacharyya, and G. Qu, "DSP address optimization using evolutionary algorithms," in *Proc. Workshop Softw. Compilers Embedded Syst.*, 2005, pp. 91–98.

[8] F. Kri and M. Feeley, "Genetic instruction scheduling and register allocation," in *Proc. Quantitative Eval. Syst. Conf.*, 2004, pp. 76–83.

[9] K. D. Cooper, P. J. Schielke, and D. Subramanian, "Optimizing for reduced code space using genetic algorithms," in *Proc. Workshop Languages, Compilers Tools Embedded Syst.*, 1999, pp. 1–9.

[10] P. Kulkarni, S. Hines, J. Hiser, D. Whalley, J. Davidson, and D. Jones, "Fast searches for effective optimization phase sequences," in *Proc. Conf. Programming Language Design Implementation*, 2004, pp. 171–182.

[11] G. Fursin, C. Miranda, O. Temam, M. Namolaru, E. Yom-Tov, A. Zaks, B. Mendelson, P. Barnard, E. Ashton, E. Courtois, F. Bodin, E. Bonilla, J. Thomson, H. Leather, C. Williams, and M. O'Boyle, "Milepost GCC: Machine learning based research compiler," in *Proc. GCC Developers' Summit*, 2008, pp. 1–13.

[12] K. Hoste and L. Eeckhout, "Cole: Compiler optimization level exploration," in *Proc. Int. Symp. Code Gener. Optimization*, 2008, pp. 165–174.

[13] X. Li, M. J. Garzaran, and D. Padua, "Optimizing sorting with genetic algorithms," in *Proc. Int. Symp. Code Generation Optimization*, 2005, pp. 99–110.

[14] C. P. Gomes and B. Selman, "Practical aspects of algorithm portfolio design," in *Proc. 3rd ILOG Int. Users Meeting*, 1997, pp. 200–201.

[15] W. B. Langdon and P. Nordin, "Seeding genetic programming populations," in *Proc. EuroGP*, 2000, pp. 304–315.

[16] N. E. Fenton and S. L. Pfleeger, *Software Metrics: A Rigorous and Practical Approach*. Boston, MA: PWS Publishing, 1998.

[17] M. Harman and L. Tratt, "Pareto optimal search based refactoring at the design level," in *Proc. GECCO*, 2007, pp. 1106–1113.

[18] C. Ryan, A. H. M. van Roermund, and C. J. M. Verhoeven, *Automatic Re-Engineering of Software Using Genetic Programming*. Norwell, MA: Kluwer, 1999.

[19] W. B. Langdon, "Scheduling maintenance of electrical power transmission networks using genetic programming," in *Proc. Late Breaking Papers GP-96 Conf.*, 1996, pp. 107–116.

[20] C. H. Westerberg and J. Levine, "Investigation of different seeding strategies in a genetic planner," in *Proc. EvoWorkshops*, 2001, pp. 505–514.

[21] A. J. Marek, W. D. Smart, and M. C. Martin, "Learning visual feature detectors for obstacle avoidance using genetic programming," in *Proc. Late Breaking Papers GECCO*, 2002, pp. 330–336.

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

[23] S. Forrest, T. Nguyen, W. Weimer, and C. Le Goues, "A genetic programming approach to automated software repair," in *Proc. 11th Annu. Conf. GECCO*, 2009, pp. 947–954.

[24] M. D. Schmidt and H. Lipson, "Incorporating expert knowledge in evolutionary search: A study of seeding methods," in *Proc. GECCO*, 2009, pp. 1091–1098.

[25] R. Poli, W. B. Langdon, and N. F. McPhee. (2008). *A Field Guide to Genetic Programming* [Online]. Available: http://www.gp-field-guide.org.uk

[26] T. Mitchell, *Machine Learning*. New York: McGraw-Hill, 1997.

[27] N. L. Cramer, "A representation for the adaptive generation of simple sequential programs," in *Proc. Int. Conf. Genet. Algorithms Applicat.*, 1985, pp. 183–187.

[28] J. R. Koza, *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. Cambridge, MA: MIT Press, 1992.

[29] J. R. Koza, S. H. Al-Sakran, L. W. Jones, and G. Manassero, "Automated synthesis of a fixed-length loaded symmetric dipole antenna whose gain exceeds that of a commercial antenna and matches the theoretical maximum," in *Proc. GECCO*, 2007, pp. 2074–2081.

[30] D. J. Montana, "Strongly typed GP," *Evol. Comput.*, vol. 3, no. 2, pp. 199–230, 1995.

[31] S. Luke and L. Panait, "A comparison of bloat control methods for genetic programming," *Evol. Comput.*, vol. 14, no. 3, pp. 309–344, 2006.

[32] J. Paredis, "Coevolving cellular automata: Be aware of the red queen," in *Proc. ICGA*, 1997, pp. 393–400.

[33] D. Cliff and G. F. Miller, "Tracking the red queen: Measurements of adaptive progress in co-evolutionary simulations," in *Proc. Eur. Conf. Artif. Life*, 1995, pp. 200–218.

[34] T. Miconi, "The road to everywhere: Evolution, coevolution and progress in nature and in computers," Ph.D. dissertation, School Comput. Sci., Univ. Birmingham, Birmingham, U.K., 2007.

[35] T. Miconi, "Why coevolution doesn't work: Superiority and progress in coevolution," in *Proc. 12th Eur. Conf. Genet. Programming*, LNCS 5481. 2009, pp. 49–60.

[36] S. G. Ficici and J. B. Pollack, "Challenges in coevolutionary learning: Arms-race dynamics, open-endedness, and mediocre stable states," in *Proc. 6th Int. Conf. Artif. Life*, 1998, pp. 238–247.

[37] C. D. Rosin and R. K. Belew, "New methods for competitive coevolution," *Evol. Comput.*, vol. 5, no. 1, pp. 1–29, 1997.

[38] E. D. Jong, "The incremental Pareto-coevolution archive," in *Proc. GECCO*, 2004, pp. 525–536.

[39] E. D. Jong, "The maxsolve algorithm for coevolution," in *Proc. GECCO*, 2005, pp. 483–489.

[40] H. Juillé and J. B. Pollack, "Coevolving the ideal trainer: Application to the discovery of cellular automata rules," in *Proc. Conf. Genet. Programming*, 1998, pp. 519–527.

[41] W. D. Hillis, "Co-evolving parasites improve simulated evolution as an optimization procedure," *Physica D*, vol. 42, nos. 1–3, pp. 228–234, 1990.

[42] A. Ronge and M. G. Nordahl, "Genetic programs and co-evolution: Developing robust general purpose controllers using local mating in two dimensional populations," in *Proc. 4th Int. Conf. Evol. Comput. Parallel Problem Solving Nature IV*, 1996, pp. 81–90.

[43] D. Ashlockand, S. Willson, and N. Leahy, "Coevolution and tartarus," in *Proc. IEEE CEC*, Jun. 2004, pp. 1618–1624.

[44] A. Arcuri and X. Yao, "Co-evolutionary automatic programming for software development," *Inform. Sci.*, to be published.

[45] C. A. C. Coello, G. B. Lamont, and D. A. V. Veldhuizen, *Evolutionary Algorithms for Solving Multi-Objective Problems*. Berlin, Germany: Springer, 2007.

[46] M. Reformat, C. Xinwei, and J. Miller, "On the possibilities of (pseudo-) software cloning from external interactions," *Soft Comput.*, vol. 12, no. 1, pp. 29–49, 2007.

[47] J. H. Andrews, L. C. Briand, Y. Labiche, and A. S. Namin, "Using mutation analysis for assessing and comparing testing coverage criteria," *IEEE Trans. Softw. Eng.*, vol. 32, no. 8, pp. 608–624, Aug. 2006.

[48] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms.*, 2nd ed. New York: MIT Press/McGraw-Hill, 2001.

[49] G. Myers, *The Art of Software Testing*. New York: Wiley, 1979.

[50] P. McMinn, "Search-based software test data generation: A survey," *Softw. Testing Verification Reliab.*, vol. 14, no. 2, pp. 105–156, Jun. 2004.

[51] J. C. King, "Symbolic execution and program testing," *Commun. ACM*, vol. 19, no. 7, pp. 385–394, Jul. 1976.

[52] B. Mesman, L. Spaanenburg, H. Brinksma, E. F. Deprettere, E. Verhulst, F. Timmer, H. van Gageldonk, L. D. J. Eggermont, R. van Leuken, T. Krol, and W. Hendriksen, *Embedded Systems Roadmap—Vision on Technology for the Future of PROGRESS*. Utrecht, The Netherlands: STW Technology Foundation, 2002.

[53] N. Binkert, R. Dreslinski, L. Hsu, K. Lim, A. Saidi, and S. Reinhardt, "The M5 simulator: Modeling networked systems," *IEEE Micro*, vol. 26, no. 4, pp. 52–60, Jul.–Aug. 2006.

[54] W. B. Langdon and R. Poli, *Foundations of Genetic Programming*. Berlin, Germany: Springer-Verlag, 2002.

[55] E. Zitzler, M. Laumanns, and L. Thiele, "SPEA2: Improving the strength Pareto evolutionary algorithm," Swiss Federal Instit. Technol., Zurich, Switzerland, Tech. Rep. 103, 2001.

[56] A. Arcuri, "It does matter how you normalize the branch distance in search based software testing," in *Proc. IEEE Int. Conf. Softw. Testing Verification Validation*, 2010, pp. 205–214.

[57] J. Miller, M. Reformat, and H. Zhang, "Automatic test data generation using genetic algorithm and program dependence graphs," *Inform. Softw. Technol.*, vol. 48, no. 7, pp. 586–605, 2006.

[58] R. Sagarna and J. Lozano, "Scatter search in software testing, comparison and collaboration with estimation of distribution algorithms," *Eur. J. Oper. Res.*, vol. 169, no. 2, pp. 392–412, 2006.

[59] Mälardalen WCET Research Group. *WCET Project Benchmarks* [Online]. Available: http://www.mrtc.mdh.se/projects/wcet/benchmarks.html

[60] ECJ. *Evolutionary Computation in Java* [Online]. Available: http://www.cs.gmu.edu/∼eclab/projects/ecj

[61] *Javacc* [Online]. Available: http://javacc.dev.java.net

[62] D. R. White and S. Poulding, "A rigorous evaluation of crossover and mutation in genetic programming," in *Proc. 12th EuroGP*, 2009, pp. 220–231.

[63] R. Poli and N. McPhee, "Parsimony pressure made easy," in *Proc. 10th Ann. Conf. GECCO*, 2008, pp. 1267–1274.

[64] R. Development Core Team, *R: A Language and Environment for Statistical Computing*. Vienna, Austria: R. Foundation for Statistical Computing, 2008 [Online]. Available: http://www.R-project.org

[65] A. Vargha and H. Delaney, "A critique and improvement of the CL common language effect size statistics of McGraw and Wong," *J. Educ. Behav. Statist.*, vol. 25, no. 2, pp. 101–132, 2000.

David R. White received the M.Eng. and Ph.D. degrees in computer science from the University of York, York, U.K., in 2002 and 2010, respectively.

He is currently with the Department of Computer Science, University of York. His current research interests include the theory, empirical analysis and application of genetic programming, and search-based software engineering.

Andrea Arcuri received the B.S. and M.S. degrees in computer science from the University of Pisa, Pisa, Italy, in 2004 and 2006, respectively, and the Ph.D. degree in computer science from the University of Birmingham, Birmingham, U.K., in 2009.

He is currently with the Simula Research Laboratory, Lysaker, Norway. His current research interests include search-based software testing and analyses of randomized algorithms.

John A. Clark is currently a Professor of Critical Systems with the Department of Computer Science, University of York, York, U.K.

His work is focused on software engineering and secure systems engineering. He was with the Secure Systems Division of the software and systems house Logica, Cobham, Surrey, U.K., and London, U.K., for five and half years before joining the University of York in 1992. He has used heuristic search techniques to address a range of security and software engineering problems, including automated testing of implementations against formal specifications, breaking proposed program invariants, automated secure protocol synthesis, cryptanalysis of zero-knowledge schemes, the design of cryptographic components, and the synthesis of quantum circuitry.