Algorithms of the Combination of Compiler Optimization Options for Automatic Performance Tuning

Suprapto and Retantyo Wardoyo

Department of Computer Science and Electronics
Faculty of Mathematics and Natural Sciences,
Universitas Gadjah Mada Yogyakarta, Indonesia
{sprapto,rw}@ugm.ac.id

Abstract. It is very natural when people compile their programs, they would require a compiler that gives the best program performance. Even though today's compiler have reached the point in which they provide the users a large number of options, however, because of the unavailability of program input data and insufficient knowledge of the target architecture; it can still seriously limit the accuracy of compile-time performance models. Thus, the problem is how to choose the best combination of optimization options provided by compiler for a given program or program section. This gives rise the requirement of an orchestration algorithm that fast and effective to search for the best optimization combination for a program.

There have been several algorithms developed, such as Exhaustive Search (ES); Batch Elimination (BE); Iterative Elimination (IE); Combined Elimination (CE); Optimization Space Exploration (OSE); and Statistical Selection (SS). Based on those of algorithms, in this paper we proposed Heuristics Elimination (HE) algorithm, a simple algorithm that was mostly inspired by OSE with some differences. The HE algorithm uses a heuristic approach by applying genetic algorithm to find the best combination of compiler's optimization options. It is unlike OSE, however, this proposed algorithm starts from a set of some possible combinations randomly selected, then they are iteratively refined by some genetic operators to find one optimal combination (as the solution).

Keywords: Compiler optimization, optimization options, performance, orchestration algorithm, exhaustive search, batch elimination, iterative elimination, combined elimination, optimization space exploration, statistical selection.

1 Introduction

As we all know that optimizations of compiler for modern architectures have achieved high level of sophistication [11]. Although compiler optimizations have made a significant improvements in many programs, however, the potential for the degradation of performance in certain program patterns is still seen by compiler developer and many users. The state of the art is allowing the users deal

K. Mustofa et al. (Eds.): ICT-EurAsia 2013, LNCS 7804, pp. 91–100, 2013.

with this problem by providing them many compiler options. This compiler options's existence indicates that today's optimizers are not capable of making optimal choices at compile time. Moreover, the availability of input data of program is very minimum, and the lack of knowledge about the target architecture can limit the accuracy of compile-time performance models.

Therefore, the determination of the best combination of compiler optimizations for a given program or program section remains an unattainable compile-time goal. Today's compilers have evolved to the situation in which users are provided with a large number of options. For instance, GCC Compilers include 38 options, roughly grouped into three optimization levels, O1 through O3 [11]. On the other hand, compiler optimizations interact in unpredictable manners, as many have observed [2], [4], [10], [11], [8], [9]. Therefore, it is desired a fast and effective orchestration algorithm to search for the best optimization combination for a program.

Many automatic performance tuning systems have taken a dynamic, feedback-directed approach to orchestra compiler optimizations. In this approach, many different binary code versions generated under different experimental optimization combinations are being evaluated. The performance of these versions is compared using either measured execution times or profile-based estimates. Iteratively, the orchestration algorithms use this information to decide the next experimental optimization combinations, until converge criteria are reached [11].

2 Algorithms of Orchestration

In this section, we briefly present an overview of some algorithms that have goal finding an optimal combination of compiler's options. To do this, let we first define the goal of optimization orchestration as follows :

Given a set of compiler optimization options $\{F_1, F_2, \ldots, F_n\}$, where n is the number of optimization. Find the combination that minimizes the execution time of program efficiently, without using a priori knowledge of the optimization and their interactions.

2.1 Algorithm of Exhaustive Search

The exhaustive search (ES) approach, which is called the *factorial design* [2], [8], would try to evaluate every possible compiler's options in finding the best. This approach provides an upper bound of an application's performance after optimization orchestration. However, its complexity $O(2^n)$, which is prohibitive if it involves a large number of compiler's options. As an illustration, for GCC compiler with 38 options, it would take up to 2^{38} program runs – a million years is required for a program that runs in two minutes. Considering this fact, this algorithm will not be evaluated under the full set of options [11]. By the use of pseudo code, ES can be depicted as follows.

1. Get all 2^n combination of n compiler's options, $\{F_1, F_2, \ldots, F_n\}$.
2. For the optimized version compiled under every combination of n compiler's options, measure application execution time.

3. An optimal combination of compiler's options is one that give the smallest execution time to the program.

2.2 Algorithm of Batch Elimination

The idea of Batch Elimination (BE) is to identify the optimizations with negative effects and turn them off at once. BE achieves good program performance, when the compiler's options do not interact each other [11], [12], and [13]. The negative effect of one compiler's option, F_i can be represented by its *RIP* (*Relative Improvement Percentage*), $RIP(F_i)$, (see equation 1) which is the relative difference of the execution times of the two versions with and without F_i, that means $T(F_i = 1)$ and $T(F_i = 0)$ respectively ($F_i = 1$ means F_i is on, and $F_i = 0$ means F_i is off).

$$RIP(F_i) = \frac{T(F_i = 0) - T(F_i = 1)}{T(F_i = 1)} \times 100\% \tag{1}$$

The baseline of this approach switches on all compiler optimization options. $T(F_i = 1)$ is the execution time of the baseline T_B as shown in equation 2.

$$T_B = T(F_i = 1) = T(F_1 = 1, \ldots, F_n = 1) \tag{2}$$

The performance improvement by switching off F_i from the baseline B relative to the baseline performance can be calculated with equation 3.

$$RIP_B(F_i = 0) = \frac{T(F_i = 0) - T_B}{T_B} \times 100\% \tag{3}$$

If $RIP_B(F_i = 0) < 0$, the optimization of F_i has a negative effect. The BE algorithm eliminates the optimizations with negatives RIP in a batch to generate the final combination tuned version. The complexity of BE algorithm is $O(n)$.

1. Compile the application under the baseline $B = \{F_1, \ldots, F_n\}$. Execute the generated code version to obtain the baseline execution time T_B.
2. For each optimization F_i, switch it off from B and compile the application. Execute the generated version to obtain $T(F_i = 0)$, and compute $RIP_B(F_i = 0)$ according to equation 3.
3. Disable all optimizations with negative RIP to generate the final tuned version.

2.3 Algorithm of Iterative Elimination

Iterative Elimination (IE) algorithm was designed to consider the interaction of optimizations. Unlike BE algorithm, which turns off all optimizations with negative effects at once, IE algorithm iteratively turns off one optimization with the most negative effect at a time.

IE algorithm starts with the baseline that switches all compiler's optimization options **on** [11], and [7]. After computing the RIPs of the optimizations according to equation 3, IE switches the one optimization with the most negative effect **off** from the baseline. This process repeats with all remaining optimizations, until none of them causes performance degradation.

1. Let B be the combination of compiler optimization options for measuring the baseline execution time, T_B. Let the set S represent the optimization searching space. Initialize $S = \{F_1, \ldots, F_n\}$ and $B = \{F_1 = 1, \ldots, F_n = 1\}$.
2. Compile and execute the application under the baseline setting to obtain the baseline execution time T_B.
3. For each optimization option $F_i \in S$, switch F_i **off** from the baseline B and compile the application, execute the generated code version to obtain $T(F_i = 0)$, and compute the RIP of F_i relative to the baseline B, $RIP_B(F_i = 0)$, according to equation 3.
4. Find the optimization F_x with the most negative RIP. Remove F_x from S, and set F_x to 0 in B.
5. Repeat Steps 2, 3, and 4 until all options in S have non-negative RIPs. B represent the final option combination.

2.4 Algorithm of Combined Elimination

Combined Elimination (CE) algorithm combines the ideas of the two algorithms (BE and IE) just described [11], and [7]. It has a similar iterative structure as IE. In each iteration, however, CE applies the idea of BE : after identifying the optimization with negative effects (in this iteration), CE tries to eliminate these optimizations one by one in a greedy fashion.

Since IE considers the interaction of optimizations, it achieves better performance of program than BE does. When the interactions have only small effects, however, BE may perform close to IE in a faster way. Based on the way CE designed, it takes the advantages of both BE and IE. When the optimizations interact weakly, CE eliminates the optimizations with negative effects in one iteration, just like BE. Otherwise, CE eliminates them iteratively, like IE. As a result, CE achieves both good program performance and fast tuning speed.

1. Let B be the baseline option combination, and let the set S represent the optimization search space. Initialize $S = \{F_1, \ldots, F_n\}$ and $B = \{F_1 = 1, \ldots, F_n = 1\}$.
2. Compile and execute the application under baseline setting to obtain the baseline execution time T_B. Measure the $RIP_B(F_i = 0)$ of each optimization options F_i in S relative to the baseline B.
3. Let $X = \{X_1, \ldots, X_l\}$ be the set of optimization options F_i with negative RIPs. X is stored in increasing order, that is, the first element, X_1, has the most negative RIP. Remove X_1 from S, and set X_1 to 0 in the baseline B (in this step, B is updated by setting the optimization option with the most negative RIP to zero). For i from 2 to l,

- Measure the RIP of X_i relative to the baseline B.
- If the RIP of X_i is negative, remove X_i from S and set X_i to 0 in B.

4. Repeat Steps 2 and 3 until all options in S have non-negative RIPs. B represents the final solution.

2.5 Algorithm of Optimization Space Exploration

The basic idea of algorithm ***pruning*** is to iteratively find better combination of optimization options by merging the beneficial ones [9]. In each iteration, a new test set Ω is constructed by merging the combination of optimization options in the old test set using **union** operation. Then, after evaluating the combination of optimization options in Ω, the size of Ω is reduced to m by dropping the slowest combinations. The process repeats until the performance increase in the Ω set of two consecutive iteration become negligible. The specific steps are as follows :

1. Construct a set, Ω, which consists of the default optimization combination, and n combinations, each of which assigns a non-default value to a single optimization. In the experiment [11], the default optimization combination, O3,turns **on** all optimizations. The non-default value for each optimization is **off**.
2. Measure the application execution time for each optimization combination in Ω. Keep the m fastest combination in Ω, and remove the rest (i.e., n - m combinations).
3. Construct a new set of Ω, each element in which is a union of two optimization combinations in the old Ω. (The **union** operation takes non-default values of the options in both combinations.)
4. Repeats Steps 2 and 3, until no new combinations can be generated or the increase of the fastest version in Ω becomes negligible. The fastest version in the final Ω as the final version.

2.6 Algorithm of Statistical Selection

Statistical Selection (SS) algorithm uses a statistical method to identify the performance effect of the optimization options. The options with positive effects are turned **on**, while the one with negative effects are turned **off** in the final version, in an iterative fashion. This statistical method takes the interaction of optimization options into consideration.

The statistical method is based on orthogonal arrays (OA), which have been proposed as an efficient design of experiments [3], [1]. Formally, an OA is an $m \times k$ matrix of zeros and ones. Each column of the array corresponds to one compiler option, while each row of the array corresponds to one optimization combination. SS algorithm uses the OA with strength 2, that is, two arbitrary columns of the OA contain the patterns 00, 01, 10, 11 equally often. The experiments [11] used the OA with 38 options and 40 rows, which is constructed based on a *Hadamard* matrix taken from [6].

By a series of program runs, this SS algorithm identifies the options that have the largest effect on code performance. Then, it switches on/off those options with a large positive/negative effect. After iteratively applying the above solution to the options that have not been set, SS algorithm finds an optimal combination of optimization options. The pseudo code is as follows.

1. Compile the application with each row from orthogonal array A as the combination of compiler optimization options, and execute of the optimized version.
2. Compute the *relative effect*, $RE(F_i)$, of each option using equation 4 and 5, where $E(F_i)$ is the *main effect* of F_i, s is one row of A, $T(s)$ is the execution time of the version under s.

$$E(F_i) = \frac{(\sum_{s \in A:s_i=1} T(s) - \sum_{s \in A:s_i=0} T(s))^2}{m} \tag{4}$$

$$RE(F_i) = \frac{E(F_i)}{\sum_{j=1}^{k} E(F_j)} \times 100\% \tag{5}$$

3. If the relative effect of an option is greater than a threshold of 10%.
   - if the option has a positive *improvement*, $I(F_i) > 0$, according to equation 6, switch the option **on**,
   - else if the option has a negative *improvement*, switch the option **off**.

$$I(F_i) = \frac{\sum_{s \in A:s_i=0} T(s) - \sum_{s \in A:s_i=1} T(s)}{\sum_{s \in A:s_i=0} T(s)} \tag{6}$$

4. Construct a new orthogonal array A by dropping the columns corresponding to the options selected in the previous step.
5. Repeat all above steps until all of the options are set.

3 Algorithm of Heuristics Elimination

As mentioned in the previous section, *Heuristic Elimination* (HE) algorithm was mostly inspired by OSE algorithm; that is why the way it works is similar to OSE algorithm with some differences. HE algorithm iteratively find the combination of compiler's optimization options by applying heuristic approach using genetic algorithm.

The basic genetic algorithm is very generic and there many aspects that can be implemented very differently according to the problem [5]. For instance, representation of solution or chromosomes, type of encoding, selection strategy, type of crossover and mutation operators, etc. In practice, genetic algorithms are implemented by having arrays of bits or characters to represent the chromosomes. How to encode a solution of the problem into a chromosome is a key issue when using genetic algorithms. The individuals in the populations then go through a process of simulated evolution. Simple bit manipulation operations allow the

implementation of crossover, mutation and other operations. Individual for producing offspring are chosen using a selection strategy after evaluating the fitness value of each individual in the selection pool. Each individual in the selection pool receives a reproduction probability depending on its own fitness value and the fitness value of all other individuals in the selection pool. This fitness is used for the actual selection step afterwards. Some of the popular selection schemes are Roulette Wheel, Tournament, etc.

Crossover and **mutation** are two basic operators of genetic algorithm, and the performance of genetic algorithm very much depends on these genetic operators. Type and implementation of operators depends on encoding and also on the problem [5]. A new population is formed by selecting the fitter individual from the parent population and the offspring population (elitism). After several generations (iterations), the algorithm converges to the best individual, which hopefully represents an optimal or suboptimal solution to the problem.

Given the set of optimization options $\{F_1, F_2, \ldots, F_n\}$, there are exist 2^n possible combinations. It is unlike ES algorithm that evaluate every possible combination of optimization options, however, HE algorithm only needs to evaluate a certain number of combinations, and known as population size (m) of each generation, for instance $\{C_1, C_2, \ldots, C_m\}$. The optimal combination of optimization options is obtained by improving their fitness value iteratively with genetic operators in each generation (or iteration), and the combination with optimal fitness in the last generation will be final (an optimal) solution.

Chromosome that represents the solution of the problem (i.e., the combination of compiler's optimization options) is defined as one has fixed length n, and the value in each location indicates the participation (or involvement) of the compiler's option; which is formally defined in equation 7.

$$F_i = \begin{cases} 1 & \text{if } F_i \text{ involved} \\ 0 & \text{otherwise} \end{cases} \tag{7}$$

where, $F_i = 1$ for some i if the *i-th* compiler optimization option F_i is involved in the combination; otherwise $F_i = 0$.

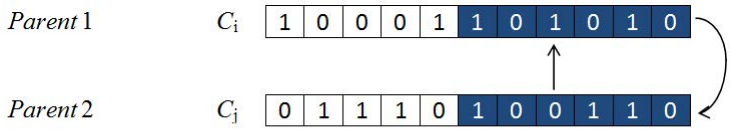
According to equation 7, then chromosome is encoded as a sequence of 0's or 1's (binary representation). For instance, 01100011110101 ($n = 14$) represents the combination of optimization options that involves respectively options number 2, 3, 7, 8, 9, and 10 in optimization. Since the binary representation used to represent the chromosome, the uniform (regular) crossover with either single point or more can be implemented depends on how many options compiler provides; while the mutation can be done by simply flipping the value in the single mutated location. In this case, the **fitness function** is defined by summing each $RIP_B(F_i = 0)$ (adopted from equation 3) in each C_j, and the formal definition is shown in equation 8.

$$Fitness(C_j) = \sum_{i=1}^{n} RIP_B(F_i | F_i = 0) \tag{8}$$

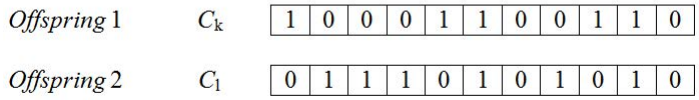
where C_j is one of the combination in population, and n is the number of optimization options $\{F_1, F_2, \ldots, F_n\}$.

Equation 3 says that, the optimization option of F_i has a negative effect when the value of $RIP_B(F_i = 0) < 0$. So that, according to the selection criteria, only the combinations with higher value of fitness function (i.e., fitter) will be considered as a parent candidate for next generation (iteration).

Having mentioned at the previous discussion, and by considering the representation of chromosome; the uniform (or regular) crossover with either single point or more would be implemented in the reproducing process of individual for the next generation. The process of crossover is illustrated in Fig. 1 and Fig. 2.

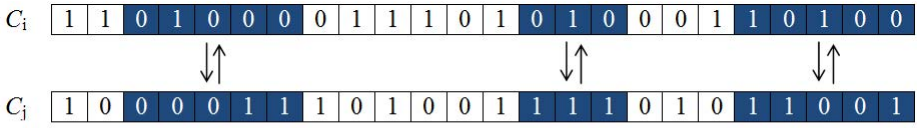


(a) The process of Regular Crossover Operation with single-point

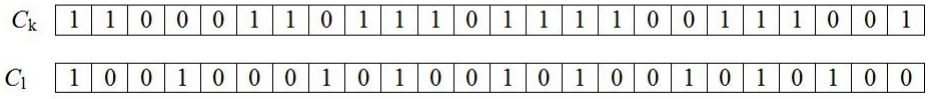


(b) The result of Regular Crossover Operation

Fig. 1. The operation of Regular Crossover with two-point



(a) The process of Regular Crossover Operation with five-point



(b) The result of Regular Crossover Operation with five-point

Fig. 2. The operation of Regular Crossover with five-point

It is unlike the crossover operator which is binary, mutation is unary. First, the mutated location is determined randomly, and the value of that location is then replaced by only flipping the value from '0' to '1' and vice versa. The process of mutation is shown in Fig. 3.

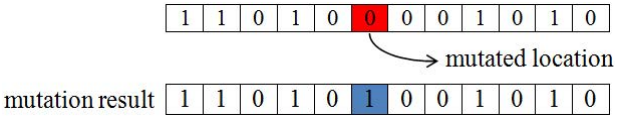


Fig. 3. The operation of mutation operator

The pseudo code of HE algorithm is as follows.

1. Determine genetics's parameters, i.e., the size of population, the probability of crossover, and the probability of mutation respectively m, p_c, and p_m,
2. Generate the initial population (collection of the combination of optimization options) randomly, $P = \{C_1, \ldots, C_m\}$, with $C_i = \{F_1 = 0 \text{ or } 1, \ldots, F_n = 0 \text{ or } 1\}$,
3. Compute the value of fitness of each chromosome C_j using equation 8,
4. Based on fitness values computed in the previous step, and certain selection method (for instance Roulette wheel selection), select chromosome as parent candidates,
5. Check the termination condition, if the condition is false, then do step (6); otherwise STOP.
6. Crossover the parent by considering the value of p_c, to yield new chromosomes (*offspring*) as many as the population size for the next generation, go to steps (3) - (5).

Note that the termination condition could be either the determined number of generations (iterations) or some determined value of threshold as an indicator of its convergence.

4 Conclusion

In accordance to the way the algorithm find the best combination, HE algorithm is only relevant to be compared with ES, BE and OSE algorithms. The following are some remarks about that comparison.

- As the name implies, ES algorithm finds the best combination of the compiler's optimization options by exhaustively checking all possible ones.
- The result of BE algorithm is an optimal combination of the compiler's optimization options obtained by removing optimization option with the most negative RIP iteratively.
- OSE algorithm build the combination of compiler optimization options starting from the set with single default optimization combination, then iteratively the set is updated by performing the union operation.
- HE algorithm finds the optimal solution (combination of compiler's optimization options) starting from the initial population contains a number of combinations of compiler's optimization options which were initially chosen

in random manner. Then, each chromosomes (representation of combinations) in the population were evaluated using fitness function to determine fitter chromosomes to be chosen for the next generation. This process was performed iteratively until some determined condition satisfied, and the best combination is obtained.

References

1. Hedayat, A., Sloane, N., Stufken, J.: Orthogonal Arrays: Theory and Applications. Springer (1999)
2. Chow, K., Wu, Y.: Feedback-directed selection and characterization of compiler optimizations. In: Second Workshop on Feedback Directed Optimizations, Israel (November 1999)
3. Box, G.E.P., Hunter, W.G., Hunter, J.S.: Statistics for Experimenters: an introduction to design, data analysis, and model building. John Wiley and Sons (1978)
4. Chow, K., Wu, Y.: Feedback-directed selection and characterization of compiler optimizations. In: Second workshop of Feedback-directed Optimizations, Israel (November 1999)
5. Nadia, N., Ajith, A., Luiza de Macedo, M.: Genetic Systems Programming - Theory and Experience. Springer (2006)
6. Sloane, N.J.A.: A Library of Orthogonal Arrays, `http://www.research.att.com/njas/oadir/`
7. Kulkarni, P., Hines, S., Hiser, J., Whalley, D., Davidson, J., Jones, D.: Fast Searches for Effective Optimization Phase Sequences. In: PLDI 2004: Proceeding of the ACM SIGPLAN 2004 Conference of Programming Language Design and Implementation, pp. 171–182. ACM Press, New York (2004)
8. Pinkers, R.P.J., Knijnenburg, P.M.W., Haneda, M., Wijshoff, H.A.G.: Statistical selection of compiler optimizations. In: The IEEE Computer Societies 12th Annual International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems (MAS - COTS 2004), Volendam, The Netherlands, pp. 494–501 (October 2004)
9. Triantafillis, S., Vacharajani, M., Vacharajani, N., August, D.I.: Compiler Optimization-space Exploration. In: Proceedings of the International Symposium on Code generation and Optimization, pp. 204–215 (2003)
10. Kisuki, T., Knijnenburg, P.M.W., O'Boyle, M.F.P., Bodin, F., Wijshoff, H.A.G.: A Feasibility Study in Iterative Compilation. In: Fukuda, A., Joe, K., Polychronopoulos, C.D. (eds.) ISHPC 1999. LNCS, vol. 1615, pp. 121–132. Springer, Heidelberg (1999)
11. Pan, Z., Eigenmann, R.: Compiler Optimization Orchestration for peak performance. Technical Report TR-ECE-04-01. School of Electrical and Computer Engineering, Purdue University (2004)
12. Pan, Z., Eigenmann, R.: Rating Compiler Optimizations for automatic performance tuning. In: SC 2004: High Performance Computing, Networking and Storage Conference, 10 pages (November 2004)
13. Pan, Z., Eigenmann, R.: Rating Compiler Optimizations for Automatic Performance Tuning. IEEE (2004)