Genetic Improvement for Approximate Computing

Lukas Sekanina and Zdenek Vasicek
Brno University of Technology, Faculty of Information Technology, IT4Innovations Centre of Excellence
Bozetechova 2, 61266 Brno, Czech Republic
Email: sekanina@fit.vutbr.cz, vasicek@fit.vutbr.cz

***Abstract*—This position paper connects the Genetic Improvement (GI) method, recently established in the search-based software engineering community, with approximate computing, in order to obtain improvements in the cases when errors in computations can be tolerated. It is argued that Genetic Improvement which shares many objectives with the approximate computing can be adopted to solve typical problems in the area of approximate computing.**

I. Introduction

Genetic programming (GP) is an artificial intelligence method capable of automated designing of programs in a given programming language [1]. GP has been used is *search-based software engineering* (SBSE) which applies metaheuristic search techniques to software engineering problems that can be formulated as optimization problems [2]. Conventional techniques of operations research such as linear programming are mostly impractical for complex software because of their high time complexity. Hence search methods are employed to provide "good enough solutions" in a reasonable time.

Genetic Improvement (GI) of software is in SBSE defined as the application of evolutionary and search-based optimization methods with the aim of improving functional and/or non-functional properties of existing software [3]. Contrasted to *approximate computing* that has been developed to improve energy efficiency and performance for the cost of accuracy, GI has always kept the code functionality identical (or even better) than the original software.

In this paper, we first briefly survey GI and approximate computing. It is then argued that GI can naturally be applied for approximate computing.

II. Genetic Improvement

Applications of GI were surveyed in [3]. For example, GI was employed for automatic bug fixing work [4] and improving of sorting algorithms [5], where it enabled to discover code optimization tricks probably unreachable by current compilers. Langdon and Harman showed that GI can, in addition to non-functional parameters, improve functionality of existing code (DNA sequence matching code) [6]. Such improvements can be expected in software which is processing large volumes of data using various heuristic procedures and trying to minimize an error metric. Genetic Improvement has also been used to create an improved version of C++ code from multiple versions of a program written by different domain experts. For example, some of MiniSAT implementation variants were evolved together to give a new MiniSAT tailored to solve interaction testing problems [7].

The *non-functional* properties of software (such as the execution time, power consumption and lines of code) can be improved by replacing the existing code fragments by newly evolved code fragments providing that they are semantically equivalent [5]. In a general case, deciding whether two code fragments are semantically equivalent is an undecidable problem. In practice, GI systems try to estimate the distance between the existing and candidate code fragments using data sets and a suitable fitness function. The original code serves as a "golden model".

Improving *functional properties* is possible only for a specific class of software. The main feature of this class is that the specification is in principle *incomplete* since a correct output is not defined for every legal input. Filters, classifiers, or predictors which are evaluated using big (image, video, speech) data sets fall into this class. An improvement in functionality is due discovering a better algorithm for particular data. On the other hand, when the specification is *complete* (accurate) and the implementation is correct, no improvements in functionality are possible. For example, there is no way to (functionally) improve a correct implementation of a Boolean function which is explicitly provided in the specification.

In the past 20 years, a very similar notion of genetic improvement has been developed in the field of *evolutionary circuit design*. Considering the fact that the majority of research in this area is performed using circuit simulators, where circuits are represented as software (using hardware description languages or netlits), we can speak about genetic improvement of specific software. In this context, the spectrum of non-functional properties is rich, including all key circuit parameters. A typical task for GI is to minimize the number of gates (the number of instructions in terms of software) of a fully functional combinational circuit. A considerable progress has recently been obtained by combining GP with methods of formal verification. GP can now solve the logic optimization problem even for complex instances (circuits with hundreds of inputs and thousands of gates) using a SAT-based functional equivalence checking embedded in the fitness function [8].

III. Approximate computing

One of the approximation techniques is *functional approximation* whose purpose is to implement a slightly different function to the original one provided that the error is acceptable and power consumption, performance or other parameters are improved. The functional approximation can be conducted at the level of software as well as hardware. An approximate solution is typically obtained by a heuristic procedure that modifies the original implementation. For example, artificial

neural networks were used to approximate software modules [9] and search-based methods allowed to approximate hardware components [10], [11].

In addition to functional approximations, timing induced approximations and components showing "unreliable" behavior (such as memory elements) are often employed. In the Chisel project, reliability- and accuracy-aware optimizations of computational kernels are performed by means of integer linear programming (ILP) and intended for approximate hardware platforms [12]. EnerJ [13] is an extension to Java that systematically supports approximate software development for approximate hardware. A specialized processor supporting approximate computing at the level of SW and HW was developed in [14].

IV. Towards Genetic Improvement for Approximate Computing

So far, GI has been used to improve functional and non-functional properties of software. However, by allowing errors in the fitness function of GP-based GI, one can easily obtain approximate solutions. In our previous work in this direction, we employed GP to approximate elementary circuits such as multipliers and median outputting circuits [15].

Applying the GI methodology for approximate computing (particularly for approximate software) seems to be straightforward. The main outcomes would be reducing the optimization time with respect to commonly used solvers (such as ILP in [12]) and obtaining better trade-offs among key system parameters (note that the search-based methods are not constrained by various assumptions of mathematically rigorous methodologies). The key advantage is that the GI systems can be constructed as multi-objective, i.e. they provide a Pareto front showing the best trade-offs among the error, speed, memory usage, energy consumption, network loading, etc., at the end of each run.

We can suggest that before introducing any approximations, the accurate HW/SW systems should firstly be optimized (e.g. by GI) in order to improve their functional (in the case of incomplete specified problems) as well as non-functional properties. The reason is GI/GP systems are often capable of improving existing solutions and hence the original systems could be improved in such a way that no intended approximations will be needed to reduce energy consumption and optimize other parameters.

Another advantage connected with GP and GI is that they can be executed in situ to dynamically adapt software as well as hardware as a response to a changing environment which can be represented by changing specifications, data patterns or hardware platforms (e.g. when it is operated with variable power budget).

Finally, to provide a complete picture, weak points of GI/GP methodology have to be mentioned. GP/GI can be time consuming and non-deterministic. Resulting solutions can be less trustworthy in terms of functionality. In order to obtain useful results, a GP expert providing insights into the problem encoding, genetic operators, fitness function and performance tuning has to be employed.

V. Conclusions

Software engineering and hardware engineering currently share a new challenge consisting in satisfying the requirements of energy efficiency. It has been argued in this paper that GI can naturally be used for purposes of software as well as hardware approximations. We believe that establishing a collaboration between the approximate computing community and genetic improvement community will provide new opportunities, insights and potential applications for both sides.

VI. Acknowledgments

This work was supported by the Czech science foundation project 14-04197S *Advanced Methods for Evolutionary Design of Complex Digital Circuits*.

References

[1] J. R. Koza, "Human-competitive results produced by genetic programming," *Genetic Prog. E. Machines*, vol. 11, no. 3–4, pp. 251–284, 2010.

[2] M. Harman and B. J. Jones, "Search-based software engineering," *Information and Software Technology*, vol. 43, pp. 833–839, 2001.

[3] W. B. Langdon, "Genetic improvement of software for multiple objectives," in *7th International Symposium on Search-Based Software Engineering, SSBSE*, ser. LNCS, vol. 9275. Springer, 2015, pp. 12–28.

[4] W. Weimer, S. Forrest, C. Le Goues, and T. Nguyen, "Automatic program repair with evolutionary computation," *Communications of the ACM*, vol. 53, no. 5, pp. 109–116, 2010.

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

[6] W. B. Langdon and M. Harman, "Optimizing existing software with genetic programming," *IEEE Trans. Evol. Comput.*, vol. 19, no. 1, pp. 118–135, 2015.

[7] 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," in *17th European Conference on Genetic Programming*, ser. LNCS, vol. 8599. Springer, 2014, pp. 137–149.

[8] Z. Vasicek and L. Sekanina, "Formal verification of candidate solutions for post-synthesis evolutionary optimization in evolvable hardware," *Genetic Prog. E. Machines*, vol. 12, no. 3, pp. 305–327, 2011.

[9] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger, "Neural acceleration for general-purpose approximate programs," *Commun. ACM*, vol. 58, no. 1, pp. 105–115, Dec. 2015.

[10] K. Nepal, Y. Li, R. I. Bahar, and S. Reda, "Abacus: A technique for automated behavioral synthesis of approximate computing circuits," in *Proceedings of the Conference on Design, Automation and Test in Europe*, ser. DATE '14. EDA Consortium, 2014, pp. 1–6.

[11] S. Venkataramani, A. Sabne, V. J. Kozhikkottu, K. Roy, and A. Raghunathan, "Salsa: systematic logic synthesis of approximate circuits," in *The 49th Annual Design Automation Conference, DAC '12*. ACM, 2012, pp. 796–801.

[12] S. Misailovic, M. Carbin, S. Achour, Z. Qi, and M. C. Rinard, "Chisel: Reliability- and accuracy-aware optimization of approximate computational kernels," in *Proc. of the 2014 Conference on Object Oriented Programming Systems Languages & Applications*. ACM, 2014, pp. 309–328.

[13] A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, and D. Grossman, "EnerJ: Approximate data types for safe and general low-power computation," in *Proc. of the 32nd ACM SIGPLAN Conf. on Prog. Language Design and Implementation*. ACM, 2011, pp. 164–174.

[14] V. Chippa, S. Venkataramani, S. Chakradhar, K. Roy, and A. Raghunathan, "Approximate computing: An integrated hardware approach," in *2013 Asilomar Conference on Signals, Systems and Computers*. IEEE, 2013, pp. 111–117.

[15] Z. Vasicek and L. Sekanina, "Evolutionary approach to approximate digital circuits design," *IEEE Trans. Evol. Comput.*, vol. 19, no. 3, pp. 432–444, 2015.