A Design Space Exploration of Compiler Optimizations Guided by Hot Functions

Marcos Yukio Siraichi
Departament of Informatic
State Univerity of Maringá
Maringá, Brazil
Email: sir.yukio@gmail.com

Caio Tonetti
Departament of Informatic
State Univerity of Maringá
Maringá, Brazil
Email: caio.tonetti@gmail.com

Anderson Faustino da Silva
Departament of Informatic
State Univerity of Maringá
Maringá, Brazil
Email: anderson@gmail.com

***Abstract*—It is well-known that a large amount of program runtime is spent by hot functions. It highly indicates that such functions should guide the process of exploring which compiler optimization sequence will be used during the translation of source code into target code. Although the literature presents several Design Space Exploration (DSE) techniques, these are not guided by hot functions. To fill this gap, we present a DSE of compiler optimizations which is guided by hot functions, which employs a case-based reasoning technique to find a good compiler optimization sequence for unseen programs. We performed a number of experiments targetting the Intel processor Core I7-3770 using the Clang/LLVM 3.7.0 compiler, considering 131 LLVM optimizations and the benchmarks CBENCH and POLYBENCH. The results show that our DSE is able to achieve a geometric mean speedup of 2.013 over the O0 flag; versus geometric mean speedups from 1.632 to 2.036 obtained with other approaches.**

***Keywords*—Design Space Exploration, Compiler, Optimization, Hot Function.**

I. INTRODUCTION

Modern compilers, a program that transforms source code from one language (source language) unto another (target language) [1], employ several code transformations [2] - known as optimizations - to improve the target code quality. However, a specific compiler optimization sequence can be useful to a program, but not to another. Then, the most appropriate approach is to find a good sequence, considering the features of the program.

A well-known DSE is iterative compilation [3]. In this technique, the program is compiled with different sequences, and the best target code is chosen. Due to the diversity of sequences, iterative compilation techniques try to cover the search space selectively [3] [4]. In order to reduce the number of sequences evaluated (to compile the program using a specific compiler optimization sequence, and after running the target code and measuring its runtime), several researches proposed the use of machine learning. This technique creates in a training stage a prediction model, which will predict, in a deployment stage, the optimization sequence that will be enabled during the compilation of the unseen (test) program [5] [6] [7].

Although, it is known that a large amount of program runtime is spent in hot functions, neither iterative compilation nor machine learning techniques, presented in the literature, take into account such functions to guide the process of exploring compiler optimizations. In fact, these hot functions are the portions at which compiler optimizations will provide the greatest benefit.

In this paper we propose a DSE of compiler optimizations, which is guided by hot functions. Our DSE is classified as a machine learning technique. In fact, it is a case-based reasoning (CBR) technique [8], an approach considered a subfield of machine learning [9], [10] that tries to solve a new problem using a solution of an previous similar situation. It can be seen as a learning process [11] that stores past experiences in a database, which is updated incorporating new experiences [12]. Thus, based on past experiences (previously-generated sequences) our DSE infers good sequences for unseen programs.

We implemented the DSE as a tool of the LLVM infrastructure, and the results indicate that the geometric mean speedup over the compiler optimization Level O0 is 2.013; while the geometric mean speedup obtained by O3 is 1.849. In addition, experiments with iterative compilation techniques indicate that our proposal surpasses these techniques, also in terms of speedup.

II. CASE-BASED REASONING

CBR, a machine learning approach, can be subdivided in four processes:

1) Retrieve a case from a collection of past experiences (previous cases) by similarity measure.
2) Reuse the knowledge of a past experience to solve a new case.
3) Revise the result of this new case, evaluating the success of the solution.
4) Retain the useful experience for future reuses.

The retrieval process in every CBR requires some parameters, such as:

- **Collection guide** indicates the strategy used to build the collection of past experiences.
- **Similarity measure** measures the level of similarity between a previous case and a new one.
- **Standardization** transforms all attributes values according to a specific rule.

978-1-5090-3339-3/16/$31.00

- **Number of analogies** indicates the number of past experiences that will be used to estimate a solution to an unseen problem.

Our DSE performs these four basic process, in a deployment stage. On the other hand, in a training stage we build a collection of past experiences, which will be perused in order to find an optimization sequence for a specific unseen program. Perusing the collection is based on similar patterns among programs, which are represented by a standardized feature vector.

III. The Design Space Exploration

The aim of our DSE is to find a compiler optimization sequence, which is able to outperform the well-engineered compiler optimization levels. The DSE infers from previously-generated sequences, the best sequence that fits the features of the test program. This process is based on two premises, namely:

1) We can find similar patterns among programs, which give important insights for exploring potential sequences; and
2) Similar programs react approximately the same way, when they are compiled using the same sequence.

To explore these premises, each program is represented by a feature vector in a multidimensional space and a similarity model operates in this space trying to find similar points, which indicates similar patterns that should be exploit.

Our DSE is outlined as:

1) Retrieve a case
   a) Extract the hot function from the unseen program
   b) Represent the hot function using a symbolic representation
   c) Peruse the collection of past experiences (database) searching the most similar symbolic representation
2) Reuse the case
   a) Compile the unseen program using the retrieved case
3) Revise the result
   a) Evaluate the retrieved case
4) Retain useful experience
   a) Update the database with this experience
5) Return the best target code

A. Extracting The Hot Function

Wu and Larus [13] proposed a static profiler to estimate the relative execution frequency of pieces of the program, such as: calls, procedure invocations, basic blocks, and control-flow edges. It is an appealing tool, due to not requiring neither program instrumentation nor execution. Given the control-flow graph G for the function F, Wu-Larus Algorithm performs two steps:

1) Calculate for each edge its probability; and
2) Transverse the control-flow graph propagating the probabilities.

In the first step, Wu-Larus starts by combining the branch prediction heuristics proposed by Ball and Larus [14], which are simple assumptions that take into account specific compiler implementations and architecture designs. Namely, there are seven of them:

1) **OpCode Heuristic:** if the branch condition is either a comparison of "less than zero" or "less than or equal zero," this branch will not be taken;
2) **Loop Heuristic:** if the successor is either a loop head or a loop pre-header, this branch will be taken;
3) **Pointer Heuristic:** if the branch condition is a comparison between two pointers, this branch will not be taken;
4) **Call Heuristic:** if the successor either contains a call or unconditionally reaches a block with a call, the other branch will be taken;
5) **Return Heuristic:** if the successor either contains a return or unconditionally reaches a block with a return, the other branch will be taken;
6) **Guard Heuristic:** if a register is an operand of this branch and it is used in the successor before it is defined, this branch will be taken;
7) **Store Heuristic:** if the successor has a store operation, the other branch will be taken.

Note that this is only a summarization of the heuristics. Also, these heuristics are only applied to what they call "non-loop branch", which consist in branches whose outgoing edges are neither exit edges nor backedges.

As these heuristics are binary predictions, Wu and Larus relied on the experiments executed by Ball and Larus and employed the frequency, at which the predictions were correct, as the branch probabilities. Table I presents these probabilities.

TABLE I
Branch probability of each heuristic[13]

Heuristic	Branch Probability
OpCode Heuristic	84%
Loop Heuristic	88%
Pointer Heuristic	60%
Call Heuristic	78%
Return Heuristic	72%
Guard Heuristic	62%
Store Heuristic	55%

With the branch probabilities calculated, the second step takes place. In this step, these probabilities are propagated throughout the basic blocks, yielding the edge and basic block's frequencies. For this step, let $\texttt{freq}(b_i)$ be the frequency of the basic block i and let $\texttt{freq}(b_i \rightarrow b_j)$ be the frequency of the edge from basic block i to basic block j, we have the following:

$$freq(b_i) = \begin{cases} 1 & if\ b_1\ is\ the\ entry\ block \\ \sum_{b_p \in pred(b_i)} freq(b_p \rightarrow b_i) & otherwise \end{cases} \quad (1)$$

$$freq(b_p \rightarrow b_i) = freq(b_p) \times prob(b_p \rightarrow b_i) \quad (2)$$

As the Equation 1 shows, the frequency of the basic block `i` is calculated by the sum of all the edge frequencies from its predecessor, with exception of the entry basic block which frequency is one. The Equation 2 uses the probability calculated on step one, and calculates the edge probability.

However, for functions that have loops these equations become mutually recursive, turning the algorithm too slow and unable to handle loops with no apparent boundaries. Thus, Wu and Larus presented an elimination algorithm as follows:

$$cp(b_0) = \sum_{i=1}^{k} r_i \times prob(b_i \rightarrow b_0) \tag{3}$$

$$\begin{aligned} freq(b_0) &= in\_freq(b_0) + \sum_{i=1}^{k} freq(b_i \rightarrow b_0) \\ &= \frac{in\_freq(b_0)}{1 - cp(b_0)} \end{aligned} \tag{4}$$

Where b_0 is the loop header, `cp(`b_0`)` is the cyclic probability and `in_freq(`b_0`)` is the incoming edge frequency. In the Equation 3, r_i represents the probability of the control flow from b_0 to b_i. Therefore, its multiplication with the probability of the branch represents the probability of taking the backedge from basic block b_i. The Equation 4 makes use of the cyclic probability to calculate the total basic block frequency of the loop header. The Algorithm 1 illustrates the algorithm described above.

By going through these two steps, it is possible to calculate an estimation of a function's total cost. This process is illustrated by Equation 5, where for each function `f`, we calculate its cost by summing the product of the basic block frequency (`freq(bb)`) by the cost of each instruction (`cost(i)`), for each basic block inside the function. Hence, we apply these steps to all functions, and identify the highest scoring function.

$$cost(f) = \sum_{bb \in f} \sum_{i \in bb} cost(i) \times freq(bb) \tag{5}$$

We are interested in the highest (hottest) scoring function because only this will guide the exploration of compiler optimizations. This means that the whole program will be compiled using a sequence that fits the features of its hottest function.

B. Representing The Hot Function

Machine learning techniques rely on exposing the similarities among programs to identify patterns, and decide what sequence should be enabled during target code generation.

Previous works represented the program using performance counters [6], control-flow graphs [15], and a symbolic representation [16]. In this work, we also use a symbolic representation, similar to a `DNA`, which encodes program elements into a single string.

As our technique works in the `LLVM`'s Intermediate Language, the transformation rules encode each `LLVM`'s instruction. Such rules are outlined in the Table II.

Algorithm 1 Step two algorithm[13]. Propagates the branch probabilities to produce the edge and basic block frequencies. `Visited` is the set of visited basic blocks, `B` is the set of backedges and `beProb` is the probability of the backedge.

procedure PROPAGATEFREQ($b, head$)
  if $b \in Visited$ **then**
    return
  if $b == head$ **then**
    $freq(b) \leftarrow 1$
  else
    for all $b_p \in pred(b)$ **do**
      if $b_p \notin Visited$ and $(b_p \rightarrow b) \notin B$ **then**
        return
    $freq(b) \leftarrow 0$
    $cycProb \leftarrow 0$
    for all $b_p \in pred(b)$ **do**
      if $(b_p \rightarrow b) \in B$ **then**
        $cycProb \leftarrow cycProb + beProb(b_p \rightarrow b)$
      else
        $freq(b) \leftarrow freq(b) + freq(b_p \rightarrow b)$
    if $cycProb > 1 - \epsilon$ **then**
      $cycProb = 1 - \epsilon$
    $freq(b) \leftarrow \frac{freq(b)}{1 - cycProb}$
  $Visited \leftarrow b \bigcup Visited$
  for all $b_s \in succ(b)$ **do**
    $freq(b \rightarrow b_s) \leftarrow prob(b \rightarrow b_s) \times freq(b)$
    if $b_s == head$ **then**
      $beProb(b \rightarrow b_s) \leftarrow prob(b \rightarrow b_s) \times freq(b)$
  for all $b_s \in succ(b)$ **do**
    if $(b \rightarrow b_s) \notin B$ **then**
      $propagateFreq(b_s, head)$

TABLE II
DNA ENCODING

Transformation Rules			
Br	A	Store	K
Switch	B	Alloca	L
IndirectBr	C	Fence, AtomicRMW, AtomicCmpXchg	M
Ret, Invoke, Resume, Unreachable	D	GetElementPTR	N
Add, Sub, Mul, UDiv, SDiv, URem, SRem	E	Trunc, ZExt, SExt, UIToFP, SIToFP, PtrToInt, IntToPtr, BitCast, AddrSpaceCast	O
FAdd, FSub, FMul, FDiv, FRem	F	FPTrunc, FPExt, FPToUI, FPToSI	P
Shl, LShr, AShr, And, Or, Xor	G	ICmp, FCmp, Select, VAArg, LandingPad	Q
ExtractElement, InsertElement, SuffleVector	H	PHI	R
ExtractValue, InsertValue	I	Call	S
Load	J	Others	X

The transformation rules group instructions into different genes. As a result, our technique can identify which instruction group dominates the hot function, and use these insights for exploring potential heuristics.

C. Scoring Past Experiences

Finding a good compiler optimization sequence for a unseen program is based on similarity among programs. Our premise is that similar programs react approximately the same way when they are compiled using the same sequence. In this manner, we need a method to find the most similar similar reactions.

We determine a similar reaction, between two programs, aligning their DNA representation. For this purpose we use Needleman-Wunsch Algorithm [17].

Needleman and Wunsch proposed an optimal global alignment algorithm to find similarities between two biological sequences. The iterative algorithm considers all possible pair combinations that can be constructed from two amino-acid sequences. Given two amino-acid sequences, A and B, Needleman-Wunsch Algorithm performs two steps:

1) Create the similarity matrix, MAT; and
2) Find the maximum match.

The maximum match can be determined by a two-dimensional array, where two amino-acid sequences, A and B, are compared. Each amino-acid sequence is numbered from 1 to N, where A_j is the jth element of the sequence A and B_i is the ith element of sequence B, with A_i representing the columns and B_i the rows of the two-dimensional matrix. Then, considering the matrix MAT, MAT_{ij} represents the pair combination of A_j and B_i.

To ensure that the sequence don't have permutations of elements, a pair combination MAT_{ij} is a part of a pathway containing MAT_{mn} if and only if their indexes are $m > i$, $n > j$ or $m < i$, $n < j$. Thus, any pathway can be represented by a number of pair permutations MAT_{ab} to MAT_{yz}, where $a >= 1$, $b >= 1$, and the subsequent indexes of the cells of MAT are larger than the indexes of the previous cells and smaller than the number of elements in the respective sequences A and B. A pathway begins at a cell in the first column or first row of MAT, where the index of i and j needs to be incremented by one and the other by one or more, leading to the next cell in the pathway. Repeating this process until their limiting values, we create a pathway where every partial or unnecessary pathway will be contained in at least one necessary pathway.

As a result of this process, the maximum match returns a score which indicates the similarity between the amino-acid sequences A and B. The Algorithm 2 illustrates this process.

Therefore, using Needleman-Wunsch Algorithm, our DSE scores (and ranks) past experiences, aligning the DNA of the unseen program (its hot function) with each DNAs from the database.

Algorithm 2 Needleman-Wunsch Algorithm [17]. Iteratively calculate the maximum match of each pair of cells in the matrix. A is the first sequence, B is the second sequence, MAT is the matrix, GP is the Gap penalty, S is a function that return the score based on similarity, AlignmentA and AlignmentB are the respective alignment of the sequences A and B.

```
1:  procedure NEEDLEMANWUNSCH(A, B)
2:      for i ∈ 1 → lenght(A) do
3:          for j ∈ 1 → lenght(B) do
4:              if A(j) == B(i) then
5:                  MAT(i, j) = 1
6:
7:      for i ∈ (lenght(A) − 1) → 1 do
8:          for j ∈ (lenght(B) − 1) → 1 do
9:              match = MAT(i + 1, j + 1) + S(A_j, B_i)
10:             delete = MAT(i + 1, j) + GP
11:             insert = MAT(i, j + 1) + GP
12:             MAT(i, j) = max(match, delete, insert)
13:
14:     i = 1
15:     j = 1
16:     while i < lenght(A) and j < lenght(B) do
17:         Score = MAT(i, j)
18:         Diag = MAT(i + 1, j + 1)
19:         Up = MAT(i, j + 1)
20:         Left = MAT(i + 1, j)
21:         if Score = Diag + S(A_j, B_i) then
22:             AlignmentA = A_i + AlignmentA
23:             AlignmentB = B_j + AlignmentB
24:             i = i + 1
25:             j = j + 1
26:         else if Score = Left + GP then
27:             AlignmentA = A_i + AlignmentA
28:             AlignmentB = " − " + AlignmentB
29:             i = i + 1
30:         else if Score = Up + GP then
31:             AlignmentA = " − " + AlignmentA
32:             AlignmentB = B_j + AlignmentB
33:             j = j + 1
34:     while i < lenght(A) do
35:         AlignmentA = A_i + AlignmentA
36:         AlignmentB = " − " + AlignmentB
37:         i = i − 1
38:     while j < lenght(B) do
39:         AlignmentA = " − " + AlignmentA
40:         AlignmentB = B_i + AlignmentB
41:         j = j − 1
```

D. Evaluating a Retrieved Case

As stated before, our technique explores optimization sequences taken from programs, which react approximately the same way when they are compiled using the same sequence.

Based on our premises, we could conclude that the good strategy is to evaluate the best previously-generated sequence used by the most *similar* program. This is true if and only if we ensure that the best sequence is safe. In fact, we can not ensure that. We need to note that some flags (optimizations) are unsafe, meaning that such flags can be enabled in a specific program, but not in another. As a result, our DSE evaluates

N previously-generated sequences compilation heuristics. It ensures that our DSE will always find a safe sequence.

E. Returning the Best Target Code

After evaluating N sequences and finding the best one, which fits the features of the unseen program, the DSE updates the database with this new information. The goal is to learn from these new compilations. Finally, the DSE returns the best target code.

F. An Example

Now, take as example a simple program that computes $a = b * c + d$, and is composed by the single source code:

Listing 1. $a = b * c + d$ in LLVM intermediate representation.

```
define i32 @mul_add(i32 %x, i32 %y, i32 %z) {
entry:
  %tmp = mul i32 %x, %y
  %tmp2 = add i32 %tmp, %z
  ret i32 %tmp2
}

define i32 @main(i32 %x, i32 %y, i32 %z) {
  %tmp3 = call i32 @mul_add(i32 %x, i32 %y, i32 %z)
  ret i32 %tmp3
}
```

First, our program will uses Wu and Larus static profiler to extract the hot function from the source code, calculating the probability for each edge and transversing the control-flow graph propagating the probabilities. In this example, both the functions mul_add and main will have cost 1 (both are composed of only one basic block). Let the hot function be mul_add. After finding the hot function, ourt DSE encodes mul_add using the rules outlined in the Table II. Such encoding is EED, respectively to the instructions mul, add and ret.

Using this encoded DNA, the Needleman and Wunsch similarity algorithm takes place, comparing the similarity of the mul_add function with all others, in the database. As a result, N similar DNAs with their optimization sequences are retrieved.

So, if the database has 3 entry's, being $EEDA$ with -inline-cost, -dce, DRF with -loop-deletion, -no-aa, -sancov and $EEGD$ with -memdep, -die and -loop-reduce, the similarities with EED would be found between $EEDA$ and $EEGD$, retrieving them and their respective sequences. Then, the program will be compiled and executed, using each sequence. Finally, the DSE returns a feedback to the database and sends the best target code to the user.

This feedback will be used for future compilations. The idea is for every compiled program, the database will add one entry with the hot function encoding and the best optimization sequence founded. Therefore, next time, when searching for the most similar program, there may be a program added afterwards that is more similar than the previous ones.

IV. A Database of Previously-generated Sequences

As our DSE relies on previously-generated sequences, it is necessary to construct a prior a database.

The database stores a DNA representation, which is collected compiling the program without using optimizations, and one compiler optimization sequence for different training programs.

This database can be constructed in a process *from factory*. Thus, at the factory, an engine collects pieces of information about a set of training programs and reduces the optimization search space in order to provide a small database, which can be handled in a easy and fast way.

The database can be viewed as a table composed by several entries, where each entry is composed by two fields, namely:

1) A DNA representation of the hot function; and
2) The best compiler optimization sequence founded by an iterative compilation technique.

A. Training Programs

The training programs are composed by programs took from LLVM's test-suite [18], and The Computer Language Benchmarks Game [19]. These are programs composed of a single source code, and have short runtime. Table III shows the training programs.

TABLE III
The Training Programs

LLVM's test-suite			
ackermann	flops-4	mandel-2	queens
ary3	flops-5	mandel	queens-mcgill
bubblesort	flops-6	matrix	quicksort
chomp	flops-7	methcall	random
dry	flops-8	misr	realmm
dt	flops	n-body	recursive
fannkuch	fp-conver	nsieve-bits	reedsolomon
fbench	hash	oourafft	richards_benchmark
ffbench	heapsort	oscar	salsa20
fib2	himenobtxpa	partialsums	sieve
fldry	huffbench	perlin	spectral-norm
flops-1	intmm	perm	strcat
flops-2	lists	pi	towers
flops-3	lpbench	puzzle	treesort
		puzzle-stanford	whetstone
The Computer Language Benchmarks Game			
binary-tree	fasta-redux	pidigits	regex-dna
fasta	mandelbrot		

B. Optimizations

The optimizations, which can compose an optimization sequence, are presented in Table IV.

C. Reducing the Search Space

We use a Genetic Algorithm (GA) to reduce the search space and find a good compiler optimization sequence for each training program.

The GA consists in randomly generating an initial population, which will be evolved in an iterative process. Such process involves choosing parents; applying genetic operators;

TABLE IV
LLVM'S OPTIMIZATIONS

Optimizations				
-aa-eval	-adce	-add-discriminators	-alignment-from-assumptions	-alloca-hoisting
-always-inline	-argpromotion	-assumption-cache-tracker	-atomic-expand	-barrier
-basicaa	-basiccg	-bb-vectorize	-bdce	-block-freq
-bounds-checking	-branch-prob	-break-crit-edges	-cfl-aa	-codegenprepare
-consthoist	-constmerge	-constprop	-correlated-propagation	-cost-model
-count-aa	-da	-dce	-deadargelim	-deadarghaX0r
-delinearize	-die	-divergence	-domfrontier	-domtree
-dse	-dwarfehprepare	-early-cse	-elim-avail-extern	-flattencfg
-float2int	-functionattrs	-globaldce	-globalopt	-globalsmodref-aa
-gvn	-indvars	-inline	-inline-cost	-instcombine
-instcount	-instnamer	-instsimplify	-intervals	-ipconstprop
-ipsccp	-irce	-iv-users	-jump-threading	-lazy-value-info
-lcssa	-libcall-aa	-licm	-lint	-load-combine
-loop-accesses	-loop-deletion	-loop-distribute	-loop-extract	-loop-extract-single
-loop-idiom	-loop-instsimplify	-loop-interchange	-loop-reduce	-loop-reroll
-loop-rotate	-loop-simplify	-loop-unroll	-loop-unswitch	-loop-vectorize
-loops	-lower-expect	-loweratomic	-lowerbitsets	-lowerinvoke
-lowerswitch	-mem2reg	-memcpyopt	-memdep	-mergefunc
-mergereturn	-mldst-motion	-nary-reassociate	-no-aa	-partial-inliner
-partially-inline-libcalls	-place-backedge-safepoints-impl	-place-safepoints	-postdomtree	-prune-eh
-reassociate	-reg2mem	-regions	-rewrite-statepoints-for-gc	-rewrite-symbols
-safe-stack	-sancov	-scalar-evolution	-scalarizer	-scalarrepl
-scalarrepl-ssa	-sccp	-scev-aa	-scoped-noalias	-separate-const-offset-from-gep
-simplifycfg	-sink	-sjljehprepare	-slp-vectorizer	-slsr
-speculative-execution	-sroa	-strip	-strip-dead-prototypes	-strip-nondebug
-structurizecfg	-tailcallelim	-targetlibinfo	-tbaa	-tti
-verify				

evaluating new individuals; and finally a reinsertion operation deciding which individuals will compose the new generation. This iterative process is performed until a stopping criterion is reached.

The first generation is composed of individuals that are generated by a uniform sampling of the optimization space. Evolving a population includes the application of two genetic operators: crossover, and mutation. The first operator has a probability of 60% for creating a new individual. In this case, a tournament strategy ($Tour = 5$) selects the parents. The second operator, mutation, has a probability of 40% for transforming an individual. In addition, each individual has an arbitrary initial length, which can ranges from 1 to $|Optimization\ Space|$. Thus, the crossover operator can be applied to individuals of different lengths. In this case, the length of the new individual is the average of its parents. Four types of mutation operations were used:

1) Insert a new optimization into a random point;
2) Remove an optimization from a random point;
3) Exchange two optimizations from random points; and
4) Change one optimization in a random point.

Both operators have the same probability of occurrence, besides only one mutation is applied over the individual selected to be transformed. This iterative process uses elitism, which maintains the best individual in the next generation. Furthermore, it runs over 100 generations and 50 individuals, and finishes whether the standard deviation of the current fitness score is less than 0.01, or the best fitness score does not change in three consecutive generations.

The strategy used to reduce the search space is similar to the strategy proposed by Martins *et al.* [20] and Purini and Jain [4].

V. EXPERIMENTAL SETUP AND METHODOLOGY

This section describes the experimental setup and the steps taken to ensure measurement accuracy, besides the methodology used in the experiments.

A. Platform

The experiments were conducted on a machine with an Intel processor Core I7-3770 3.40GHz, and 8 GB of RAM. The operating system was Ubuntu 15.10, with kernel 4.2.0-16-generic.

B. Compiler

Our technique was implemented as a tool of `LLVM` 3.7.0 [21], [18]. The choice of `LLVM` is based on the fact that it allows full control over the optimizations. This means that it is possible to enable a list of optimizations through the command line. In addition, the position of each optimization indicates its order. Neither `GCC` nor `ICC` provide these features, thus we chose to use `LLVM`.

The Figure 1 depicts the structure of `LLVM`.

Basically, `LLVM` is composed of *tools* and *libraries*. Our system consists of tree components: (1) the tool *DSE*; (2) the library *libLLVMWuLarus*; and (3) the library *libLLVMNeedlemanWunch*.

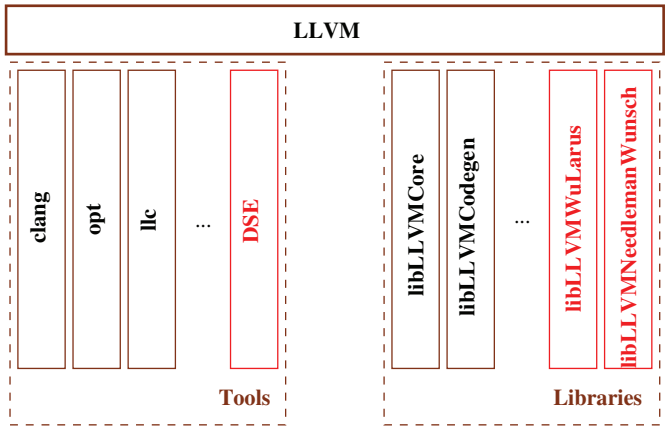

Fig. 1. The structure of LLVM

C. Benchmarks

The experiments use two benchmarks as unseen (test) programs, namely: POLYBENCH 4.1 [22] with extralarge dataset, and CBENCH [23] with dataset 1.

D. Measurement

The results are based on the arithmetic average of five executions. In the experiments, the machine workload was as minimal as possible. In other words, each instance was executed sequentially. In addition, the machine did not have an external interference, and the running time variance was close to zero.

E. Parameters and Order of Compilation

We explore 10 optimization sequences. In fact, we performed experiments exploring 1, 3, 5, and 10 sequences and the last configuration ensured that our DSE always finds a safe sequence.

As our technique updates the database, the order of compilation affects the results. Of course, we are not able to predict in which order the user will compile his/her programs. Thus, we decided to compile in alphabetic order.

F. Baseline

We evaluate all compiler optimizations levels, in order to find the best one. The geometric mean speedup obtained by O1, O2 and O3, over the level O0 is 1.693, 1.843, 1.849, respectively. Therefore, we chose the level O3 as the baseline.

G. Metrics

The evaluation uses four metrics to analyze the results, namely:

1) GMS: geometric mean speedup;
2) NPS: number of programs achieving speedup over the optimizations level O3;
3) NoS: number of sequences evaluated; and
4) ReT: the technique's response time.

The speedup is calculated as follows:

$$Speedup = Running\_time\_Level\_O0 / Running\_time$$

H. Other Techniques

To evaluate the effectiveness of our technique, we compare it with four techniques:

1) *Random Algorithm (*Random10*)* This iterative algorithm randomly generates 10 sequences.
2) *Genetic Algorithm with Tournament Selector (*GA50*)* It is similar to the technique described on IV-C.
3) *Genetic Algorithm with Tournament Selector (*GA10*)* It is also similar to the technique described on IV-C, except that it runs over 10 generations and 20 individuals.
4) Best10 It is a technique proposed by Purini and Jain [4]. They founded 10 good sequences, which is able to cover several classes of programs. Thus, in this technique the unseen programs is compiled with all sequences, and the best target code is returned.

I. Costs

The training, which builds the database, is a time-consuming phase. It took precisely 20 days. However, it is important to note that it was performed only one-time at the factory.

The search for a sequence is a fast task, which includes extracting the DNA of the test program, scoring the training programs, and selecting N sequences. It takes less than 0.1% of the entire response time. It means that at least 99.9% of the time is spent evaluating target codes. Therefore, we can conclude that our technique is very fast.

VI. EXPERIMENTAL RESULTS

This section evaluates our technique. We hope that it will outperform the well-engineered optimization level O3, in a few evaluations. First of all, we evaluate the speedup. Second, we evaluate the number of sequences evaluated. After, we evaluate the response time.

A. Speedup

Figure 2 shows the speedups obtained by our technique, Random10, GA50, GA10, and Best10.

The GMS obtained by the compiler optimization level O3 is 1.849. Using our technique the GMS is 2.013, while using Random10, GA50, GA10 and Best10, this metric is 1.632, 2.036, 1.799 and 1.980, respectively. These results indicate that our technique does not outperform only GA50; however, the performance lost is very small, 1.143%.

It is important to remember that our technique and GA have different premises. The former is a machine learning technique, whose goal is to find a good sequence in a few steps. The latter is an iterative compilation technique, which needs in general more steps to achieve performance. Therefore, we show that it is possible to achieve a similar performance (our DSE versus GA50), using a simple strategy. In fact, if we have good sequences we need only an efficient strategy to identify similar patterns. In this case, a symbolic representation, similar to a DNA, is a good strategy.

The results indicates that an unsupervised strategy tends to achieve the worst speedup, which is the case of Random10. Although GA10 employs a supervised strategy, it is not able

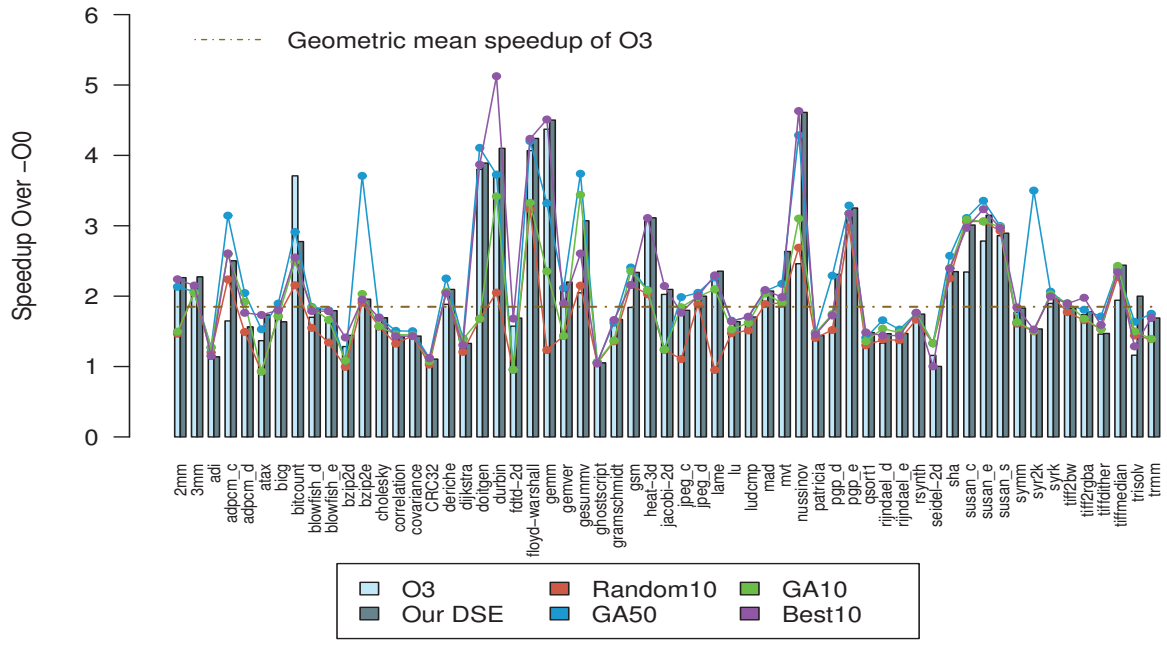

Fig. 2. The Speedup

to achieve good speedup in a short response time. Another problem which limits the speedup is not to address the problem of finding good sequences as program-dependent, which is the case of `Best10`. In fact, only our technique employs supervised and program-dependent strategies.

Figure 3 shows the metric `NPS` for all techniques.

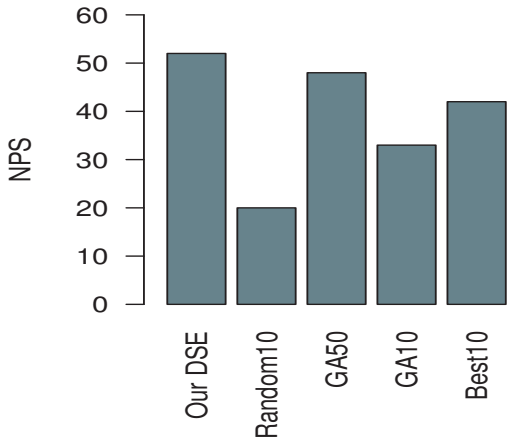

Fig. 3. Number of programs achieving speedup over the optimizations level `O3`

The metric `NPS` indicates that our `DSE` outperforms all other techniques. While our `DSE` outperforms `O3` in 88% of all programs; `Random10`, `GA50`, `GA10`, and `Best10` outperform `O3` in 34%, 81%, 56%, and 71%, respectively. Our `DSE` does not outperforms only `O3` in seven programs: BICG, BITCOUNT, JPEG_D, MAD, RSYNTH, SEIDEL_2D, SHA. However, only in three the performance lost is considerable, namely: 9.091%, 25.162%, 13.457%, respectively in BICG, BITCOUNT and SEIDEL_2D. On the other programs, the performance lost is only up to 1.498%.

Our `DSE` performs better than `GA50` in 36% of all programs evaluated, and better than `Best10` in 51% of all programs evaluated. We need to take into account one point. This metric needs to be correlated with others - for example speedup - , so that we will be able to draw precise conclusions. This correlation indicates that our `DSE` is thoroughly better than the other techniques, because it achieves, in general, the best average speedup. Our `DSE` is able to handle distinct classes of programs and not only specific classes, which is the case of `Best10`. Furthermore, our `DSE` is more efficiency in perusing the search space than `GA` and `Random`.

Summarizing, a good speedup is related to the strategy used to infer good optimization sequences, meaning that it is possible to outperform the compiler optimization levels evaluating a few points in the search space. Evaluating *several* points tends to achieve good performance (`GA50`); however, it increases the system response time, becoming the strategy impractical for real applications. In fact, there is a trade-off between the number of points evaluated and the response time, as we will see on next sections.

B. Sequences

First of all, we need to remember two points. First, `GA` finishes whether the standard deviation of the current fitness score is less than 0.01 or the best fitness score does not change in three consecutive generations, which means that

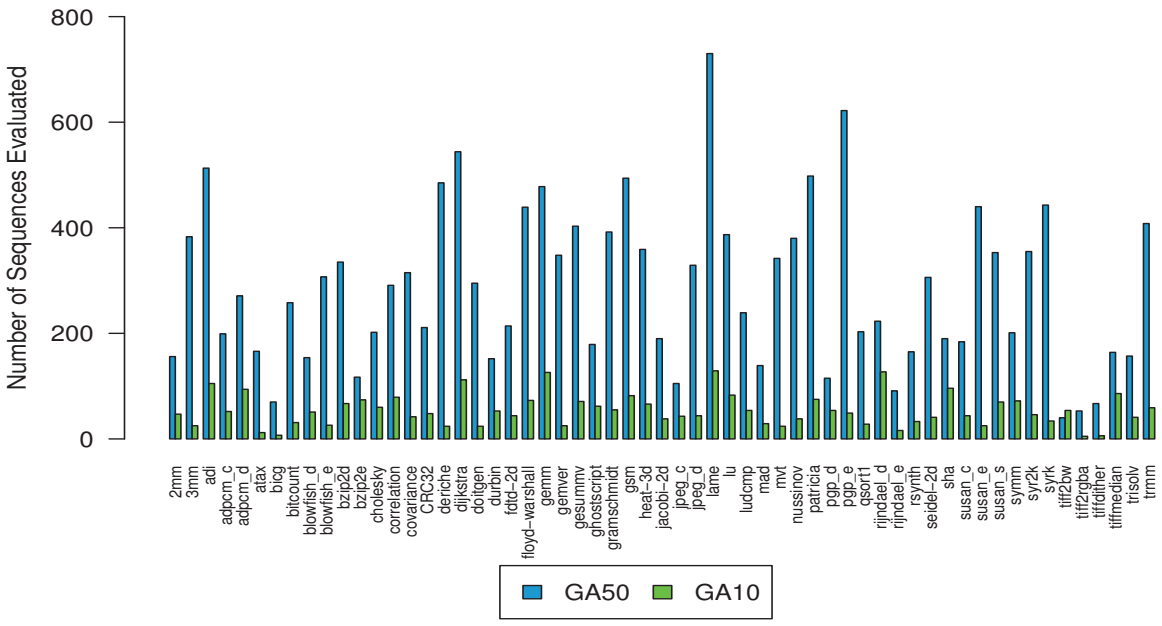

Fig. 4. Number of sequences evaluated by GA50 and GA10

GA50 evaluates at least 150 sequences, and GA10 at least 30. Second, the other techniques evaluate only 10 sequences.

The goal of comparing a technique with another that need to evaluate more sequences is to evaluate whether is possible to achieve a good speedup in a few evaluations or not. In the Figure 4, which shows the metric NoS only for GA, we can notice that the results indicates that it is possible. In fact our DSE outperforms the other techniques in terms of speedup and/or sequences evaluated.

GA10 needed to evaluate, on average, 45 sequences to achieve the stopping criterion. On the other hand, GA50 needed, on average, 243 to finish the strategy. Evaluating only 10 sequences, our DSE outperforms GA10 - in terms of speedup - and has a performance lost of just 1.143% comparing with GA50.

The results show that even though the fixed sequences (compiler optimization levels) were well engineered, they can be easily outperformed by clever strategies to guide the optimization space. This can be perceived taking into account that we used only the 10 most similar candidates, and for only seven programs we did not find a good sequence.

C. Response Time

In the Figure 5, we can notice the great difference in speed (response time) between our technique and GA. As the size of the population grows, GA takes more time and more sequences to process, while our speed remains stable with the Random10 and Best10 techniques.

This show that even with better results, GA is some orders of magnitude slower than our technique. These strategies, when you take into account the speed of high-cost genetic operations and the quality of simple strategies like Best10 or Random10, are most suitable for an environment where performance and real-time answers are needed.

The trade-off between speed and performance also should be taken into account. For example, when compiling the program SYR2K, it is possible to see that the genetic algorithm (GA50) doubled the speedup that our technique (1.534) achieved, achieving about 350% of speedup over O0. However it spent 26 times more seconds, taking roughly 122735.73 seconds, while our technique took 4653.0 seconds. It happened something similar to the program BZIP2E where our DSE got 370% faster than O0, while we obtained 195%. The time spent searching for optimization sequences, however, was about 195 times greater.

As stated before, the GA50 strategy performed better in about 86% of all programs evaluated. However, if we check for an expressive difference between the performance, we see that the genetic algorithm performed 10% better than our algorithm only in 18% of the cases. In fact, the average speedup over our speedup was 1.035 with standard deviation 0.244, which means that GA50 is only 3.5% better.

It is clear that GA50 is too slow, becoming impractical for real applications. It can be seen as the average of the ratio between the time it took, and the time our technique took, was 45 with standard deviation 51. These numbers shows that this technique is at least 45 times slower than ours.

These results strongly indicates that the trade-off offered by these time consuming algorithms like genetic algorithms are not positive. As we observed above, our technique's speedup was roughly better or similar, while GA's was orders of magnitude slower.

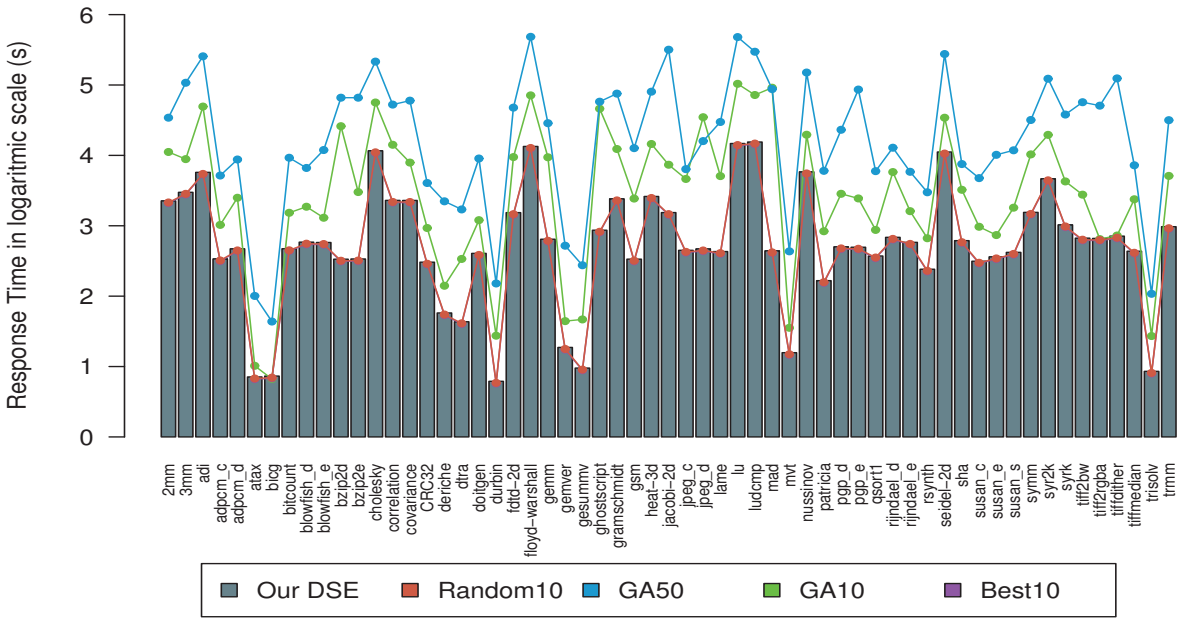

Fig. 5. The time consumed by each algorithm in order to find the best sequence.

VII. Related Work

Cavazos *et al.* [6] proposed a machine learning strategy to find compiler optimizations for a specific program. In a training stage, Cavazos's strategy randomly creates several optimization sequences for a group of training programs. After the creation of several sequences, their strategy collects the performance counters of each training programs. Based on these two pieces of information a logistic regression model is created, which will predict the sequence. The deployment stage collects the performance counters of the test program, invokes the prediction model, and finally returns the best target code.

They demonstrated that a machine learning strategy is able to outperform the compiler optimization levels. Furthermore, they also demonstrated that the use of performance counters is a good strategy to measure the similarity between two programs. Our strategy is similar to such strategy, because we also use a machine learning scheme; however, Cavazos *et al.* does not take into account hot functions, in order to guide the exploration of compiler optimization sequences. Moreover, our work does not have the overhead of executing the program to collect performance counters.

De Lima *et al.* [24] proposed the use of a case-based reasoning strategy to find compiler optimizations for a specific program. They argue that it is possible to find good compiler sequences, from previous compilations, for an unseen program. This strategy creates several sequences of compiler optimizations in a training stage. Afterwards, in a deployment stage, the strategy infers a good sequence for a test program. This step is based on the similarity between two programs. De Lima *et al.* proposed several models to measure similarity, also based on feature vectors which is composed of performance counters. They demonstrated that it is possible to infer a good sequence that achieves multiple goals; for example, runtime and energy efficency. The limitation of this work is that it does not explore the portions of code at which compiler optimizations will provide the greatest benefit.

Purini and Jain [4] proposed a strategy to find good sequences, which are able to cover several programs. This means that they do not handle this problem as program-dependent. The strategy to find several sequences consists in using random and genetic algorithms to create effective sequences. After creating several sequences, they eliminate, from each sequence, the optimizations that does not contribute to the performance. Finally, they proposed an algorithm that analyzes all sequences, and extracts the best 10 sequences. As a result, each test program is compiled using 10 sequences, and the best target code is returned. They demonstrated that it is possible to find a small group of sequences that are able to cover several programs.

Different from Purini's and Jain's technique, we handle the problem of finding good sequences as program-dependent.

Tartara and Crespi [15] proposed a long-term strategy, which its goal is to eliminate the training stage. In their strategy, the compiler is able to learn during every compilation, how to generate good target code. In fact, they proposed the use of a genetic algorithm that creates several heuristics based on the static characteristics of the test program [25]. Basically, this strategy performs two tasks. First, it extracts the characteristics of the test program. Second, a genetic algorithm creates heuristics inferring which optimizations should be enabled. They demonstrated that is possible to eliminate the

training stage, using a long-term learning. Although Tartara's and Crespi's work does not need a training stage, our work does.

Martins *et al.* [20] proposed a clustering strategy in order to find good sequences of compiler optimizations. In fact, they proposed algorithms to find good optimizations, besides algorithms to order optimizations. The strategies used by Martins *et al.* is similar to Purini's and Jain's, both use random and genetic algorithms. This means that their strategy can be considered as an iterative compilation technique, where the test program is compiled with different sequences of optimizations, and the best version is chosen. Our strategy is classified as a machine learning strategy, which tries to reduce the number of times that a test program needs to be evaluated.

Park *et al.* [3] proposed a machine learning guided approach, which they called tournament predictor. They trained a model in which given a vector of performance counters and two optimization sequences, the model is able to decide whether the first sequence performs better than the second one. They compared this model with a speedup predictor model, which was trained to calculate the speedup of a given program over a given sequence, and a sequence predictor model, which outputs a probability for each optimization, meaning the probability of activating that optimization. Park's work resulted in an increase of the speedup for some cases when compared with the speedup model, and for all cases, when compared with the sequence model. They took into account the whole program instead of only the hot functions, as we do.

Zhou *et al.* [26] used the genetic algorithm `NSGA-II` in order to minimize both execution time and code size. They showed that their approach outperforms a random approach. While, they include as a possibility in the sequences, the well-enginered compiler optimization sequences, provided by the compiler, our approach aims to surpass the well-engineered sequences.

Agakov *et al.* [27] proposed an approach independent of search algorithm, search space or compiler infrastructure, based on a predictive model from the domain of machine learning to focus the search in areas likely to give greatest performance. This strategy have the training stage, where programs are iteratively evaluated, and program features are modeled. Evaluating two different strategies (independent and Markov model), they show that these models can speed up iterative search on large spaces, outperforming a random approach. Our work differs from this in the deployment stage.

Lu *et al.* [28] proposed an algorithm that combine the Univariate Marginal Distribution Algorithm (UMDA) and the Nelder-Mead simplex method to find the near optimal parameter values. In their strategy, they formalize the iterative compilation parameter selection problem as a global optimization problem and use an estimation of distribution algorithm (EDAs) as a alternative to evolutionary algorithms. These model use a probabilistic model to guide the exploration of the search space, having a population of potential solutions and evolving them using a combination of evolutionary computation and machine learning. They demonstrated that these strategy has a better ability than a `GA` for optimization, having a higher fitness most of the times. Our work also differs from this work in the deployment stage.

VIII. Concluding Remarks

Finding a good sequence of compiler optimizations is a program dependent problem. So that, a good strategy is to inspect the features of the program, and based on these features to explore the search space looking for good optimizations. In addition, a considerable amount of running time is spent in a small portion of code. So that, the ideal features to consider is that extracted from hot functions.

In this paper, we have proposed a `DSE` of compiler optimizations. Our `DSE` finds the sequence of compiler optimizations that will be enabled during target code generation, inspecting hot functions that are represented by a symbolic representation similar to a `DNA`. Furthermore, our technique relies on a database of good sequences and use Wu-Larus and Needleman-Wunsch Algorithms.

Experimental results indicate that our design space exploration of compiler optimizations is a good technique to find good sequences of optimizations, evaluating a few points in the optimization space. Our `DSE` was able to achieved a geometric mean speedup of 2.013, while `Random10`, `GA10`, `O3`, `Best10` and `GA50`, achieved 1.632, 1.799, 1.849, 1.980 and 2.036, respectively. Therefore, they indicate that our technique surpasses several other techniques.

Ongoing work is focusing on reducing the time spent on the training stage.

References

[1] K. Cooper and L. Torczon, *Engineering a Compiler*, 2nd ed. USA: Morgan Kaufmann, 2011.

[2] S. S. Muchnick, *Advanced Compiler Design and Implementation*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1997.

[3] E. Park, S. Kulkarni, and J. Cavazos, "An Evaluation of Different Modeling Techniques for Iterative Compilation," in *In Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems*. New York, NY, USA: ACM, 2011, pp. 65–74.

[4] S. Purini and L. Jain, "Finding Good Optimization Sequences Covering Program Space," *ACM Transactions on Architecture and Code Optimization*, vol. 9, no. 4, pp. 1–23, Jan. 2013.

[5] 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 *In Proceedings of the International Symposium on Code Generation and Optimization*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 295–305.

[6] 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 *In Proceedings of the International Symposium on Code Generation and Optimization*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 185–197.

[7] S. Kulkarni and J. Cavazos, "Mitigating the Compiler Optimization Phase-ordering Problem Using Machine Learning," in *In Proceedings of the ACM international conference on Object oriented programming systems languages and applications*. New York, NY, USA: ACM, 2012, pp. 147–162.

[8] M. M. Richter and R. Weber, *Case-Based Reasoning: A Textbook*. USA: Springer, 2013.

[9] T. M. Mitchell, *Machine Learning*, 1st ed. New York, NY, USA: McGraw-Hill, Inc., 1997.

[10] S. Shalev-Shwartz and S. Ben-David, *Understanding Machine Learning: From Theory to Algorithms*. Cambridge, USA: Cambridge University Press, 2014.

[11] A. Aamodt and E. Plaza, "Case-based Reasoning; Foundational Issues, Methodological Variations, and System Approaches," *AI Communications*, vol. 7, no. 1, pp. 39–59, 1994.

[12] T. Jimenez, I. Miguel, J. C. Aguado, R. J. Duran, N. Merayo, N. Fernandez, D. Sanchez, P. Fernandez, N. Atallah, E. J. Abril, and R. M. Lorenzo, "Case-Based Reasoning to Estimate The Q-factor in Optical Networks: An Initial Approach," in *In Proceedings of the European Conference on Networks and Optical Communications*, July 2011, pp. 181–184.

[13] Y. Wu and J. R. Larus, "Static branch frequency and program profile analysis," in *Proceedings of the 27th Annual International Symposium on Microarchitecture*, ser. MICRO 27. New York, NY, USA: ACM, 1994, pp. 1–11.

[14] T. Ball and J. R. Larus, "Branch prediction for free," in *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, ser. PLDI '93. New York, NY, USA: ACM, 1993, pp. 300–313.

[15] M. Tartara and S. Crespi Reghizzi, "Continuous learning of compiler heuristics," *ACM Trans. Archit. Code Optim.*, vol. 9, no. 4, pp. 46:1–46:25, Jan. 2013.

[16] L. G. Martins, R. Nobre, A. C. Delbem, E. Marques, and J. a. M. Cardoso, "Exploration of compiler optimization sequences using clustering-based selection," *SIGPLAN Not.*, vol. 49, no. 5, pp. 63–72, Jun. 2014.

[17] S. B. Needleman and C. D. Wunsch, "A general method applicable to the search for similarities in the amino acid sequence of two proteins," *Journal of Molecular Biology*, vol. 48, no. 3, pp. 443–453, Mar. 1970.

[18] LLVM Team, "The LLVM Compiler Infrastructure," 2016, http://llvm.org. Access: January, 20 - 2016.

[19] Bechmarks Game Team, "The Computer Language Benchmarks Game," 2016, http://http://benchmarksgame.alioth.debian.org/ Access: January, 20 - 2016.

[20] L. G. A. Martins, R. Nobre, J. a. M. P. Cardoso, A. C. B. Delbem, and E. Marques, "Clustering-based selection for the exploration of compiler optimization sequences," *ACM Trans. Archit. Code Optim.*, vol. 13, no. 1, pp. 8:1–8:28, Mar. 2016.

[21] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," in *In Proceedings of the International Symposium on Code Generation and Optimization*, Palo Alto, California, Mar 2004.

[22] Louis-Noël Pouchet, "The Polyhedral Benchmark Suite," 2016, http://www.cs.ucla.edu/pouchet/software/polybench/. Acess: January, 20 - 2016.

[23] "The Collective Benchmarks," 2014, http://ctuning.org/wiki/index.php/CTools:CBench. Access: January, 20 - 2016.

[24] E. D. de Lima, T. C. de Souza Xavier, A. F. da Silva, and L. B. Ruiz, "Compiling for performance and power efficiency," in *Power and Timing Modeling, Optimization and Simulation (PATMOS), 2013 23rd International Workshop on*, Sept 2013, pp. 142–149.

[25] M. Namolaru, A. Cohen, G. Fursin, A. Zaks, and A. Freund, "Practical Aggregation of Semantical Program Properties for Machine Learning Based Optimization," in *International Conference on Compilers Architectures and Synthesis for Embedded Systems (CASES'10)*, Scottsdale, United States, Oct. 2010.

[26] Y.-Q. Zhou and N.-W. Lin, "A Study on Optimizing Execution Time and Code Size in Iterative Compilation," *Third International Conference on Innovations in Bio-Inspired Computing and Applications*, pp. 104–109, Sep. 2012.

[27] 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*, ser. CGO '06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 295–305.

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