Improved Batch Elimination: A Fast Algorithm to Identify and Remove Harmful Compiler Optimizations

Ewerton Daniel de Lima
Departament of Informatic
State University of Maringá
Maringá, Paraná, Brazil
Email: ewertondanieldelima@gmail.com

Anderson Faustino da Silva
Departament of Informatic
State University of Maringá
Maringá, Paraná, Brazil
Email: anderson@din.uem.br

***Abstract*—Modern compilers provide several optimizations that can be applied to the source code, in order to increase its performance. Due to the complex relationship between various optimizations, discovering harmful compiler optimizations is a problem in the context of compilers. Strategies based on iterative compilation try to solve this problem evaluating the performance of the compiled program using different sets. In this context, Combined Elimination is an efficient iterative compilation strategy. The purpose of Combined Elimination is to identify the harmful optimizations and remove them in an iterative compilation process. Combined Elimination provides good results, which are close to those founded by an exhaustive search approach. However, its drawback is the number of program runs. In this paper, we proposed an iterative compilation algorithm, named Improved Batch Elimination. This algorithm is based on the first step towards Combined Elimination, the Batch Elimination algorithm. The goal of Improved Batch Elimination is to produce results similar to Combined Elimination, with a complexity similar to Batch Elimination. In other words, the goal is to produce good results and to be faster than Combined Elimination. We evaluate our algorithm by measuring the performance of** SPEC CPU2006**,** POLYBENCH **and** CBENCH **benchmarks under a set of** 63 `LLVM` **compiler optimizations. The results indicate that Improved Batch Elimination is a good strategy to remove harmful compiler optimizations, using few program runs.**

I. INTRODUCTION

Iterative compilation is a well-known strategy used to discover the best compiler optimization set to compile a specific program, as well as to remove harmful optimizations from a compiler optimization set [1], [2], [3], [4], [5]. In this approach, the program is compiled with different compiler optimizations sets, and the best version is chosen. Due to the diversity of sets and the need to compile and run the program several times, iterative algorithms try to cover the search space selectively. Based on the behaviour of the search, the algorithms can be classified into three categories: partial search algorithms, random algorithm or genetic algorithms.

Partial search algorithms try to analyze a portion of all possible solutions to find an optimization set that provides the best performance for the test program [5], [6], [7], [8], [9], [10], [11]. Random or statistical algorithms perform the search employing statistical and randomization techniques, in order to reduce the number of sets evaluated [12], [13], [14]. Genetic algorithms use random searches based on a set of transformations to find a good optimization set [15], [16], [17], [18], [19]. Besides, there is research that combines several strategies [20].

In order to reduce the required program evaluations by iterative compilations, machine learning techniques create in an offline phase a prediction model, which will be used to determine the compiler optimization set that should be used on a test (unseen) program [4], [20], [21], [22], [23], [24], [25], [26], [27], by the online phase. The main advantage of this technique is that it reduces the number of required program runs (evaluations), besides outperforms iterative compilation strategies.

In the context of machine learning, the offline phase collects pieces of information about a set of training programs and downsamples the search space in order to provide a small space, which can be handled by the online phase in a easy and fast way. Based on the downsampled space and the pieces of information, the offline phase creates a collection of previous cases (the prediction model), which will be used to solve an unseen case, i.e., to determine the compiler optimization set that should be enabled by the compiler during the final code generation. Online phase will infer from the cases provided by the offline phase, the best compiler optimization set that fits the features of the unseen program as defined by its input.

After collecting pieces of information about a set of training programs, it is necessary to improve the quality of the prediction model. Due to each information (compiler optimization set) in the model can contain optimizations that do not contribute to the program speedup or have a negative impact on the program speedup (due to false interactions with other optimizations). Therefore, we need to eliminate these unnecessary optimizations.

In this context, an interesting algorithm is the work of Pan and Eigenmann [5]. They have proposed an algorithm, which identifies and removes harmful optimizations in an iterative compilation process. In their work, Pan and Eigenman presented three algorithms: Batch Elimination, Iterative Elimination, and Combined Elimination. The first two algorithms were steps towards Combined Elimination.

In their research, the authors showed that Combined Elimination is efficient to remove harmful optimizations, and the results are similar to an exhaustive search approach. However, this algorithm complexity is $O(n^2)$. It means that Combined Elimination can take up to n^2 program runs (evaluations).

In this paper, we proposed an algorithm based on Batch Elimination, which provides results similar to Combined Elimination, named Improved Batch Elimination. Our algorithm identifies and removes harmful optimizations in batch similar to Batch Elimination.

We compare our algorithm with the three algorithms proposed by Pan and Eigenmann on a set of benchmarks. For this purpose, we use all SPEC CPU2006 benchmarks written in C and C++ [28][29], all POLYBENCH benchmarks [30], and all CBENCH benchmarks [31].

Using all LLVM [32] O2 optimizations, Improved Batch Elimination is able to remove harmful optimizations, and discover an optimization set that outperforms O2 on several programs. Besides, Improved Batch Elimination outperforms Pan and Eigenmann's algorithms on several programs.

The remainder of this paper is organized as follows. Section II describes the algorithms proposed by Pan and Eigenmann. Section III describes the Improved Batch Elimination algorithm. Section IV describes the experimental setup. Section V presents the experimental results. Finally, the Section VI presents the concluding remarks.

II. The Pan and Eigenman's Algorithms

Pan and Eigenmann [5] have proposed three algorithms to find efficient compiler optimization sets: Batch Elimination (BE), Iterative Elimination (IE), and Combined Elimination (CE). The first two algorithms were steps towards CE. The main objective of CE is to discover and remove harmful optimizations from a baseline set.

The BE compiles the input program p using the baseline set, runs the program, and saves its performance P_b (in this case, the runtime). After that, BE disables each optimization i ($i = 1..n$) from the baseline, one by one, compiles p using each generated subset, and finally saves its performance P_{si}. The final compiler optimizations set is formed by the baseline, with all optimizations i where $P_{si} < P_b$ disabled.

A limitation of BE is that it does not consider the effect of interactions between the optimizations. The IE attempts to address this limitation by disabling, on each step, only the optimization that has the worst performance. Initially, the performance using the baseline (P_b) and the subsets without each optimization i (P_{si}) are obtained like BE. However, only the optimization i with the worst performance P_{si} is removed. This set of size $n-1$ is now the new baseline. In the next step, IE tries to discover the worst optimization based on this new baseline. This iterative process finishes when does not exist harmful optimization in the baseline.

CE is a combination of the previous two algorithms. CE has an iterative structure like IE. However in each iteration, CE tries to eliminate several optimizations that produce negative effects, like BE. CE evaluates the performance of the baseline without each optimization. After that, the algorithm creates an ordered list of all harmful optimizations. The optimization with the worst performance, which is the first on the list, is removed from the baseline, and the baseline is updated. The new baseline is evaluated without each remaining optimization of the list and all harmful optimizations are removed. The whole iterative process finishes when does not exist harmful optimizations in the baseline, as in IE.

III. The Improved Batch Elimination Algorithm

The BE uses an iterative process wherein each iteration it tries to identify and remove harmful optimizations in batch. The attractiveness of BE is to reduce the number of program runs, removing the harmful optimization in only one step. However, the disadvantage is the potential performance degradation.

In this paper, we show that is possible to remove harmful optimizations in batch in only one step similar to BE, deciding which optimization to remove in a way similar to CE.

We proposed the Improved Batch Elimination (IBE) algorithm, an efficient orchestration of compiler optimizations. The main objective is to propose an algorithm that has performance similar to CE, besides a complexity similar to BE.

The IBE is an improved version of BE, which uses a different strategy to remove harmful optimizations. While BE remove all harmful optimizations in batch, based on the performance, IBE does the same, however based on a harmful factor.

In IBE, each optimization that has negative performance has a harmful factor. Intuitively, an optimization can be more harmful than another one. Thus, the IBE uses this assumption to remove only the most harmful optimizations, and not all harmful optimizations like BE. The harmful factor considers how much each optimization impact negatively the program. In IBE, the probability of a given optimizations to be removed is proportional to its harmful factor.

The harmful factor of optimization is 0, if the performance of the baseline with this optimization enabled is better than that with this optimization disabled. Otherwise, the factor is the runtime of the program, when compiled with the baseline without this optimization.

Based on the harmful factor, IBE calculates the probability of an optimization to be removed. The probability of the most harmful optimization is 1, and the others harmful optimizations will receive proportional probabilities.

At this point, there is an important issue that has to be addressed. Based on the probabilities, what optimizations should be removed.

It is possible to generate a random number and remove all optimization with probability greater than or equal to this random number. However, it is easy to realize that it is necessary several iterations to identify the best optimization subset to remove from the baseline.

Considering that the *random algorithm* generates a different random number in each iteration, and some number indicates the best optimization subset to remove in c iterations. If $c \to \infty$ is possible to find a point in the interval $[0, 1]$ that provides the best result that can be searched by the *random algorithm*.

Based on this assumption, the problem can be reduced to a simple search through this point. The IBE search this point evaluating equidistant points in the interval $[0, 1]$, according to the number of iterations. Therefore, with c iterations, the distance between these points is given by:

$$interval = \frac{1}{c-1}$$

The *interval* value indicates that the distance between the points is inversely proportional to the number of iteractions; consequently there will be more points throughout the range. The IBE is described in detail in Algorithm 1.

Algorithm 1: *Improved Batch Elimination*

```
Input: Program (P); Compiler Optimization Set (B);
       Points (C)
Output: The best compiler optimization set (best_set)
$len_B \leftarrow |B|$
$best\_set \leftarrow B$
$best\_time, time_B \leftarrow Execute(P, B)$ /* Compile P
with the optimizations set B, and execute P */
$factor \leftarrow [\,]$
$prob \leftarrow [\,]$
for $i \leftarrow 1$ to $len_B$ do
    prob.append(0)
for $i \leftarrow 1$ to $len_B$ do
    Turn on all optimizations in $B$
    Turn off the optimization $i$ in $B$
    $time_i \leftarrow Execute(P, B)$
    if $time_i < time_B$ then
        $factor[i] \leftarrow time_i$
    else
        $factor[i] \leftarrow 0.0$
$M \leftarrow max(factor)$
if $M = 0$ then
    return $B$
for $i \leftarrow 1$ to $len_B$ do
    if $factor[i] > 0.0$ then
        $prob[i] \leftarrow \frac{factor[i]}{M}$
    else
        $prob[i] \leftarrow 0.0$
$R \leftarrow 0.0$
$interval \leftarrow \frac{1}{C-1}$
for $j \leftarrow 1$ to $C$ do
    $conj \leftarrow B$
    for each harmful optimization $i$ do
        if $prob[i] \geq R$ then
            Turn off the optimization $i$ in $conj$
    if conj not evaluated then
        $time_t \leftarrow Execute(P, conj)$
        if $time_t < best\_time$ then
            $best\_time \leftarrow time_t$
            $best\_set \leftarrow set$
    $R \leftarrow R + interval$
return best_set
```

The IBE can be summarized as follows. Initialy, IBE compiles the input program p using the baseline set, runs the program, and saves its performance P_b. After that, IBE disables each optimization i ($i = 1..n$) from the baseline, one by one, compiles p using each generated subset, and finally saves its performance P_{si}. After that, IBE analyzes c points in the interval $[0, 1]$ to discover what is the best optimization subset to remove from the baseline.

We will show that IBE has a short tuning time, while achieving comparable or better performance than CE.

IV. The Experimental Setup

The Platform The experiments are carried out on an Intel x86_64 based machine, supporting a Core I7-3770 processors running at 3.4GHz with an I&D L1, L2, L3 cache and RAM of 32K, 256K, 8M and 4GB, respectively. The operating system on the machine was Ubuntu, running kernel 3.11.0-15-generic.

The Benchmark To evaluate the proposed solution, we use all SPEC CPU2006 benchmarks [28] [29] written in C and C++ with dataset train, all POLYBENCH benchmarks [30] large dataset, and all CBENCH benchmarks [31] with dataset 1.

The Optimizations The experiments are conducted using all LLVM version 3.4 [33] O2 optimizations. This compiler optimization level is the baseline, based on several experiments that indicated it achieves the best overall performance for several benchmarks. The Figure 1 presents the LLVM O2 optimizations.

-targetlibinfo -no-aa -tbaa -basicaa -notti -globalopt -ipsccp -deadargelim -instcombine -simplifycfg -basiccg -prune-eh -inline-cost -inline -functionattrs -sroa -domtree -early-cse -lazy-value-info -jump-threading -correlated-propagation -simplifycfg -instcombine -tailcallelim -simplifycfg -reassociate -domtree -loops -loop-simplify -lcssa -loop-rotate -licm -lcssa -loop-unswitch -instcombine -scalar-evolution -loop-simplify -lcssa -indvars -loop-idiom -loop-deletion -loop-unroll -memdep -gvn -memdep -memcpyopt -sccp -instcombine -lazy-value-info -jump-threading -correlated-propagation -domtree -memdep -dse -adce -simplifycfg -instcombine -strip-dead-prototypes -globaldce -constmerge -preverify -domtree -verify

Fig. 1. LLVM 3.4 O2 optimizations.

The Compilation Framework The compilation process for each program represents the basic cycle performing in all experiments. This process was accomplished with the LLVM version 3.4, which is formed by a collection of tools.

The *clang* generates the LLVM's intermediate representation without the use of any optimization. After, the *opt* transforms the code by applying a compiler optimizations set. Then, the *llc* generates from the optimized version, the target assembly code. Finally, the tools *as* and *ld* generate the executable.

The IBE Parameters There is only one parameter for IBE, which is the number of points to evaluate c. We chose to evaluate four values, namely: 5, 10, 20, and 30.

The Validation The validation of the results is based on the average of several executions for each program's instance. In the experiments, the machine workload was minimum as possible, in other words, each instance was executed sequential. Besides, the machine did not have external interference, and the standard deviation of the results was less close to 0.0.

Metrics The experimental evaluation use two metrics to evaluate the behavior of our algorithm.

1) Improvement: this metric indicates the program performance, when the program is compiled with the best optimizations set founded by the algorithm. The improvement is calculated as follows:

$$Speedup = \frac{baseline\_runtime}{new\_runtime}$$

$$Improvement = ((Speedup) - 1) * 100$$

2) Evaluations: this metric indicates the number of program runs performed by the algorithm.

V. The Experimental Evaluation

We compare our algorithm, `IBE`, with the three algorithms developed by Pan and Eigenmann, namely: `BE`, `IE` and `CE`. It is important to remember that `BE` and `IE` are steps towards `CE`. We want to show that it is possible to have good results using an algorithm which complexity is close to $O(n)$.

A. Benchmark Performance

The Figures 2, 3 and 4 show the results of `BE`, `IE`, `CE` and `IBE` in terms of improvement for Spec Cpu2006, Polybench and cBench benchmarks, respectively.

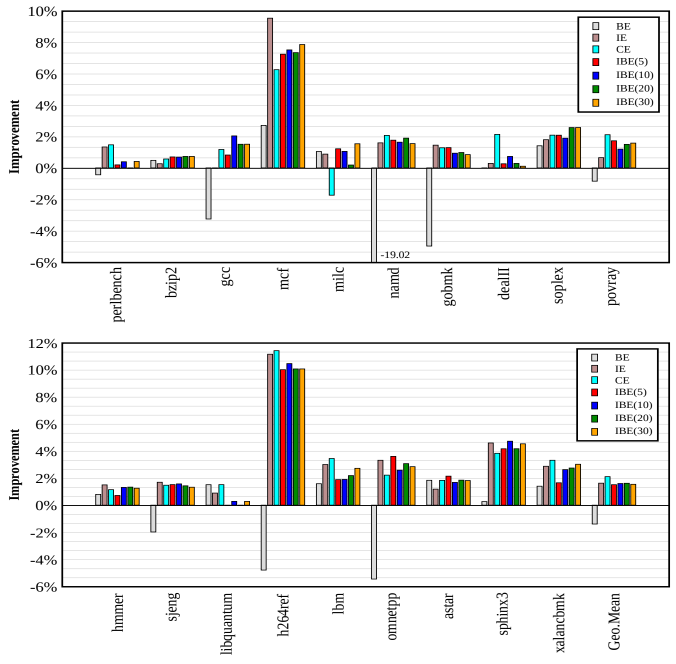

Fig. 2. Program performance achieved by the iterative compilation algorithms relative to the `LLVM` optimization level `O2` for the Spec Cpu2006 benchmarks.

The results indicate that `IBE` is a good strategy to find effective optimization sets. `CE` outperformed `IBE` in 8 Spec Cpu2006 benchmarks (*perlbench, dealII, libquantum, namd, povray, h264ref, lbm* and *xalancbmk*), and `IE` in 4 programs (*gobmk, mcf, hmmer* and *sjeng*). However, it is interesting to note that only in 4 programs (*perlbench, dealII, libquantum* and *gobmk*) the performance gap is not small. It means that in 8 Spec Cpu2006 benchmars the performance of `IBE` is close to the performance of `CE` and/or `IE`.

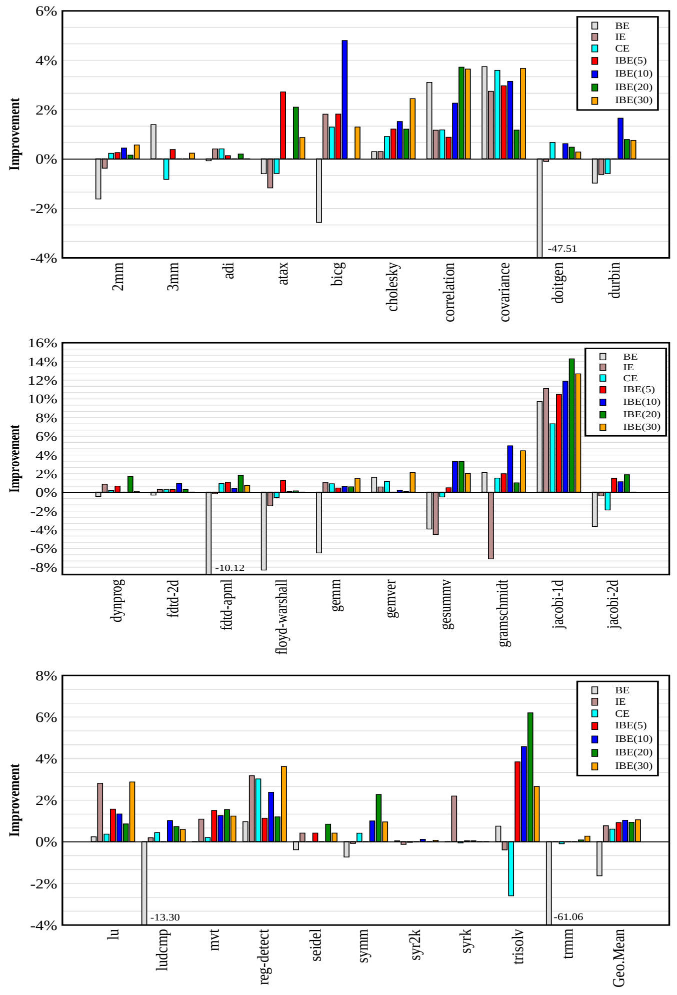

Fig. 3. Program performance achieved by the iterative compilation algorithms relative to the `LLVM` optimization level `O2` for the Polybench benchmarks.

The better performance of Polybench benchmarks is related to the optimization sets founded by `IBE`. In this case, `IBE` achieved the least performance only in 3 cases, namely: *3mm*, *adi* and *doitgen*.

The performance of cBench benchmarks shows a behaviour similar to Spec Cpu2006 benchmarks. `CE` outperforms `IBE` in 6 programs (*bitcount, qsort1, bzip2e, tiff2bw, patricia* and *rsynth*). `IE` outperforms `IBE` in 2 prograns (*susan_e* and *dijkstra*). `BE` outperforms `IBE` also in 2 programs (*susan_s* and *jpeg_d*). However, only in 4 programs (*bitcount, bzip2e, patricia* and *dijkstra*) the performance is not close to the performance of Pan and Eigenmann's algorithms.

As described in [5], `BE` is the worst algorithm. In several cases, `BE` degrades the performance. It never occurs in `IBE`. While `BE` ignores the interaction between optimizations, `IBE` finds the relationship between them, and tries to discover the worst subset to turn off.

`IBE` uses a different strategy from Pan and Eigenmann's algorithms, which is able to achieve comparable or better performance than `BE`, `IE` and `CE`. `IBE` achieves good program

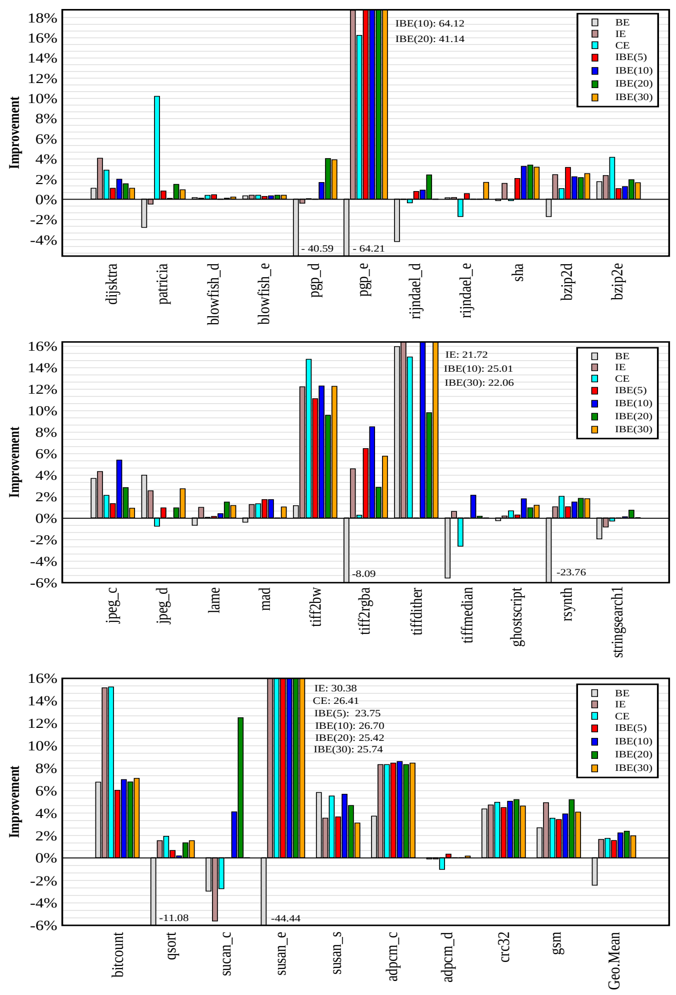

Fig. 4. Program performance achieved by the iterative compilation algorithms relative to the LLVM optimization level O2 for the cBENCH benchmarks.

performance, improving BE. Like BE, IBE tries to remove harmful optimizations in batch. Different from CE and IE, IBE does not need complex iterative steps to discover harmful optimizations.

CE outperforms IBE only in SPEC CPU2006 benchmarks. The tendency of BE is significantly to degrade the performance. The performance of IE never outperforms CE nor IBE.

The results indicate that evaluating different number of points does not change the performance. It means that IBE achieves consistent performance. On the other hand, it does not occur in POLYBENCH and cBENCH benchmarks. The gap in the results is 12.54% and 34.62% for POLYBENCH and cBENCH, respectively.

This gap indicates that for POLYBENCH and cBENCH, IBE is sensitive to the number of iterations. This is an undesirable situation, because it is difficult to discover a specific number of rounds that covers several programs. However, 10 interations are a good choice.

The results show that the program performance is not necessarily related to the length of the optimization set, but it is related to the quality of the optimization set and the characteristics of the program. The Table I shows the length of the final set and its performance improvement. Besides, this table shows two differents scenarios, namely: a marginal improvement, and a significant improvement.

TABLE I. THE LENGTH AND THE IMPROVEMENT OF THE FINAL SET.

Benchmark	Algorithm	bzip2	mcf
SPEC CPU2006	BE	25 (0.51%)	24 (2.73%)
SPEC CPU2006	IE	60 (0.29%)	47 (9.55%)
SPEC CPU2006	CE	27 (0.59%)	35 (6.28%)
SPEC CPU2006	IBE(5)	62 (0.72%)	62 (7.26%)
SPEC CPU2006	IBE(10)	62 (0.71%)	62 (7.53%)
SPEC CPU2006	IBE(20)	62 (0.76%)	62 (7.36%)
SPEC CPU2006	IBE(30)	62 (0.76%)	62 (7.87%)
Benchmark	Algorithm	adi	jacob1-1d
POLYBENCH	BE	59 (-0.07%)	41 (9.72%)
POLYBENCH	IE	53 (0.41%)	38 (11.11%)
POLYBENCH	CE	22 (0.41%)	21 (7.33%)
POLYBENCH	IBE(5)	62 (0.14%)	57 (10.49%)
POLYBENCH	IBE(10)	63 (0%)	59 (11.88%)
POLYBENCH	IBE(20)	56 (0.21%)	58 (14.29%)
POLYBENCH	IBE(30)	63 (0%)	60 (12.68%)
Benchmark	Algorithm	blowfish_e	crc32
cBENCH	BE	52 (0.35%)	55 (4.37%)
cBENCH	IE	45 (0.41%)	42 (4.72%)
cBENCH	CE	54 (0.41%)	2 (4.96%)
cBENCH	IBE(5)	62 (0.29%)	62 (4.49%)
cBENCH	IBE(10)	59 (0.35%)	58 (5.06%)
cBENCH	IBE(20)	59 (0.41%)	51 (5.20%)
cBENCH	IBE(30)	14 (0.41%)	62 (4.63%)

The programs *bzip2*, *adi*, and *blowfish_e* show a situation where it is difficult to achieve performance. Changing the way of analyzing the interactions among the optimizations does not change the program performance. The same occurs changing the length of the optimization set.

A good benchmark performance was not obtained by the largest optimization set. It is the case for *mcf*, *jacobi-1d*, and *crc32*. It is interesting to note that CE founded a good small optimization set for *crc32*.

These results indicate that it is necessary to consider the characteristics of the program, in order to determine the real interaction between the optimizations. Besides, relaxing the way of finding such interaction is a good strategy. In some cases, a good performance can be achieved turning off only few optimizations, a task that does not need several iterative compilations. This is the case of IBE.

The length of the final sets follows the behaviour showed on Table I for all programs. However, there is a difference in the algorithms. The final sets founded by IBE has more optimizations than that founded by BE, IE, and CE.

B. *The Number of Evaluations*

The Figures 5, 6 and 7 show the results of BE, IE, CE and IBE in terms of evaluations for SPEC CPU2006, POLYBENCH and cBENCH, respectively.

The results show that IE is the slowest algorithm, followed by CE. These algorithms need hundreds program evaluations to find a *good* optimization set. The number of evaluations for IE ranges from 128 to 1486, for SPEC CPU2006; from 65 to 1730, for POLYBENCH; and from 65 to 1781, for cBENCH. CE needs less evaluations than IE. The number of evaluations in CE ranges from 66 to 315, for SPEC CPU2006; from 66 to 501, for POLYBENCH; and from 66 to 433, for cBENCH.

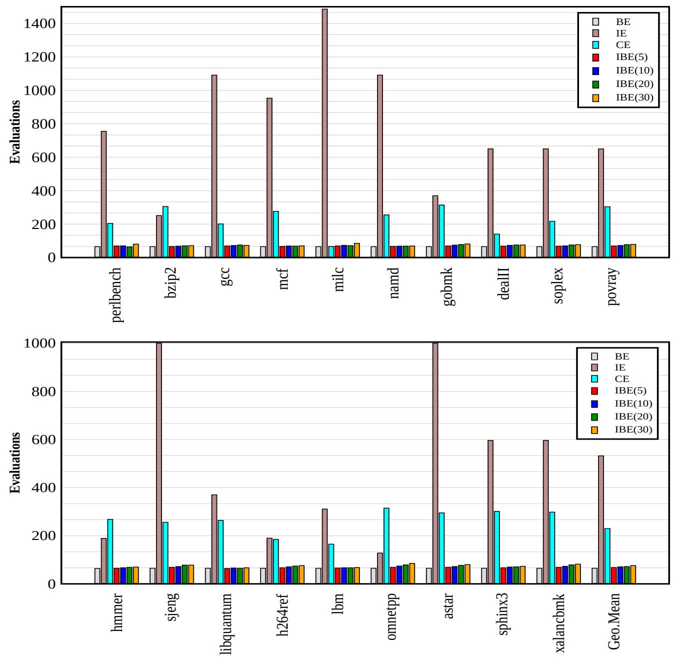

Fig. 5. The number of evaluations of the iterative compilation algorithms for SPEC CPU2006 benchmarks.

BE is the fastest algorithm. However, BE achieves the least performance. This algorithm is $O(n)$, which indicates it needs n evaluations to find an optimizations set. BE needs only 65 evaluations: 1 to evaluate the baseline, 63 to evaluate each optimization, and 1 to evaluate the final set.

BE is the lower bound on the number of evaluations, describing an algorithm that does not explore the interaction between optimizations. IE and CE, which explore the interaction between the optimizations, represents the upper bound on the number of evaluations.

As mentioned previously, the goal of this paper is to propose an algorithm that has performance similar to CE, besides a complexity similar to BE. Thus, on one hand we want an algorithm similar to BE. However, on the other hand, we want an algorithm similar to CE.

Excluding BE, IBE has the smaller number of evaluations, according to its complexity, which is $O(n + c)$. IBE needs only $64 + c$ evaluations: 1 to evaluate the baseline, 63 to evaluate each optimization, and c to evaluate the subsets. In our experiments, we used 4 different values: 5, 10, 20, and 30. This indicates that IBE needs up to $64+30$ evaluations. Table II summaries the results obtained by BE, CE, IE and IBE.

The number of evaluations for IBE(5) ranges from 65 to 69, for SPEC CPU2006; from 65 to 69, for POLYBENCH; and from 65 to 69, for cBENCH. The number of evaluations for IBE(10) ranges from 66 to 74, for SPEC CPU2006; from 65 to 74, for POLYBENCH; and from 65 to 74, for cBENCH. The number of evaluations for IBE(20) ranges from 65 to 79, for SPEC CPU2006; from 65 to 84, for POLYBENCH; and from 65 to 86, for cBENCH. The number of evaluations for IBE(30) ranges from 67 to 85, for SPEC CPU2006; from 65 to 88, for POLYBENCH; and from 65 to 86, for cBENCH.

The number of evaluations indicate that the IBE complexity is similar to BE. Besides, IBE is faster than IE and CE. IBE is 7.4 times faster than IE for SPEC CPU2006, 7.5 times for POLYBENCH, and 7.7 times for cBENCH; 3.2 times faster than CE for SPEC CPU2006, 3.1 times for POLYBENCH, and

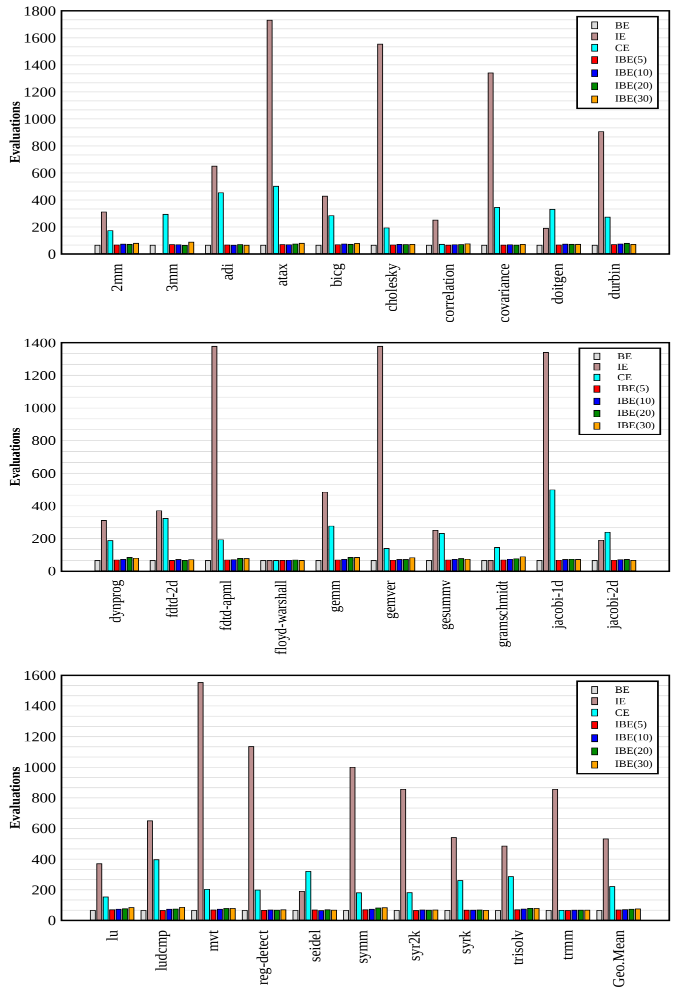

Fig. 6. The number of evaluations of iterative compilation algorithms for POLYBENCH benchmarks.

TABLE II. GEOMETRIC MEAN EVALUATIONS.

Benchmark	Algorithm	Evaluations
SPEC CPU2006	BE	65
SPEC CPU2006	IE	532
SPEC CPU2006	CE	230
SPEC CPU2006	IBE(5)	68
SPEC CPU2006	IBE(10)	71
SPEC CPU2006	IBE(20)	72
SPEC CPU2006	IBE(30)	76
POLYBENCH	BE	65
POLYBENCH	IE	532
POLYBENCH	CE	221
POLYBENCH	IBE(5)	68
POLYBENCH	IBE(10)	70
POLYBENCH	IBE(20)	73
POLYBENCH	IBE(30)	75
cBENCH	BE	65
cBENCH	IE	545
cBENCH	CE	196
cBENCH	IBE(5)	68
cBENCH	IBE(10)	69
cBENCH	IBE(20)	73
cBENCH	IBE(30)	72

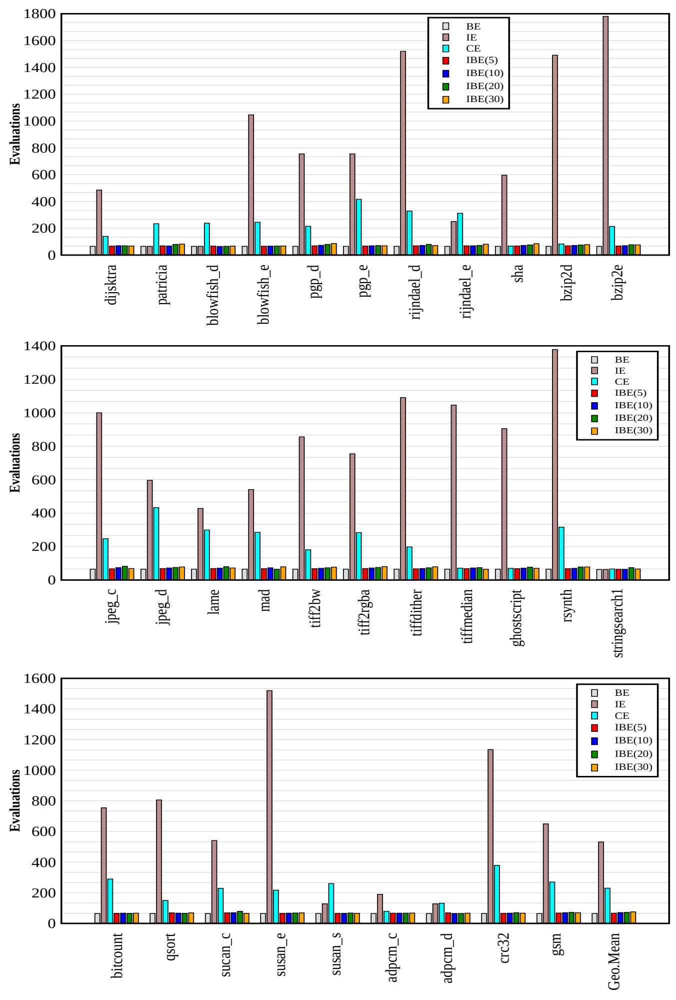

Fig. 7. The number of evaluations of the iterative compilation algorithms for cBench benchmarks.

2.8 times for cBench.

The four algorithms have a different behaviour on each benchmark. It means that the worst case (program) for an algorithm is not necessarily the same for another one. On Spec Cpu2006, the worst case for CE was *omnetpp*, for IE was *milc*, for IBE(5) was *astar*, for IBE(10) was *gobmk*, for IBE(20) was *xalancbmk*, and for IBE(30) was *milc*. The same behaviour occurs in Polybench and cBench, due to the algorithms handle the interactions between the optimizations in different ways.

The number of evaluations is not an indicative of performance. The best performance obtained by the four algorithms for Spec Cpu2006 was on *h264ref*; for Polybench was on *jacobi-1d*; and for cBench was on *susan_e*. In these cases the number of evaluations is not the lower bound nor the upper bound. However, there are exceptions. IE achieves better performance than CE on *mcf*, however it needs more evaluations than CE. On the other hand, IBE achieves better performance than CE, on this benchmark, using less evaluation than CE.

BE, the fastest algorithm, achieves it speed at the cost of performance degradation, a situation that does not occur on IBE. The latter is as fast as BE, but it does not lose performance. It indicates that the strategy of evaluating equidistant points in the interval $[0, 1]$, to discover what optimizations should be removed, is an effective strategy to achieve performance using a reduced number of evaluations.

VI. Concluding Remarks

Modern compilers provide levels of optimizations - usually called, O0, O1, O2 and O3 - to optimize a program using a predetermined compiler optimization set.

Despite of these level to improve the performance of several benchmarks, the quality of the final code varies for each program. It means that although the application of a compiler optimization set can improve the performance of the program, it does not guarantee that the final code is optimal. In addition, not all programs compiled with the same compiler optimization set will obtain a gain in performance.

The main difficulty in discovering the best compiler optimization set for a specific program is related to the complex relationship between optimizations. Besides, due to the number of possible combinations, it is very difficult for the user to select a set that provides good performance for your program.

The literature shows several researches, which goal is to automate the process of choosing a good compiler optimization set. Among these researches, machine learning techniques is nowadays the best approaches. Although, it relies in expensive processes to create the model, mainly to remove harmful compiler optimizations from the model.

Combined Elimination is an effective algorithm to remove harmful optimizations. However, at the cost of several program runs. In this paper, we presented a new algorithm, Improved Batch Elimination, based on a step towards Combined Elimination, Batch Elimination.

We evaluated our algorithm by measuring the performance of Spec Cpu2006, Polybench, and Cbench benchmarks. The results indicate that Improved Batch Elimination is a good strategy to remove harmful optimizations, using few program runs. Besides, Improved Batch Elimination, which complexity is similar to Batch Elimination, provides results similar or better than Combined Elimination. It indicates that Improved Batch Elimination is a good strategy to be used during the process of creating a prediction model.

References

[1] T. Kisuki, P. M. W. Knijnenburg, M. F. P. O'Boyle, F. Bodin, and H. A. G. Wijshoff, "A feasibility study in iterative compilation," in *Proceedings of the Second International Symposium on High Performance Computing*. London, UK, UK: Springer-Verlag, 1999, pp. 121–132.

[2] T. Kisuki, P. Knijnenburg, M. O'Boyle, and H. A. G. Wijshoff, "Iterative compilation in program optimization," in *Proceedings in Compiler for Parallel Computers*, 2000, pp. 35–44.

[3] P. M. W. Knijnenburg, T. Kisuki, and M. F. P. O'Boyle, "Iterative compilation," in *Embedded Processor Design Challenges: Systems, Architectures, Modeling, and Simulation - SAMOS*. London, UK, UK: Springer-Verlag, 2002, pp. 171–187.

[4] E. Park, S. Kulkarni, and J. Cavazos, "An evaluation of different modeling techniques for iterative compilation," in *Proceedings of the 14th International Conference on Compilers, Architectures and Synthesis for Embedded Systems*. New York, NY, USA: ACM, 2011, pp. 65–74.

[5] Z. Pan and R. Eigenmann, "Fast and effective orchestration of compiler optimizations for automatic performance tuning," in *Proceedings of the International Symposium on Code Generation and Optimization*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 319–332.

[6] M. Haneda, P. M. W. Knijnenburg, and H. A. G. Wijshoff, "Automatic Selection of Compiler Options Using Non-parametric Inferential Statistics," in *Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 123–132.

[7] G. G. Fursin, M. F. P. O'Boyle, and P. M. W. Knijnenburg, "Evaluating iterative compilation," in *Proceedings of the 15th International Conference on Languages and Compilers for Parallel Computing*. Berlin, Heidelberg: Springer-Verlag, 2005, pp. 362–376.

[8] S. V. Gheorghita, H. Corporaal, and T. Basten, "Iterative compilation for energy reduction," *Journal of Embedded Computing*, vol. 1, no. 4, pp. 509–520, Dec. 2005.

[9] P. A. Kulkarni, D. B. Whalley, G. S. Tyson, and J. W. Davidson, "Practical exhaustive optimization phase order exploration and evaluation," *ACM Transactions on Architecture and Code Optimization*, vol. 6, no. 1, pp. 1–36, Apr. 2009.

[10] J. H. Foleiss, A. F. da Silva, and L. B. Ruiz, "The Effect of Combining Compiler Optimizations on Code Size," in *Proceedings of the International Conference of the Chilean Computer Science Society*. Curicó, Chile: Sociedad Chilena de Ciencias de la Computación, 2011, pp. 1–8.

[11] ——, "An Experimental Evaluation of Compiler Optimizations on Code Size," in *Proceedings of the Brazilian Symposium on Programming Languages*. São Paulo, São Paulo, Brazil: EACH USP, 2011, pp. 1–15.

[12] M. Haneda, P. M. W. Knijnenburg, and H. A. G. Wijshoff, "Generating new general compiler optimization settings," in *Proceedings of the 19th Annual International Conference on Supercomputing*. New York, NY, USA: ACM, 2005, pp. 161–168.

[13] L. Shun and G. Fursin, "A heuristic search algorithm based on unified transformation framework," in *International Conference Workshops on Parallel Processing*, 2005, pp. 137–144.

[14] K. D. Cooper, A. Grosul, T. J. Harvey, S. Reeves, D. Subramanian, L. Torczon, and T. Waterman, "Exploring the structure of the space of compilation sequences using randomized search algorithms," *Journal of Supercomputing*, vol. 36, no. 2, pp. 135–151, May 2006.

[15] K. D. Cooper, D. Subramanian, and L. Torczon, "Adaptive optimizing compilers for the 21st century," *Journal of Supercomputing*, vol. 23, no. 1, pp. 7–22, Aug. 2002.

[16] W. Zhao, B. Cai, D. Whalley, M. W. Bailey, R. van Engelen, X. Yuan, J. D. Hiser, J. W. Davidson, K. Gallivan, and D. L. Jones, "Vista: A system for interactive code improvement," in *Proceedings of the Joint Conference on Languages, Compilers and Tools for Embedded Systems: Software and Compilers For Embedded Systems*. New York, NY, USA: ACM, 2002, pp. 155–164.

[17] P. A. Kulkarni, S. R. Hines, D. B. Whalley, J. D. Hiser, J. W. Davidson, and D. L. Jones, "Fast and efficient searches for effective optimization-phase sequences," *ACM Transactions on Architecture and Code Optimization*, vol. 2, no. 2, pp. 165–198, Jun. 2005.

[18] Y. Che and Z. Wang, "A lightweight iterative compilation approach for optimization parameter selection," in *First International Multi-Symposiums on Computer and Computational Sciences*, vol. 1. Washington, DC, USA: IEEE Computer Society, 2006, pp. 318–325.

[19] Y.-Q. Zhou and N.-W. Lin, "A study on optimizing execution time and code size in iterative compilation," in *Third International Conference on Innovations in Bio-Inspired Computing and Applications*, 2012, pp. 104–109.

[20] S. Purini and L. Jain, "Finding good optimization sequences covering program space," *ACM Transactions on Architecture and Code Optimization*, vol. 9, no. 4, pp. 56:1–56:23, Jan. 2013.

[21] F. Agakov, E. Bonilla, J. Cavazos, B. Franke, G. Fursin, M. F. P. O'Boyle, J. Thomson, M. Toussaint, and C. K. I. Williams, "Using machine learning to focus iterative optimization," in *Proceedings of the International Symposium on Code Generation and Optimization*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 295–305.

[22] J. Cavazos, G. Fursin, F. Agakov, E. Bonilla, M. F. P. O'Boyle, and O. Temam, "Rapidly selecting good compiler optimizations using performance counters," in *Proceedings of the International Symposium on Code Generation and Optimization*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 185–197.

[23] J. Cavazos and M. F. P. O'Boyle, "Method-specific dynamic compilation using logistic regression," *ACM SIGPLAN Notices*, vol. 41, no. 10, pp. 229–240, Oct. 2006.

[24] M. R. Jantz and P. A. Kulkarni, "Performance potential of optimization phase selection during dynamic jit compilation," in *Proceedings of the 9th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. New York, NY, USA: ACM, 2013, pp. 131–142.

[25] S. Long and M. O'Boyle, "Adaptive java optimisation using instance-based learning," in *Proceedings of the 18th Annual International Conference on Supercomputing*. New York, NY, USA: ACM, 2004, pp. 237–246.

[26] A. M. Malik, "Spatial based feature generation for machine learning based optimization compilation," in *Proceedings of the Ninth International Conference on Machine Learning and Applications*. Washington, DC, USA: IEEE Computer Society, 2010, pp. 925–930.

[27] J. Thomson, M. O'Boyle, G. Fursin, and B. Franke, "Reducing training time in a one-shot machine learning-based compiler," in *Proceedings of the 22nd International Conference on Languages and Compilers for Parallel Computing*. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 399–407.

[28] J. L. Henning, "Spec cpu2006 benchmark descriptions," *SIGARCH Computer Architecture News*, vol. 34, no. 4, pp. 1–17, Sep. 2006.

[29] S. Team, "Standard performance evaluation corporation," http://www.spec.org, 2015, online; accessed 05-August-2005.

[30] L.-N. Pouchet, "Polyhedral benchmark suite," http://www.cs.ucla.edu/ pouchet/software/polybench/, 2015, online; accessed 05-August-2005.

[31] cBench., "The collective benchmarks," http://ctuning.org/wiki/index.php/CTools:CBench, 2015, online; accessed 05-August-2005.

[32] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *Proceedings of the International Symposium on Code Generation and Optimization*. Washington, DC, USA: IEEE Computer Society, 2004, pp. 75–.

[33] L. Team, "The llvm compiler infrastructure," http://llvm.org, 2015, online; accessed 05-August-2005.