Automatic selection of compiler options using genetic techniques for embedded software design

Mena NAGIUB*, Wael FARAG**
* Valeo/Interior Switching and Controls, Smart Village, Egypt
** Valeo/Interior Switching and Controls, Smart Village, Egypt
*mena.nagiub@valeo.com
**wael.farag@valeo.com

Abstract— **ROM size and CPU load are considered as critical resources for the software design process of the embedded software. Thus it is necessary to produce software that follows specific ROM and CPU load requirements. Compiler options play major role in the optimization of code size and CPU load of the software. Selection of the best compiler option-set that provides the required code size and CPU load is a challenging process due to the wide range of options provided by modern compilers. In this paper we are providing a new technique that enables the designers to select automatically the best compiler options set that matches their design requirements based on genetic techniques. We have also added a new genetics operator called pass-over operator to enhance the chromosomes selection for the next generation.**

I. INTRODUCTION

In the domain of the embedded systems the ROM and the CPU load together are considered as critical factors for the design of the embedded software. As result the software requirements specification pay great attention for these factors and put strict specifications for them. Usually the compiler options selected to compile the software play major role in deciding the final ROM size and the CPU load of the final software product. Through experience it has been found that the compiler options are selected manually by some software experts to give the desired results. However, when it comes to complex software that includes many libraries it becomes more difficult to know the best compiler options suitable for the software, especially when the software has many sophisticated features. In this case, the designer of the software makes many trials to know the best compiler options those suit his product needs. This is usually due to the wide variety of options defined by the modern compilers, especially for some complex CPU architectures like ARM or Intel ATOM and the sophisticated compiler tool chains like GCC.

In this case, the designer chooses either to optimize the code according to the CPU load or the code size but rarely for both of them. This is also due to the fact that it is very difficult to optimize a code on both factors as the relation between them is inversely proportional. In our paper, we tackle the point of selection of compiler options in a defined way such that it allows the designer to have automatic method to select the ratio in the optimization between the ROM consumption and the CPU load of the final software product. In this solution an automated technique based on genetic algorithm concepts is used to select the options and it is going to take into consideration how the designer will be able to specify the ratio of optimization in CPU load and code size.

II. PREVIOUS AND RELATED WORK

This paper is going to add new value to compiler options selection based on genetic algorithms. Hence it is necessary to show the previous work. All the referenced papers here have represented the compiler options in the form of genes inside a chromosome. This chromosome is representing the compiler options set used to compile the benchmarking software. In these papers the authors have used the genetic algorithms operators (Crossover, Mutation, and Generation) on the chromosomes to add new members to the population. The main procedure has been to create initial population of chromosomes, compile the benchmark software using each chromosome, calculate the fitness function for the resulting executable, then apply the genetics operators on the old generation members to produce a new generation with enhanced features, and finally apply the previous steps again to the new chromosomes.

The main contribution among the papers has been either optimization of the CPU load only or the code size only. For example in Thayalan and Zakarias' paper [1] they have proposed optimization technique based on genetics techniques to provide optimization in terms of compilation and execution duration. They have used weighting technique to give a weight for each compiler option specifically based on the effect it causes on the execution and compilation durations. Their fitness function was the inverted sum of the execution duration and compilation duration. In Chang and Lin's paper [2] they used the same technique while their fitness function has been to get how many cycles consumed by the software during execution. Also Ivan and Boja [3] have shown that they have used the same fitness function (time consumed by the software during execution) for the evaluation of the chromosomes. On the other hand, Cooper and Schielke [4] have used the same genetic technique to select the compiler options sets but those ones would provide optimized code size in their goal to optimize the space for the embedded systems software.

III. PROBLEM ANALYSIS AND SOLUTION

A. Main problem

It is required to find a set of compiler options such that the output of the compilation process using these options is software with balanced optimization in terms of Code size and CPU load, according to the desired ratio between the two factors defined by the software designer. The process of compiler options selection should be in automatic manner and within the expected cost. The cost here is defined by the time frame needed to select these options. This kind of problem is classified as optimization problem for search, and hence it requires a search technique that can tackle optimization.

B. *Compiler options selection using genetic techniques*

Considering the case of using a compiler like GNU C compiler [10] for ARM processor, the possible compiler options which are affecting the code size and CPU load are 28 options. Equation 1 represents the possible combinations of options to be used for software compilation where C_i^n represents possible combinations for selecting (i) options for compilations from (n) possible options provided by the GCC compiler.

$$\sum_{i=0}^{i=n} C_i^n = \sum_{i=0}^{i=28} C_i^{28} = 2.68E+8 \qquad (1)$$

It means that there is a huge space of optimization compiler options sets. Genetic algorithms [5] techniques are subset of evolutionary search optimization algorithms techniques which can solve optimization problem where there are bounded set of variables and local minima.

Using genetic techniques, the variables which represent the compiler options are represented as genes in a chromosome. So the chromosome will represent the CFLAGS value used to compile the software (Following GNU compiler terms, the CFLAGS holds the compiler options set used to compile the software). At the first generation, the chromosomes are selected at random. Then the software is compiled using the generated chromosomes. After compilation each version of compiled software is executed and then the time spent by the software during execution (representing CPU load) and the size of the software itself (code size) are measured. The fitness function for the chromosomes is calculated based on the CPU load and the code size and using the genetics operators new chromosomes are generated for the next run from the old generation.

C. *The main algorithm for the compiler options selection*

The algorithm for the compiler options selection is described as following:

1. Generate the initial population of chromosomes.
2. Evaluate the fitness of each chromosome in the population.
3. Select two chromosomes CI and C2 from the population according to their fitness.
4. Perform a recombination on CI and C2 to generate two new chromosomes rCI and rC2.
5. Perform a mutation on rCI and rC2 to generate two new chromosomes mCI and mC2.
6. Evaluate the fitness of mCI and mC2.
7. Replace mCI and mC2 for chromosomes in the population according to their fitness to generate a new population.
8. If the end condition isn't satisfied, go to step 3.
9. Else, stop and return the chromosome with best fitness in the population.

Termination condition here becomes true when the enhancement in optimization in the new generation is negligible compared to the previous generation. Also there are two additional factors taken into consideration which are time needed in hours to compile the software using each new chromosome, and the cost of running the software on the embedded target used. These two factors have played important role in the introduction of a new operator called the pass-over operator. This operator means that the chromosome will pass as it is to the next generation. We have considered that chromosomes with highest fitness values are valuable and since they have provided good results we can keep them as they are without changes. So we select the top m chromosomes with highest fitness functions over certain threshold fitness $T_{fitness}$ to pass as they are to the next population. Such that at any generation if the selection process cost is exceeding the planned limits, for example by exceeding number of hours planned to select the best options, it will be possible to stop and get the best possible optimization results available.

The previously described factors also have helped in building a fitness function that depends on learning techniques. Equation 2 is built based on the analysis for several trails for tracking enhancement in the results of several generations:

$$CR_{fitness} = \frac{CZ_{Original}}{CZ_{Chromosome}} + \frac{CL_{Original}}{CL_{Chromosome}} \qquad (2)$$

Where $CR_{fitness}$ is the fitness of the chromosome, $CZ_{Original}$ is the code size of the original software, $CZ_{Chromosome}$ is the code size optimization of the software after using the compiler options defined by the chromosome, $CL_{Original}$ is the CPU load optimization of the original software, and $CL_{Chromosome}$ is the CPU load of the software after using the compiler options defined by the chromosome.

In addition since it is required to keep the optimization cost minimal, it will be necessary to monitor the optimization during each run. Hence in each generation R_i chromosomes are sorted in descending order according to their fitness value. And in the next generation R_{i+1} the following fitness function defined by equation 3 is used:

$$CR_{fitness} = \frac{CZ_{Best}}{CZ_{Chromosome}} + \frac{CL_{Best}}{CL_{Chromosome}} \qquad (3)$$

Where $CR_{fitness}$ is the fitness of the chromosome, CZ_{Best} is the best code size optimization of the software in the generation R_i, CL_{Best} is the best CPU load optimization of the software in generation R_i, $CZ_{Chromosome}$ code size optimization of the software after using the compiler options defined by the chromosome in the generation R_{i+1}, and $CL_{Chromosome}$ is the CPU Load optimization of the software after using the compiler options defined by the chromosome in generation R_{i+1}.

Based on equations (2) and (3) the original fitness function defined by equation 2 will be used for generation R_0 only because it is the start-up phase. Since it is required to keep the balanced optimization between the code size and CPU load of the software, new factor of the code size and CPU load will be added to the fitness function. The code size-CPU load factors will be represented in terms of percent so the code factor is defined as C% and hence the CPU load factor equals to (100-C%). As result, the final fitness function used is going to be defined using equation 4:

$$CR_{fitness} = (C\%)\left(\frac{CZ_{Best}}{CZ_{Chromosome}}\right) + (100 - C\%)\left(\frac{CL_{Best}}{CL_{Chromosome}}\right) \quad (4)$$

All of the chromosomes with fitness values below $T_{fitness}$ will undergo cross-over operator with probability of 100%. The mutation operator is applied on each gene in the chromosome with probability of $P_{mutation}$ which is constant during all the generations. The termination condition is either the optimization results is kept constant SAT_{margin} (saturation level within certain +/- margin) over three continuous generations or the time is exceeding the expected budget. The passed-over and the crossed-over chromosomes are all interested in the new generation.

IV. Experiments and Analysis

A. *Test Envrionment*

In our experiments we have selected to use DLib C++ library [6] as benchmarking algorithms. We have selected this library due to the following reasons:

- It provides many of the algorithms used currently in modern embedded systems like machine vision, networking, image processing, and data compression and integrity check. So we can test them on various applications as we are going to see soon.
- It is used by many trusted users and has well known publications.

Figure 1 show 10 trails to optimize manually software that performs kernel ridge regression classification algorithm provided by DLib test example on a set of 1000 features

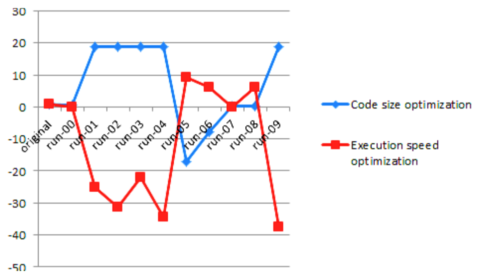

Figure 1. 10 manual trails to optimize the kernel ridge regression classification algorithm example

The software has run on Intel Atom N2000 [8], software has been compiled using GNU GCC compiler, and compiler options set provided by GNU have been used. 10 compilations have been performed, for each compilation a certain complier options set has been selected manually, the software has been compiled and tested its execution time and code size. As illustrated, the optimization has been impossible to be done on both code size and CPU load at the same time. It has been possible only to reach maximum optimization on one feature (code size or CPU load) per each trial.

Experiments have been done on a set of benchmarking algorithms provided by DLIB. We have selected the following algorithms:

- Kernel ridge regression classification algorithm (KRR).
- Custom binary classification trainer algorithm for the multiclass classification (CBC).
- Empirical kernel map algorithm (EKM).
- Support vector machine recognition algorithm (SVM).
- Multilayer perceptron algorithm (MLP).
- Rank features algorithm for supported vector machines (RFR).
- Pegasos algorithm for online training of support vector machines (SVM_PG).

The compiler used is GNU 4.63. It has 128 optimization compiler options, the OS family covers 100 of them and the remaining ones are used, so there are total of 28 options. Experiments have been done on Intel Atom N2000 on Aventech MIO-5250 embedded computer board [11]. Also some tests are done on ARM processor [7] on Beagleboard xM Rev.C [12]. The testing has been done on Julius Continuous Speech Recognition Engine [13] and using Mentor Graphics GNU GCC cross compiler Sourcery G++ Lite 2009q1-203 [9]. For all the algorithms a cost of 1 day has been selected. 10,000 chromosomes per generation have been created. The chromosomes in the initial generation are created randomly. The value of $P_{mutation}$ is 0.1. The value of $T_{fitness}$ is 90%. Selection of $(100 - T_{fitness})$ top chromosomes is selected for pass-over. So we are going to select 10% of top chromosomes for pass-over. SAT_{margin} factor has been selected to be 5% optimization.

B. Optimization results of Kernel ridge regression classification algorithm

For the Kernel ridge regression classification algorithm the value of C% is selected to be 50%. Figure 3 illustrates the results of optimization using the technique

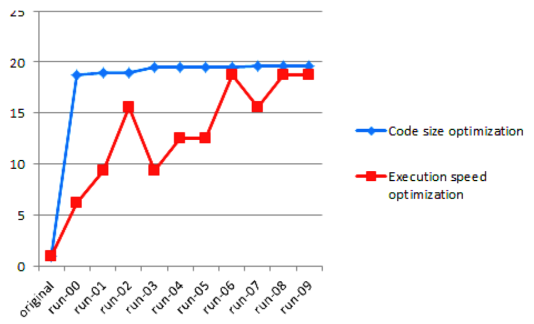

Figure 3. Optimization results of kernel ridge regression classification algorithm example after applying the technique

The execution of the technique has taken 1 day cost and generated 10 generations. This selection has reached 19.58% optimization in code size and 18.75% in CPU load.

C. Optimization results of the other algorithms

The following figures are results of optimization of the other algorithms on the Intel Atom processor with the value of C% is 50%

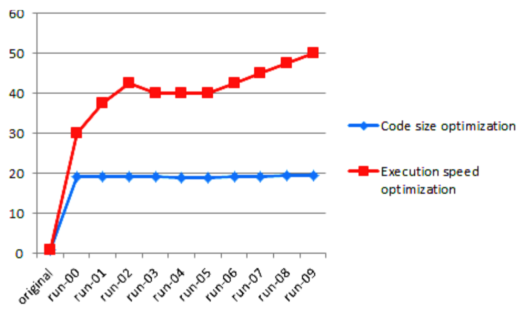

Figure 4. Optimization results of Empirical kernel map algorithm

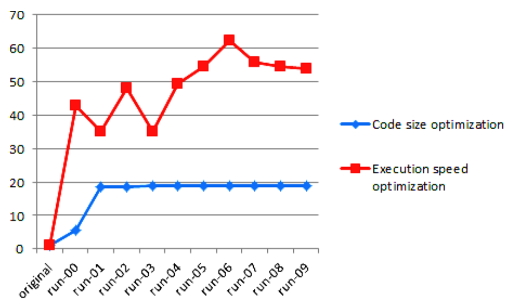

Figure 5. Optimization results of Support vector machine recognition algorithm

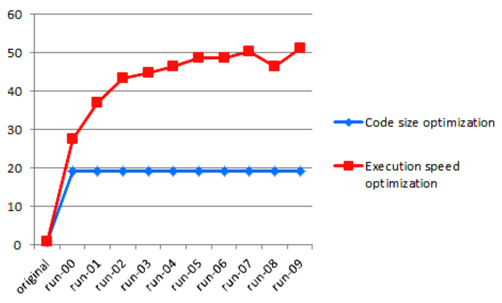

Figure 6. Optimization of Multilayer perceptron algorithm

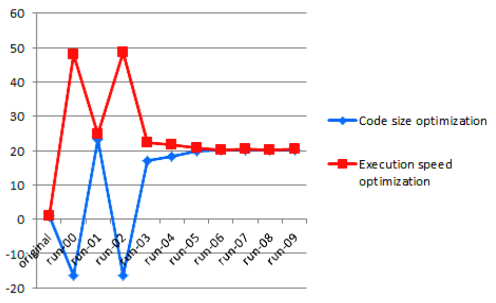

Figure 7. Optimization of Rank features algorithm for supported vector machines

D. Discussion about the results

As illustrated in the figures, the technique reaches the saturation level SAT_{margin} (the optimization results don't change within certain margin for 3 continuous generations) after about 100,000 chromosomes are being tested.

Table I summarizes the results of the optimization of the algorithms

TABLE I.
RESULTS FOR THE ALGORITHMS WHERE C% EQUALS 50%

Algorithm	Original Code size	Original Execution Speed	Optimization in code size	Optimization in execution speed
EKM	1.7 MB	0.4 sec.	19.39%	50.1%
SVM	1.8 MB	0.77 sec.	18.87%	53.98%
MLP	1.6 MB	1.27 sec.	19.29%	51.18%
RFR	1.7 MB	2.39 sec.	20.02%	20.01%

Notice that while the selected code size / CPU load is 50/50 not all results are giving the expected ratio. That is because the for example in the SVM case (Figure 5) the optimization of the code size has reached the saturation limits SAT_{margin} very early while the CPU load has kept to change in many generations till it has reached the saturation level SAT_{margin} in the 10th generation. Since the saturation here is calculated based on optimization of code size and CPU load 50/50 so the termination condition has been reached in the 9th run. While for example in the case of RFR algorithm (Figure 7) the result has been 50/50 as expected because both the code size and execution time have reached saturation SAT_{margin} together at the same time.

Experiments have been done also on CBC algorithm where value of C% is 30% as in Figure 8

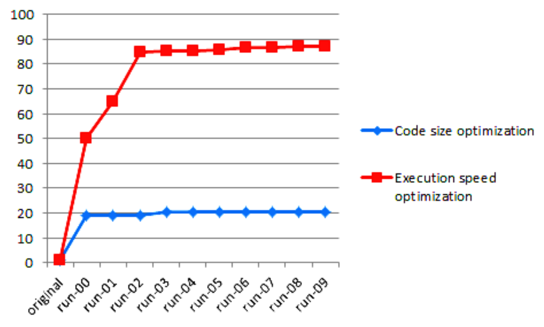

Figure 8. Optimization results of Custom binary classification trainer algorithm for the multiclass classification

In this case the optimization level reached 87% in CPU load while the code size optimization reached 20.1%. This is as expected. This experiment has been done on Intel atom and for 10 generations in 1 day.

Experiments have been done also on Julius engine on ARM core with Mentor GCC cross compiler with the value of C% is 70% as in Figure 9

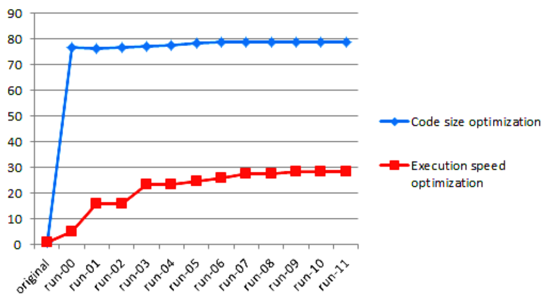

Figure 9. Optimization results of Julius continuous speech recognition engine

In this case the optimization level reached 78.76% in the code size while the CPU load optimization reached 28.36% matching the expectation.

E. *Analysis of optimization of code size versus CPU load*

To emphasize our work, special focus has been made on the case of the optimizations of the rank features algorithm for supported vector machines. In this case, when optimization coefficients have been chosen to be 50%-50% the optimization results have been oscillating. So the experiments have been repeated several times for different values of C%. To avoid the pitfall of the oscillation it has been chosen to test the software using 1,000,000 chromosomes for each C%. Hence for each run it has taken 2 weeks. For this experiment a normal PC has been used, with single processor composed of 4 cores. The results for the runs is summarized in the following table

TABLE II.
OPTIMIZATION RESULTS FOR RFR WITH DIFFERENT VALUES OF C%

Value of C%	Optimization in code size	Optimization in execution speed
90%	28.13%	2.1%
70%	27.47%	7.53%
30%	11.83%	31.38%
10%	1.83%	39.33%

As illustrated in the table, dramatic optimization in terms in code size hasn't been possible even with huge population space. Although in terms of CPU load good level of optimization has been reached compared to the previous results (we have reached double optimization in execution speed). Based on the results it has been clear that the original code was originally optimized in terms of code size and hence the applied effort has no major effect on results. Deeper analysis has been made for the functions and data structures used in RFR algorithm to make sure of this point. Also it has been found that the code is actually much optimized in terms of data structures and has used C++ techniques in efficient way to reduce code size. Since the code used efficient C++ techniques like template functions and inheritance, the code size is optimized but the execution speed is left without any extra effort for optimization. In the following diagrams there is a study we have made on the optimization results in little bit more details to emphasize our point of view.

For the case with C% equals to 90%, figure 10 was the result of running the technique

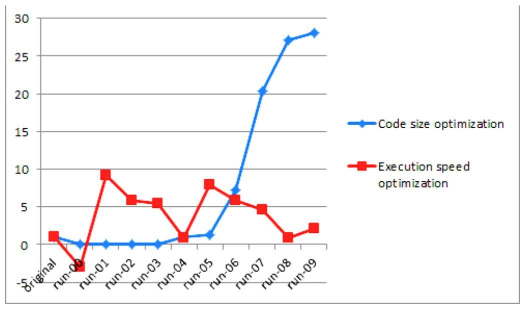

Figure 10. Optimization of Rank features algorithm for supported vector machines with C% equals 90%

In the graph there have been several local maximums where the code reaches balanced optimizations. At run 7 the optimization graph started to saturate in terms of code size to reach 28.1%. Compared with results of 50%-50% graph it is clear that the code size has reached maximum optimization.

Also for the case where C% equals to 70% figure 11 was the output of the optimization technique

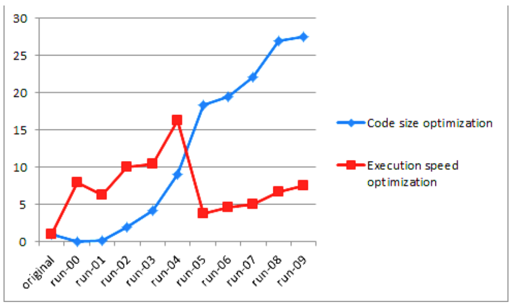

Figure 11. Optimization of Rank features algorithm for supported vector machines with C% equals 70%

This diagram has no local maximum however still there is the oscillation case between run 4 and run 5 even with large population.

However in the case of C% equals to 30% as in figure 12

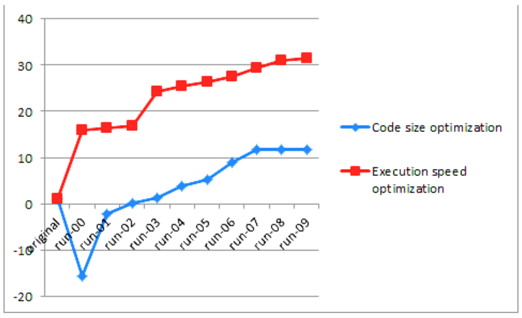

Figure 12. Optimization of Rank features algorithm for supported vector machines with C% equals 30%

There have been no oscillation case but at run 0 (the initial run) the code size reached negative optimization. Then the graph starts to grow normally again. However the optimization in the execution speed is not great.

The last graph value of C% is 10%. The figure 13 reflects precisely the fact that the code can be optimized in terms of CPU load

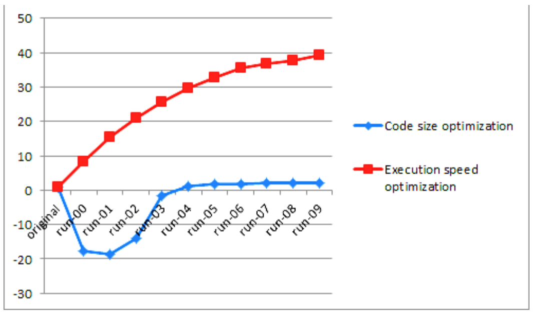

Figure 13. Optimization of Rank features algorithm for supported vector machines with C% equals 10%

As illustrated from the above analysis, for runs 0, 1, 2, and 3 the selected chromosomes have resulted in negative optimization of code size in favor of CPU load. After that the code size optimization reached very low optimization results saturated at 1% with maximum 1.83% where the CPU load results have kept growing until saturation point at value of 39%. This means that the code can no longer optimized in code size but in the speed. Hence using our technique it will be possible to know if the code needs really optimization in any of the two factors. Also it will be recommended to monitor the results of optimization during each run, to know if really the code needs optimization in a certain factor or not. Besides, trusting the technique results, designer can avoid wasting time trying to select options to optimize the code in the wrong direction.

Conclusion

The technique provides promising results which can help the designers to optimize their software automatically. Also the designer can select balanced ratio between the code size and CPU load according to their needs. Besides, the technique can help the designers also to know if the code is originally optimized in a certain factor and hence it can recommend to the designer to focus on optimizing the other factor using compiler options.

Future Work

The test bench used to apply the genetic algorithm will be used as a seed for automatic tool for compiler options selection. In addition to that algorithm is going to be enhanced to work on level functional blocks rather than whole software to enhance the results. Also the technique will be enhanced to provide the designer with information about the original state of the software and propose optimization factors for the designers.

Acknowledgement

As an engineer coming from the background of automotive and have good knowledge about AUTOSAR [14] standard and vision over the AUTOSAR coding style and examples, I would like to acknowledge AUTOSAR community to adopt that technique to be used for compiler options selection for the AUTOSAR code. Also I would like to acknowledge GenIVI [15] community for that tool to be used for development of Linux applications and projects. Also I would recommend GNU team to add the technique as a standalone tool in their tool chain.

References

[1] Thayalan S., M. N. B. Zakaria, A. J. Pal, A Genetic Algorithm Approach Towards Compiler Flag Selection Based on Compilation and Execution Duration, Universiti Teknologi PETRONAS, 2012

[2] S.C Lin, C.K. Chang, and N.W. Lin, Automatic Selection of GCC Optimization Options Using A Gene Weighted Genetic Algorithm, National Chung Cheng University, 2008

[3] I. IVAN, C. BOJA, M. VOCHIN, I. NITESCU, C. TOMA, and M. POPA, Using Genetic Algorithms in Software Optimization, Proceedings of the 6th WSEAS Int. Conference on TELECOMMUNICATIONS and INFORMATICS, Dallas, Texas, USA, March 22-24, 2007

[4] K. D. Cooper, P. J. Schielke, and D. Subramanian, Optimizing for Reduced Code Space using Genetic Algorithms, Department of Computer Science, Rice University, 2006

[5] D. Whitley, A Genetic Algorithm Tutorial, Computer Science Department Colorado State University

[6] DLib C++ Library, http://dlib.net/

[7] ARM technical reference manual, https://www.scss.tcd.ie/~waldroj/3d1/arm_arm.pdf

[8] Intel Atom Processor D2000, and N2000 series technical reference and user manual, https://wwwssl. intel.com/content/dam/doc/datasheet/atom-d2000-n2000-vol-2-datasheet.pdf

[9] Mentor Graphics Sourcery G++ compiler user manual for ARM architecture, GNU cross tool chain, https://sourcery.mentor.com/GNUToolchain/release858?lite=arm

[10] GNU Consortium, GCC Online Document, http://gcc.gnu.org/onlinedocs/

[11] Advantech MIO-5250 embedded computer board for N2600 intel atom, http://downloadt.advantech.com/ProductFile/PIS/MIO-5250/Product%20-%20Datasheet/MIO-5250_DS(06.11.12)20120621201706.pdf

[12] Beagleboard xM C embedded computer board for ARM, http://beagleboard.org/static/BBxMSRM_latest.pdf

[13] Julius Continuous Speech Recognition Engine, http://julius.sourceforge.jp/en_index.php.

[14] AUTOSAR (AUTmotive Open Software ARchiecture) Consortium http:// http://autosar.org/

[15] GenIVI Alliance (Geneva In-Vehicle-Infotainment) http://http://www.genivi.org/