Using Genetic Improvement for Multi-Objective Optimization of Software

Gregory Timmon
Department of Computer Science
North Carolina State University
Raleigh, NC 27606
gbtimmon@ncsu.edu

Siddharth Sharma
Department of Computer Science
North Carolina State University
Raleigh, NC 27606
ssharm24@ncsu.edu

Megha Umesha
Department of Computer Science
North Carolina State University
Raleigh, NC 27606
mumesha@ncsu.edu

ABSTRACT

Software Development requires the fulfillment of both functional and non-functional requirements and maintenance of the software after development. Optimizing non-functional requirements and maintenance proves to be a burden on the programmer and hence, to ease with this burden, Genetic Programming techniques are being extensively used. In this study, we try and summarize the work done in this area using techniques like Genetic Improvement, Code Transplantation and Automated Program Repair. All the papers reviewed by us are related to this new kind of software development environment. We also comment on the possible improvements for those papers and future work.

Keywords

Search Based Software Engineering, Genetic Programming, Genetic Improvement, Multi-objective optimization, Software Transplantation

1. INTRODUCTION

In areas such as artificial intelligence and machine learning, there are many problems whose resolution can be considered as the search of a computer program, inside a space of possible programs. This search should be carried out in such a way that the searched program be the more adequate to the problem that is being considered. This idea can be successfully translated to Search Based Software Engineering, where search paradigms are used to optimize software. The genetic programming (GP) paradigm provides us the appropriate framework to develop this type of search in an efficient and versatile way, since it adapts to any problem type.

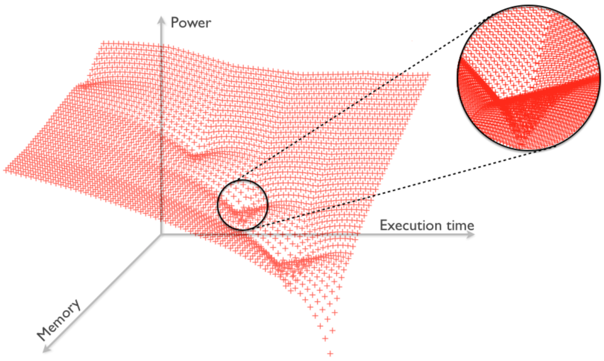
Genetic Programming can effectively search a large solution space with the help of candidate solutions. Selection, Mutation, Fitness are some of the key factors in genetic programming. Applying genetic programming to Multiple-objective problems gives us a spectrum of ”best” solutions. This spectrum is called the Pareto frontier.

Most of the research work studied in this paper use genetic programming or a variant of it like genetic improvement to solve problems of the development of software.

2. MOTIVATION

Harman *et al.* [7] put forward a vision to automate much of the slow and inefficient way we currently develop software, thus reducing the burden on humans, who can then focus better on the earlier stages of the development process. Recent advancements in Genetic Programming have shown that we can efficiently and quickly generate very large space of candidate solutions which can then be navigated for optimality, by humans. In addition, software maintenance and repair is another tedious and time-consuming task. Application of Genetic Programming for this task is explored in [5]. The authors have described how GP can be combined with program analysis methods to repair bugs in off-the-shelf legacy C programs.

In [7], to compare the different trade-offs and find the best program, visualization is done by graphing the pareto program surface in three dimensions (x, y, z), where x is the execution time, y is the power consumed and z is the memory consumed. Knee points, which are regions in solution space, of rapid change in trade-offs between variables were also observed.



Engineers will also need to strike a balance between performance and accuracy when performing computations. Maximum possible accuracy may seem desirable, but achieving that might mean sacrificing significantly on the performance. To achieve good performance while compromising on accuracy to an acceptable level, Douskas *et al.* [15] proposed a technique called loop perforation. Loop perforation is not a genetic technique, but it is an alternative approach for software optimization.

Several factors have to be considered when optimizing a software for the non-functional properties such as dependence on low-level details that are invisible to a developer or external factors like operating system and memory cache events. This makes manual optimization impossible. Hence, Genetic programming's success in optimization problems is exploited for the task. In [1], Arcuri *et al.* proposed a technique where only execution time is optimized, but a similar technique could be used to optimize other properties as well.

In [12], Petke *et al.* examined if GP is a feasible approach for generating novel optimized code by comparing the speed up of GP optimized code to human expert optimized examples.

Harman *et al.* [11] investigated the application of genetic improvement (GI) to MiniSAT in order to obtain optimized code, additionally introducing multi-donor software transplantation.

In [3], the authors state two primary needs to worry about energy consumption of an application. The first one being the increase of smartphone and cloud server usage, which benefit greatly from energy efficiency. Our energy grid is affected by servers and optimizing them can help in reduction of CO_2 emission. And the other one is that the programmer disconnect between source code written and energy consumption of the compiled application.

3. BACKGROUND AND CONCEPTS

- Search Based Software Engineering (SBSE) : Search Based Software Engineering is an approach to software engineering in which search-based optimizations techniques like genetic programming, simulated annealing, etc are applied to obtain near-optimal solutions to the given problem.

- Pareto Program Surface : It is the surface that is generated when programs are plotted with non-functional requirements (such as memory, execution time, power consumed, etc) as the dimensions.

- Oracle : An oracle is a mechanism for determining whether the program has passed or failed a test. It involves comparing the output generated to the desired output for a test case.

- Sensitivity Analysis : It is a method by which we can determine which parts of the program affects the non functional requirements the most, and how. It can inform us which code to optimize.

- Loop Perforation : Loop Perforation involves identifying the loops in the application code that can be executed for lesser number of iterations, thereby reducing the amount of computation and increasing the performance, while obtaining sufficiently accurate results.

- Criticality Testing : It is the first phase of loop perforation, in which critical loops whose modification may lead to unexpected results are identified and filtered.

- Perforation rate : It is a parameter for loop perforation which represents the expected percentage of loop iterations to skip.

- Pareto-optimal perforation : A perforation which is the best in terms of both accuracy and performance.

- Perforation space exploration : It is the second phase of loop perforation, where the set of tunable loops (output of critical testing) are tested at various perforation rates and a set of Pareto-optimal loop/perforation rate pairs are returned.

- Global patterns : In the applications that are studied in the paper, several patterns have been identified such as loops that iterate over the search space, loops that use certain search metrics to find an element, etc, which are called global patterns. Finding these patterns in an application means there is scope for loop perforation in the application.

- Abstract Syntax Tree : AST is a tree model of an entire program or a certain program structure. It âĂIJabstractâĂİ in the sense that some of the actual characters used in the program code do not appear in the AST.

- Delta debugging : It is a methodology to automate the debugging of programs using an algorithm that builds on unit testing to isolate failure causes automatically - by systematically narrowing down failure-inducing circumstances until a minimal set remains.

- Fitness Function : âĂIJA fitness function is a particular type of objective function that is used to summarise, as a single figure of merit, how close a given design solution is to achieving the set aims.âĂİ Analogously, in this paper, the solution that is being searched for is a program that achieves the required functionality while avoiding the program bugs. In the paper, the internal representation of a program is compiled into an executable one and run against the set of positive and negative test cases. Fitness is calculated as the weighted sum of the test cases passed. Uncompilable programs and those whose runtimes exceed a predetermined threshold are assigned fitness zero.

- Genetic Operators : It is an operator used in genetic algorithms to guide the algorithm towards a solution to a given problem. There are three types of genetic operators: mutation, crossover and selection. These must work in conjunction with one another in order for the algorithm to be successful.

- Multi-objective optimization (MOO) : Multiobjective optimization involves minimizing or maximizing multiple objective functions subject to a set of constraints.

- Strength Pareto Evolutionary Algorithm (SPEA) : Strength Pareto Evolutionary Algorithm is a Multiple Objective Optimization algorithm and an Evolutionary Algorithm. The objective of the algorithm is to locate and and maintain a front of non-dominated solutions, ideally a set of Pareto optimal solutions.

- Branch coverage : âĂIJBranch coverage is a testing method, which aims to ensure that each one of the possible branch from each decision point is executed at least once and thereby ensuring that all reachable code is executed. That is, every branch taken each way, true and false.âĂİ In the paper, branch coverage is used to generate a set of test cases with high behavioral diversity.

- Semantic Score : In this paper, semantic score is a component of the fitness function of a GP individual. It is defined as the sum of the errors from the expected outcomes.

- MiniSAT : is a minimalistic, open-source SAT solver, developed to help researchers and developers alike to get started on SAT.

- BNF (Backus Normal Form or BackusâĂŞNaur Form) : one of the two main notation techniques for context-free grammars, often used to describe the syntax of languages used in computing.

- Boolean Satisfiability Problem or SAT : is the problem of determining if there exists an interpretation that satisfies a given Boolean formula In other words, it asks whether the variables of a given Boolean formula can be consistently replaced by the values TRUE or FALSE in such a way that the formula evaluates to TRUE.

- Assertion : A statement that a predicate (Boolean-valued function, a trueâĂŞfalse expression) is expected to always be true at that point in the code. If an assertion evaluates to false at run time, an assertion failure results, which typically causes the program to crash, or to throw an assertion exception.

- Combinatorial Interaction Testing (CIT) : Combinatorial Interaction Testing (CIT) is a black box sampling technique derived from the statistical field of design of experiments. It has been used extensively to sample inputs to software, and more recently to test highly configurable software systems and GUI event sequences.

- Interaction Testing (CIT) is a black box sampling technique derived from the statistical field of design of experiments. It has been used extensively to sample inputs to software, and more recently to test highly configurable software systems and GUI event sequences.

- Code transplants : Code transplant is the process of evolving a new program by reusing and modifying existing code.

- Multi-donor transplantation : The original program is genetically improvised using parts and structures from several âĂIJdonorâĂİ programs that solve the same problem. Edit List: A list of mutation changes that have to be applied to the original program to obtain the final mutated program. This method of representation saves a lot of memory as the entire mutated program need not be stored in the memory.

- Genetic Improvement : An area of Search Based Software Engineering which seeks to improve softwareâĂŹs non-functional properties by treating program code as if it were genetic material which is then evolved to produce more optimal solutions.

- Downstream applications : The applications that use the upstream(orignal) MiniSAT solver as the basis to do more complex tasks.

- Ensemble Computation : The study of an NP-complete variant of the Boolean circuit problem where one must find the smallest circuit that satisfies a set of Boolean functions simultaneously. This problem can be translated into a satisfiability problem.

- AProVE: Automated Program Verification Environment is a system for the generation of automated termination proofs of term rewrite systems. AProVE uses a Boolean satisfiability solver to determine which paths can or cannot be reached.

- Automatic Transplantation : An application of Genetic Improvement to identify and port(extract-isolate-embed) features from a host application to a donor application.

- in-situ Testing : A form of testing that con- strains the input space of traditional testing to more closely approximate behaviourally relevant inputs.

- Regression Testing : Testing carried out to ensure that changes made in the fixes or any enhancement changes are not impacting the previously working functionality. It is executed after enhancement or defect fixes in the software or its environment.

4. RELATED WORK

Recent works in genetic improvement include evolving versions of Pseudo random generator by White, and code migration of a kernel component of gzip UNIX utility, to GPU(an entirely different platform) by a GP(Genetic Programming) engine. In software companies like Google [19], Microsoft [4, 9] , GP engines have recently been used for Bug detection and fixing. These works lay the foundation for GISMOE approach.

In the past, there have been several efforts to trade accuracy with different factors like performance, robustness, energy consumption, etc. Among these efforts, one of them explored task skipping (which is comparable to loop perforation [15]) to reduce resource usage when maintaining acceptable accuracy. In another effort, programmers had to provide several implementations for a specific functionality and the implementations represented different points on the performance-accuracy space. An appropriate implementation would be chosen according to the problem at hand. Yet another work called Autotuners [6, 17, 18] involved the exploration of alternatives that had the exact same accuracy.

Previously, Finite State Machines were used to write formal specifications that a modified program had to follow, which is not the case in the approach described in the paper [5]. The approach deals with long term program repair, but further research in short term repair strategies like patching buggy data structures can complement this paperâĂŹs work.

Earlier, genetic programming was applied on a single program, parts of code from it were extracted, modified and reinserted back into the code, compared to the approach adopted in [11] where code is transplanted from multiple programs. Previously, Genetic improvement has been used for a variety of tasks: automation of the bug-fixing process, improvement of non-functional properties of programs, automatic migration of a system from one platform to another.

Genetic Improvement has been demonstrated to effectively reduce execution time while working with compiled code in the post compilation process. Also it has been tested on the specific SAT solver problem to improve it by 17%.Scalability of GP approaches has also been addressed using novel parallel hardware platforms. For example, Poli [14] distributes the GP population across multiple computers, while Langdon exploited the parallelism of GPGPU hardware [10].

5. INFORMATIVE VISUALIZATIONS

Performance vs accuracy trade-off space is the visualization technique used for Loop Perforation [15]. It involves plotting graphs for each perforation where the x coordinate is the percentage of accuracy loss and the y coordinate is the mean speedup of the perforation.

Two kinds of visualizations are used in [5] : the first one with generations on the x-axis and average fitness on the y-axis. These graphs depict how the average fitness changes over time in one genectic programming trial.

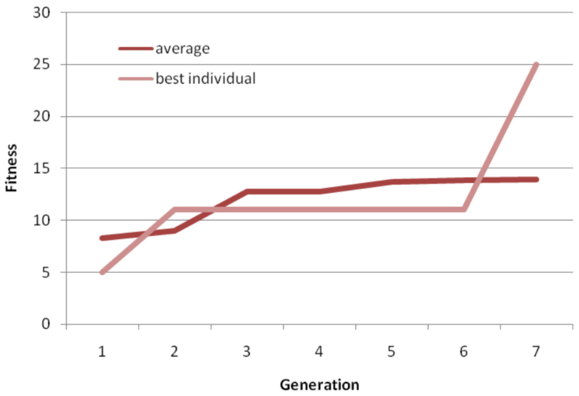
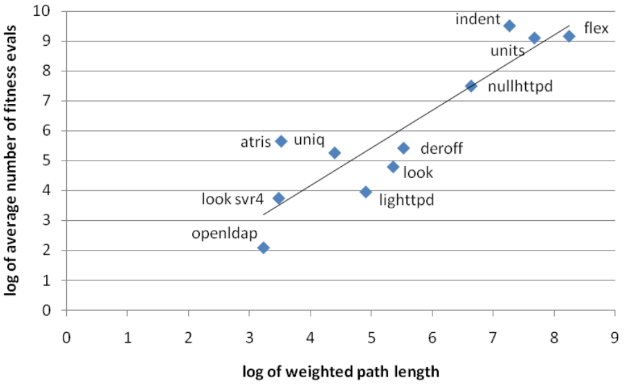
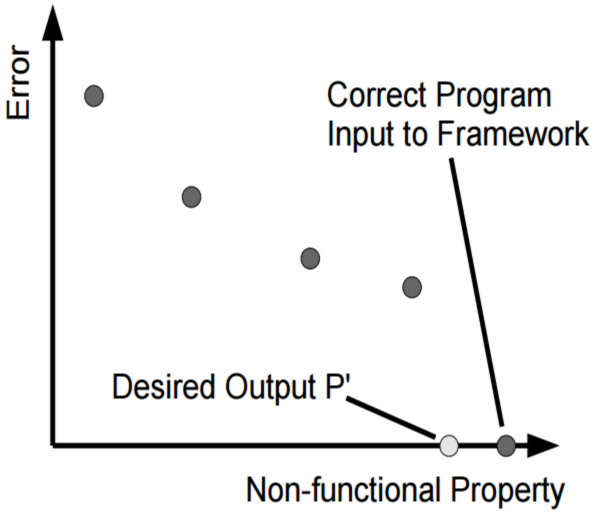


Figure 1: Evolution of the Zune bug repair for one successful GP trial. The darker curve plots the average fitness of the population, and the lighter curve plots the fitness of the individual *V* that becomes the primary repair.

In the second kind of visualization of [5], natural logarithm of weighted path length is on the x-axis and the natural logarithm of the total number of fitness evaluations performed before the primary repair is on the y-axis. This graph shows that genetic programming search time scales with execution path size.



In [1], visualization is done by plotting graphs for each genetic programming individual (program) where non-functional properties are along the x-axis and number of errors are along the y-axis.



In [12], the entire genetic programming improvement process of MiniSAT was presented.

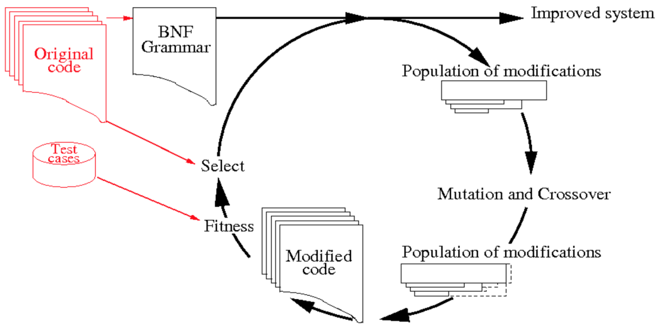


Fig. 1. GP improvement of MiniSAT.

The authors of [2] provide architectural diagrams for the proposed algorithm and approach. A graph flow for identifying organ; Mutation, Crossover, Validation and methods were visualized. The resulting code after transplantation was visualized by diff command, and highlighting modified sections.

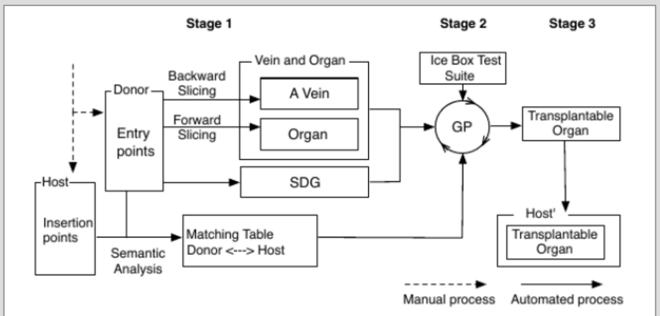


Figure 2: Overall architecture of μTrans; an SDG is a system dependency graph that the genetic programming phase uses to constrain its search space.

6. BASELINE RESULTS

A few papers that a part of this study had baseline results to compare against. The Genetically Improved MiniSAT solvers evolved in [12] are compared against four human written solvers :- MiniSAT (the original one), MiniSAT-best09 (winner of the MiniSAT-hack competition from 2009), MiniSAT-bestCIT (best performing solver from the competition when run on the CIT-specific benchmarks) and a hybrid of the above three called MiniSAT-best09+bestCIT.

Solver	Donor	Lines	Time
MiniSAT (original)	—	1.00	1.00
MiniSAT-best09	—	1.46	1.76
MiniSAT-bestCIT	—	0.72	0.87
MiniSAT-best09+bestCIT	—	1.26	1.63
MiniSAT-gp	best09	0.93	0.95
MiniSAT-gp	bestCIT	0.72	0.87
MiniSAT-gp	best09+bestCIT	0.94	0.96
MiniSAT-gp-combined	best09+bestCIT	**0.54**	**0.83**

The downstream applications, which use MiniSAT solver tested in [3] are CIT, Ensemble and AProVE. The modified and the unmodified versions are run 20 times and averaged for execution time. The fitness(and the correctness) of the solutions is also tested against the original versions and only those are measured which are correct.

In [2], the baseline results were time taken in porting tasks done by programmers. In the case study, it was estimated that over 11 years and 39 updates, it took an average of 20 days (of elapsed time) for VLC developers to port x264 encoder. The framework completed the same task in 26 hours (continuous time).

7. IMPROVEMENTS

The papers from Herman et. al, suffer from a common symptom of a need for more work in the optimization area of genetic algorithm. The softwares that are used for optimization are often hand selected for their desirable features and therefore results cannot be generalized. Techniques like Code Transplantation and Genetic Program repair exhibit a high degree of locality in terms of programming language, operating system, and platform being used. Automatic Test Case Generation is another challenge that needs to be addressed, as the quality of test cases decide the result, and degree of optimization [13].

In the paper [3]the mutations were only allowed from inside the program source code(which were typically 500 lines) based on the assumption that source code is generally redundant. This assumption might not be true when source code is very small. The optimizations could be seen as general programming bugs, having no strong co-relation to energy consumption.

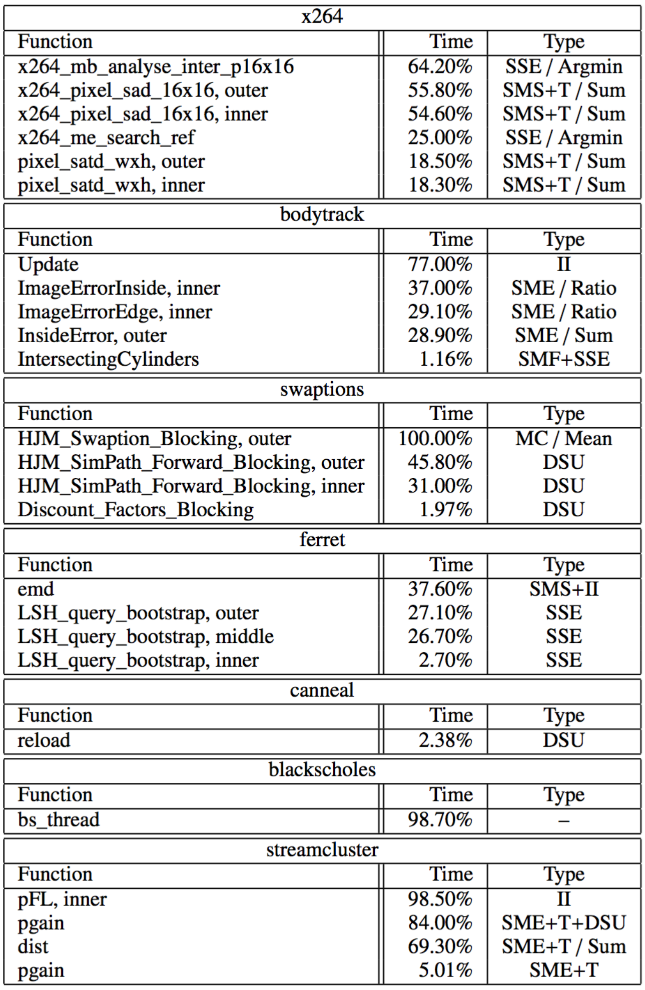
8. RESULTS

Co-evolutionary approaches have also been demonstrated to give performance improvements. The paper was able to demonstrate a marginal improvement in the running time of algorithms. Although the results were mostly obtained by removing extraneous assertions from the code, something easily achieved by a human coder, this is promising since it shows reasonable intelligence in identifying extraneous code.

Some interesting results of applying genectic improvement to MiniSAT [12]:

- The solver evolved using MiniSAT-best09 alone was not faster than the original MiniSAT solver.
- The solver evolved using the donor MiniSAT-bestCIT was as efficient as the donor itself, but was 13% faster than the original solver.
- Transplant from the both the above solvers resulted in a solution that was 4% faster than the original one, but had a lot of dead code, which was later removed by GP.
- A final âĂIJcombinedâĂİ solver that retained only changes that did not reduce performance or correctness perfomed 17% faster than the original solver and outperformed all human-written solvers by at least 4%.

GP has been successfully used to provide a layer of abstraction in the sense that developers do not have to provide multiple implementations for the same functionality. The trade-off space is automatically explored to find alternatives. There are several global computational patterns defined in the paper [15] as shown in the table which can be used to identify parts of an applications where loop perforation can be applied. Loop perforation, unlike autotuners [17, 6], does not explore alternatives with exact same accuracy, but with accuracy within a particular bound.



x264		
Function	Time	Type
x264_mb_analyse_inter_p16x16	64.20%	SSE / Argmin
x264_pixel_sad_16x16, outer	55.80%	SMS+T / Sum
x264_pixel_sad_16x16, inner	54.60%	SMS+T / Sum
x264_me_search_ref	25.00%	SSE / Argmin
pixel_satd_wxh, outer	18.50%	SMS+T / Sum
pixel_satd_wxh, inner	18.30%	SMS+T / Sum

bodytrack		
Function	Time	Type
Update	77.00%	II
ImageErrorInside, inner	37.00%	SME / Ratio
ImageErrorEdge, inner	29.10%	SME / Ratio
InsideError, outer	28.90%	SME / Sum
IntersectingCylinders	1.16%	SMF+SSE

swaptions		
Function	Time	Type
HJM_Swaption_Blocking, outer	100.00%	MC / Mean
HJM_SimPath_Forward_Blocking, outer	45.80%	DSU
HJM_SimPath_Forward_Blocking, inner	31.00%	DSU
Discount_Factors_Blocking	1.97%	DSU

ferret		
Function	Time	Type
emd	37.60%	SMS+II
LSH_query_bootstrap, outer	27.10%	SSE
LSH_query_bootstrap, middle	26.70%	SSE
LSH_query_bootstrap, inner	2.70%	SSE

canneal		
Function	Time	Type
reload	2.38%	DSU

blackscholes		
Function	Time	Type
bs_thread	98.70%	–

streamcluster		
Function	Time	Type
pFL, inner	98.50%	II
pgain	84.00%	SME+T+DSU
dist	69.30%	SME+T / Sum
pgain	5.01%	SME+T

6: Patterns in Pareto-optimal Perforations

Source Code optimization was an area of success for Genetic Programming, where it was able to detect extraneous statements, and removing resource heavy checks and assertions which had very low probability of being reached. We can conclude from this that sometimes its cheaper to fail than to aim for perfection, in cases where accuracy is not critical. In Code transplantation using GP, a wide variety of real world applications were successfully tested and used as donors and hosts for feature transplantation. The results were generalized as every donor was paired with every host, and even itself for sanity check. Automated Transplantation was shown to reduce the time and programmer workload on a big feature of real work application(VLC media player).

Recent work in optimization has shifted towards more Machine Learning techniques like Neural Networks, and this could be a reason in decrease in popularity of evolutionary techniques for optimization. However this can be a positive sign too as the increase in AI and Machine Learning in general can provide new ideas and interesting approaches to Evolutionary Programming.

9. FUTURE WORK

[7] provides a vision for automation techniques, for which research data from many sub-fields will be required. One such field is how functional requirements can be treated in the system as just another dimension in the non-functional requirement space, but with a higher priority. This can give us approximate solutions which can be many times as efficient as fully-correct solutions. Also, GISMOE relies heavily on the quality test data generation and testing methodologies, which would require future work.

For paper [5], future work includes: variation of repairs after minimization, comparison of repair quality to human-engineered solutions, necessity of crossover to the GP search, optimality of GP design, possible improvement of results by using a multi-objective fitness function, exploration of different parameter values, selection strategies, and operator design. In addition, there are also suggestion to explore differential weighting on the test cases and dynamic selection of test cases to be tested with the fitness function.

In [1], future work includes testing the results obtained in this paper for other problems, investigating optimal parameter settings, exploring alternative seeding strategies and using extended evolutionary runs.

For [12], finding ways to extend what can be modified so that the GP algorithm can make more significant changes, should be explored. Right now it can only reorder code and remove assertions which was discovered to be limiting. Decomposition has proved to be an important problem for work on GP [8, 16]

10. REFERENCES

[1] A. Arcuri, D. R. White, J. Clark, and X. Yao. *Multi-objective Improvement of Software Using Co-evolution and Smart Seeding*, pages 61–70. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[2] E. T. Barr, M. Harman, Y. Jia, A. Marginean, and J. Petke. Automated software transplantation. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ISSTA 2015, pages 257–269, New York, NY, USA, 2015. ACM.

[3] B. R. Bruce, J. Petke, and M. Harman. Reducing energy consumption using genetic improvement. In *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation*, GECCO '15, pages 1327–1334, New York, NY, USA, 2015. ACM.

[4] C. Cadar, K. Sen, P. Godefroid, N. Tillmann, S. Khurshid, W. Visser, and C. S. PÄČsÄČreanu. Symbolic execution for software testing in practice âĂŞ preliminary assessment. In *In ICSE ImpactâĂŹ11*, 2011.

[5] S. Forrest, T. Nguyen, W. Weimer, and C. Le Goues.

A genetic programming approach to automated software repair. In *Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation*, GECCO '09, pages 947–954, New York, NY, USA, 2009. ACM.

[6] M. Frigo and S. G. Johnson. Fftw: An adaptive software architecture for the fft. pages 1381–1384. IEEE, 1998.

[7] M. Harman, W. B. Langdon, Y. Jia, D. R. White, A. Arcuri, and J. A. Clark. The gismoe challenge: Constructing the pareto program surface using genetic programming to find better programs (keynote paper). In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ASE 2012, pages 1–14, New York, NY, USA, 2012. ACM.

[8] D. Jackson. Self-adaptive focusing of evolutionary effort in hierarchical genetic programming. In *Proceedings of the Eleventh Conference on Congress on Evolutionary Computation*, CEC'09, pages 1821–1828, Piscataway, NJ, USA, 2009. IEEE Press.

[9] K. Lakhotia, N. Tillmann, M. Harman, and J. D. Halleux. Flopsy- search-based floating point constraint solving for symbolic execution.

[10] W. B. Langdon and A. P. Harrison. Gp on spmd parallel graphics hardware for mega bioinformatics data mining.

[11] J. Petke, M. Harman, W. B. Langdon, and W. Weimer. *Using Genetic Improvement and Code Transplants to Specialise a C++ Program to a Problem Class*, pages 137–149. Springer Berlin Heidelberg, Berlin, Heidelberg, 2014.

[12] J. Petke, W. B. Langdon, and M. Harman. Applying genetic improvement to minisat. In *Proceedings of the 5th International Symposium on Search Based Software Engineering - Volume 8084*, SSBSE 2013, pages 257–262, New York, NY, USA, 2013. Springer-Verlag New York, Inc.

[13] H. Pohlheim and J. Wegener. Testing the temporal behavior of real-time software modules using extended evolutionary algorithms. In *Proceedings of the Genetic and Evolutionary Computation Conference*, page 1795. Morgan Kaufmann, 1999.

[14] R. Poli and J. Page. Solving high-order boolean parity problems with smooth uniform crossover, sub-machine code gp and demes, 2000.

[15] S. Sidiroglou-Douskos, S. Misailovic, H. Hoffmann, and M. Rinard. Managing performance vs. accuracy trade-offs with loop perforation. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, pages 124–134, New York, NY, USA, 2011. ACM.

[16] J. A. Walker and J. F. Miller. Evolution and acquisition of modules in cartesian genetic programming. In *In Proc. of the 7th European Conference on Genetic Programming, volume 3003 of LNCS*, pages 187–197. Springer-Verlag, 2004.

[17] R. C. Whaley and J. J. Dongarra. Automatically tuned linear algebra software. In *Proceedings of the 1998 ACM/IEEE Conference on Supercomputing*, SC '98, pages 1–27, Washington, DC, USA, 1998. IEEE Computer Society.

[18] J. Xiong, J. Johnson, R. Johnson, and D. Padua. Spl: A language and compiler for dsp algorithms. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, PLDI '01, pages 298–308, New York, NY, USA, 2001. ACM.

[19] S. Yoo, R. Nilsson, and M. Harman. Faster fault finding at google using multi objective regression test optimisation. Technical report, 2011.