2014 IEEE Congress on Evolutionary Computation (CEC)
July 6-11, 2014, Beijing, China

A Clustering-Based Approach for Exploring Sequences of Compiler Optimizations

Luiz G. A. Martins*†‡, Ricardo Nobre‡, Alexandre C. B. Delbem†, Eduardo Marques† and João M. P. Cardoso‡
*Faculty of Computing, Federal University of Uberlândia, Uberlândia, Brazil
Email: gustavo@facom.ufu.br
†Institute of Mathematics and Computer Science, University of São Paulo, São Carlos, Brazil
Email: acbd@icmc.usp.br, emarques@icmc.usp.br
‡Faculty of Engineering, University of Porto / INESC-TEC, Porto, Portugal
Email: ricardo.nobre@fe.up.pt, jmpc@acm.org

***Abstract*—In this paper we present a clustering-based selection approach for reducing the number of compilation passes used in search space during the exploration of optimizations aiming at increasing the performance of a given function and/or code fragment. The basic idea is to identify similarities among functions and to use the passes previously explored each time a new function is being compiled. This subset of compiler optimizations is then used by a Design Space Exploration (DSE) process. The identification of similarities is obtained by a data mining method which is applied to a symbolic code representation that translates the main structures of the source code to a sequence of symbols based on transformation rules. Experiments were performed for evaluating the effectiveness of the proposed approach. The selection of compiler optimization sequences considering a set of 49 compilation passes and targeting a Xilinx MicroBlaze processor was performed aiming at latency improvements for 41 functions from Texas Instruments benchmarks. The results reveal that the passes selection based on our clustering method achieves a significant gain on execution time over the full search space still achieving important performance speedups.**

I. Introduction

In the compilation field, it is a common practice to apply the same set of optimizations in a fixed order on each function and/or module of a program when targeting a given architecture/platform. The selection of the sequence of optimizations can have a significant impact on performance. Moreover, the effect of the optimizations is influenced by the platform used and the application [1]. Experimenting with all possible sequences of optimizations is not a practical approach. Therefore, developers typically have used their expertise to engage in a labor-intensive source code modification process and on testing multiple alternatives, aiming at achieving satisfactory compilation sequences for a specific target application. So, it is fundamental to research Design Space Exploration (DSE) schemes able to find the optimization sequence that results in suitable performance for a given function (or code fragment), considering a target platform (e.g., a microprocessor) and the set of optimizations supported by the compiler. However, time-to-market requirements for embedded systems usually impose restrictions with respect to the DSE execution time.

In this paper, we propose a method based on software code clustering for choosing a reduced set of compilation passes potentially able to reduce the latency (number of cycles) of a function or segment of code. The reduced set is used as search space in the a DSE process in order to speedup the exploration process. Our clustering technique is able to reveal similar patterns among software codes, giving important insights for determining potential groups of compiler passes. The identification of these relationships is performed using a symbolic encoding of the program, named DNA [2], and using the DAta MIning of COde REpositories (DAMICORE) methodology proposed in [3]. The usage of the DNA encoding allows identifying the main code structures, such as loops, operations and other programming constructs, that may be related to specific optimizations. First, our approach generates a distance matrix calculating the Normalized Compression Distance (NCD) [4] for each pair of DNA code. In a next step, a phylogenetic reconstruction algorithm, such as Neighbor Joining (NJ) [5], constructs a tree topology from the distance matrix. Then, an ambiguity-based clustering algorithm detects highly correlated groups of functions, extracting potential clusters hidden in the tree topology. Our approach is based on a set of reference functions, for which the near-optimal sequence of compiler optimizations have been previously determined. Each new function is joined with those reference functions and clustering is applied. The compiler passes from sequences associated with the reference functions in the cluster where the new function belongs to compose the subset of passes (reduced space), which will be used as input by a DSE strategy.

We conducted experiments in the context of the exploration of sequences of compiler optimizations for a CoSy [6] based compiler targeting a Xilinx Microblaze softcore processor. The experiments have considered 11 functions for the reference set and 41 for the evaluation. The experimental results show that when compared to the DSE considering the full set of compiler engines, our approach reduced the execution time of the DSE in 14× (458 against 6,502 seconds on average) with an increase of around 8% in the latencies achieved. The DSE achieved geometric mean speedups of 1.36 (with our clustering-based approach) and 1.44 (with the full-set of compiler engines) over a baseline instance of the compiler.

The rest of this paper is organized as follows. Section II reviews related work on the topic of determining sequences of compiler optimizations. Section III depicts our clustering-based compiler pass selection approach and introduces the fundamentals of the techniques used for representation and data mining. Section IV presents experiments performed with our approach. Finally, Section V presents some conclusions and briefly describes ongoing and future work.

978-1-4799-1488-3/14/$31.00 ©2014 IEEE

II. Related Work

Compilers for embedded computing systems are more dependent on code transformations and optimizations as the used computer architectures have typically more constraints related to memory size and organization, processor speed, and may have high-levels of customization such as the ones allowed by the use of FPGAs [7]. For these systems, a vast Design Space Exploration (DSE) may exist and the research of techniques to efficiently and effectively search this space is a hot topic. Non-trivial compiler sequences are usually needed to achieve the required results and even state-of-the-art FPGA compilers (namely the C to HDL compilers) [8] need the user to manually select most compiler sequences to achieve efficient results. The identification of compiler sequences for typical general purpose processors has been addressed in [1], but have been focused on a limited number of simple optimizations.

The common identification of compiler sequences relies on developers expertise or in a process called iterative compilation [9] [10]. Machine-learning techniques, such as Genetic Algorithms (GAs) [11], have been commonly employed to search in compile time for the best optimization sequences for specific applications. Randomized search algorithms have been also used to identify suitable compilation sequences [12]. A system called COLE was developed in [9]. It uses a multiobjective evolutionary algorithm based on SPEA2 to automatically find Pareto optimal optimization settings for GCC. The authors in [13] compared the proficiency of several machine-learning algorithms to find the best sequence of optimization passes, observing that search techniques such as GAs achieve close to optimal performance. This work was extended in [14] where they compare the ability of GA-based program and function-level searches to find the best optimization sequence for their VPO compiler. The reuse methodology proposed in [15] uses generic programming to incorporate user-defined optimizations into the compiler. The method proposed in [16] uses a form based on the characteristics of the code to identify a number of loop transformations, such as loop unrolling, loop skewing, and loop distribution. Models based on a number of code features have also revealed interesting approaches and acceptable accuracy to predict the impact of hardware compilation for a number of benchmarks [17]. Recent research has proposed novel feature-vector based heuristic techniques to quickly customize sequences to individual functions during online JIT (Just In Time) compilation [18] [19] [20].

All previous approaches are focused on the DSE. Both static and dynamic approaches reveal that the use of models and cost functions based on certain code features may allow good metric predictors and may aid to the search of the best compiler sequences. In other hand, our approach focuses on the selection of potential compiler passes in order to restrict/reduce the search space for DSE. Therefore, our work is orthogonal to theses approaches as the prune of the search space can be seen as a separate task of the exploration process. Generally, it can be used as an important initial step of DSE strategies.

Several techniques also have been used to reduce the space of potential candidate solutions. A statistical analysis of the effect of compiler options is used in [21] to prune the search space and find a single optimization sequence for a set of programs that performs better than the standard settings used in GCC. In [22], the authors show how machine learning techniques can be used to limit the search space, increasing the speed of iterative optimizations. This methodology is able to indicate, for a given program, the areas of the solution space where the search should be focused. The main advantage of this methodology is that it is independent of the solution-space, the search algorithm and the compiler used. In [23] is presented PEAK, an automated performance tuning system, that employs three heuristic algorithms to select good compiler optimization settings. It performs a previous space prune through the withdraw of transformations with negative performance effects to speedup the search. In [10], a GCC-based framework (Milepost GCC) is used to automatically extract program features and learn the best optimizations across programs and architectures. Their framework uses machine-learning techniques to correlate the new program with the closest one seen earlier to apply a customized and potentially more effective optimization combination. This approach is close to our proposal. However, our method does not need any training (i.e., it is an unsupervised approach). Moreover, it is independent of features extracted from the source code, although they can be used as input data during clustering phase to improve the algorithm accuracy. The idea is that the code patterns extracted by clustering can aid in the appropriate selection of passes, which can be used in DSE strategies to conduct/suggest efficient compiler sequences.

In [24] we explore the reduction of the search space using our clustering-based approach and we show DSE schemes which achieve similar results than a GA approach within much less execution time. In this paper we focus on the clustering-based approach, on its properties and details, and we evaluate if the improvements achieved by our approach are consequence of the clustering, the search space reduction or both.

III. Clustering-based Compiler Pass Selection

Source code has characteristics that make difficult the direct use of typical clustering methods, mainly related to their dependence of the code features for mining [3]. To solve this problem, we initially translate the source code to a symbolic representation, referred to as DNA [2], and employ a variation of the DAMICORE approach, a data mining method proposed in [3]. Our method is based on clustering techniques and allows finding patterns among codes with independence of the code size or programming language, allowing the use of intermediate representations or other information as input.

Our method starts with a set of reference functions and their corresponding sequences of compile optimizations. Each new function to be compiled, is added to the set of reference functions and the clustering is applied. Then, the distinct passes used in optimization sequences by the reference functions in the cluster with the new function is considered in the design space employed in DSE. The exploration can thus be performed using any search method implemented in the compiler environment. Fig. 1 depicts the selection process of optimization passes based on clustering and using the DNA as code representation for the functions.

The algorithm of the selection process is presented in Fig. 2. Its inputs are a repository with the code representation (Rep_{Ref}) and the sequence of compiler passes (Seq_{Ref}) for each reference function; the source code of the new function (New_{code}); and the threshold of ambiguity (T_{Amb}). The algorithm output is a reduced compiler passes space ($Space$).

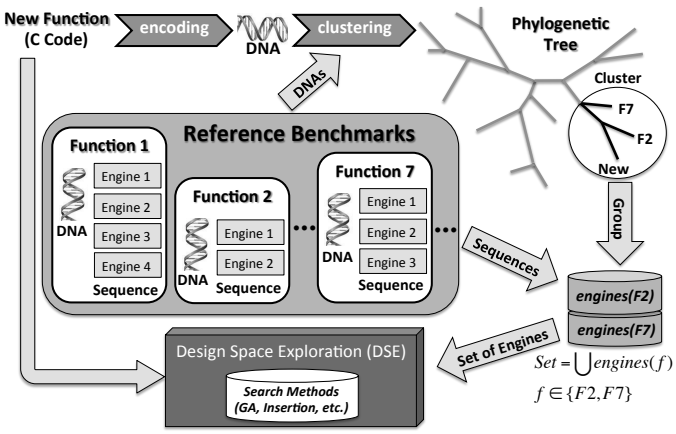

Fig. 1. Selection of compiler engines based on clustering

```
    /* DNA Encoding */
 1: tokens ← ast(New_code)
 2: for all x ∈ tokens do
 3:     Rep_New.add(translate(x))
 4: end for

    /* NCD Computation */
 5: Rep ← Rep_Ref ∪ {Rep_New}
 6: sz ← size(Rep)
 7: Dist ← zeros(sz, sz)
 8: for x = 1 to sz − 1 do
 9:     for y = x + 1 to sz do
10:         d_xy ← ncd(Rep(x), Rep(y))
11:         d_yx ← ncd(Rep(y), Rep(x))
12:         Dist(x, y) ← Dist(y, x) ← (d_xy + d_yx) ÷ 2
13:     end for
14: end for

    /* Tree Construction */
15: Tree ← starTree(Dist)
16: while size(Dist) > 2 do
17:     {A, B} ← getBestPairBranches(Dist)
18:     R ← joinBranches(Tree, A, B)
19:     computeNewBranches(Tree, R, A, B)
20:     upgrade(Dist)
21: end while

    /* Ambiguity-Based Clustering */
22: Cluster ← {}
23: for all x ∈ Rep_Ref do
24:     qt ← getNodesPath(Tree, Rep_New, x)
25:     if size(qt) < T_Amb then
26:         Cluster.add(x, qt)
27:     end if
28: end for
29: Cluster.sort(qt))
30: Space ← {engines(Cluster.node(1))}
31: for x = 2 to size(Cluster) do
32:     Seq ← engines(Cluster.node(x))
33:     for all y ∈ Seq do
34:         if y ∉ Space then
35:             Space.add(y)
36:         end if
37:     end for
38: end for
39: return Space
```

Fig. 2. Clustering-based compiler pass selection algorithm.

The selection process is based on four main steps: DNA encoding, Normalized Compression Distance (NCD) computation, phylogenetic tree construction through the Neighbor Joining (NJ) algorithm, and ambiguity-based clustering. The following subsections describe each of those steps.

A. DNA encoding

Patterns expose similarities among program codes and they may aid to determine potential sequences of compilation passes. However, the data mining over source code to identify such patterns can be a very complex task, since it needs to deal with many files, numerous lines of code and varying code structures [2]. Research has been developed in pattern-mining regarding different contexts [25] [26]. Nevertheless, many of those efforts rely on high similarity levels that hamper the identification of code patterns with subtle differences, but that may share the same optimization passes.

Here, we employ a DNA encoding due to its capability to highlight main program structures according to the transformation rules specified, allowing to identify approximate patterns present in the source code. The DNA is a symbolic representation proposed in [2], where program elements (e.g., operators and loops) are encoded in a string of symbols. The translation is based on transformation rules and produces lossy code representations. According to the used rules, this representation may reflect with higher or lower degree the sequence of tokens that identifies each program fragment.

The encoding steps are presented in Fig. 2, between *lines 1* and *4*. It involves parsing of the source code and generating the AST (Abstract Syntax Tree). Each token of the AST is then translated into a DNA representation by the transformation rules. Fig. 3 illustrates five hypothetical examples of loop structures, an example of the transformation rules employed and the corresponding DNA representations. As described in [3], this transformation rules emphasize the similarities in the loc

	C Code	DNA Representation

1	int i, o = A;
	for (i=0; i<G; i++) {
	o+= o; }
		IIEHIEZICIQIPI

2	int j, p = B;
	for (j=0; j<H; j++) {
	p*= p; }
		IIEHIEZICIQIAEI

3	int k, l, q = C;
	for (k=0; k<M; k++) {
	for (l=0; l<N; l++) {
	q*= q; } }
		IIIEHIEZICIQHIEZICIQIAEI

4	int m, n, s = E;
	for (m=0; m<K; m++) {
	for (n=0; n<L; n++) {
	s+= s; } }
		IIIEHIEZICIQHIEZICIQIPI

5	int u, v, x, y = O;
	for (u=0; u<O; u++) {
	for (v=0; v<P; v++) {
	for (x=0; x<Q; x++) {
	y*= y; } } }
		IIIIEHIEZICIQHIEZICIQHIEZICIQIAEI

Translation

Transformation Rules:	
for → "H"	++ → "Q"
identifier → "I"	+,* → "A"
+= → "P"	CMP → "C"
= → "E"	Read([ ]) → "R"
0 → "Z"	Write([ ]) → "W"

Fig. 3. Examples of a DNA encoding (source: [3]).

B. Normalized Compression Distance

The Normalized Compression Distance (NCD) is a compression-based metric proposed in [4], whose properties are based on Kolmogorov complexity theory [27]. Differently from the Kolmogorov method, it executes in a reasonable computing time. The basic idea is that two objects are considered close if one can be significantly compressed using the information in the other. Therefore, the distance $NCD(x;y)$ for each pair of functions (x and y) is the improvement due to compressing y using x as reference and compressing y from scratch, expressed as the ratio between the bit-wise length of the two compressed versions. It is calculated as follows [4]:

$$NCD(x;y) = \frac{C(xy) - min(C(x);C(y))}{max(C(x);C(y))} \quad (1)$$

where $C(\star)$ is the length of the compressed version of the file $\star$, xy is the file resulting from the concatenation of x and y.

The NCD metric does not require any knowledge about the features of the data analyzed for finding relationships. Such independence enables the NCD-based algorithm to deal with several types of program data at code level, including (but not limited to) the source codes and their representations (as DNA). It also enables one to extract information from codes at different levels of abstraction [2].

The algorithm computes the distance matrix $N \times N$ from any code representation, where N is the number of handled functions. As presented in Fig. 2, the elements of the matrix are set to zero (*line 7*) and, for each pair of functions x and y, the NCD metric is calculated through the arithmetic average between $NCD(x;y)$ and $NCD(y;x)$ (*lines 8* to *14*). This ensures the symmetry of the resulting matrix.

Table I shows the distance matrix achieved created using the DNA codes of Fig. 3. It shows that codes 2 and 3 are the most similar according to the NCD metrics, while codes 1 and 5 are the most different.

TABLE I. NCD MATRIX FROM DNA REPRESENTATIONS

DNA	1	2	3	4	5
1	0	0.29	0.36	0.21	0.40
2	0.29	0	0.20	0.34	0.31
3	0.36	0.20	0	0.28	0.25
4	0.21	0.34	0.28	0	0.35
5	0.40	0.31	0.25	0.35	0

C. Neighbor Joining Algorithm

The Neighbor Joining (NJ) [5] is a simple and computationally efficient method for constructing phylogenetic trees. This method is based on a minimum evolution principle, which aims to minimize the global tree length (sum of the length of all tree branches). Although the iterative minimization of the tree length does not ensure minimal length is achieved, the NJ method is competitive in relation to the most recent methods for this propose [3]. The basic idea is to provide a tree topology that enables recursively determining clusters composed by other clusters. Moreover, the algorithm also indicates the branch lengths of given trees. As NJ does not depend on any type of a priori knowledge about the problem domain, it is an interesting algorithm for identification of hierarchical similarities among software codes [3].

As presented in the algorithm of Fig. 2 and illustrated in Fig. 4, the tree topology is defined by successively joining pairs of neighbours. The algorithm input is a distance matrix and its output is an unrooted tree structure that describes relationships among objects. This process begins with the adoption of a star-like tree (*line 15*), where all objects (leaf nodes) are connected through a unique internal node. At each iteration, two leaf nodes are selected, based on their potential to reduce the overall tree length, and substituted by a new internal node corresponding to their common ancestor (*lines 17* to *19*). Thus, in each junction the number of leaf nodes is decreased, and the distance matrix is recalculated considering this new node (*line 20*). These iterations are performed successively while the sta

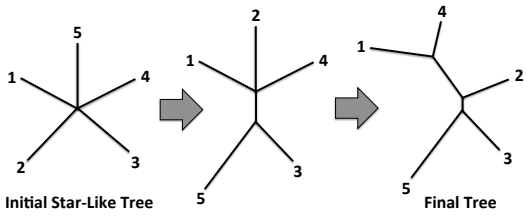

Fig. 4. Phylogenetic tree construction using Neighbor Joining. The numbers in the leaves identify the code examples in Fig. 3.

D. Ambiguity-based Clustering

Given a phylogenetic tree, it is important to verify the existence of sub-trees with a high degree of independence in their phylogeny. Thus, a hierarchical clustering algorithm is employed to extract potential clusters hidden in the tree topology.

Unlike the original approach presented in [3], where the Fast Newman (FN) [28] was used, we adopted a clustering method inspired on the measures of ambiguity of tree structures. FN is an algorithm from complex networks which is focused on discovering clusters in large-scale networks. In the other hand, an ambiguity-based approach performs better with small networks. Since we are working with a small set of reference functions (e.g., 11 in the experiments presented in this paper) plus the new function in analysis, the ambiguity-based approach is more appropriate.

Tree ambiguity can be defined from the ambiguity between two leaf nodes. Leaves i and j of a tree T are ambiguous, if there are two or more internal nodes (ancestors) between them in T, i.e., internal nodes in the unique path from i to j. In other words, both nodes are not tightly similar since they require more than one intermediate node to counterbalance their differences in relation to other object (that composes a third momentum affecting the equilibrium between i and j). The ambiguity of T (or tree ambiguity) is the result of the accumulated ambiguities involving all the pairs of leaves of T. This definition can result in imprecise evaluation of the reliability of a tree. Any tree with three nodes has ambiguity equal to one. Moreover, any tree with more than one leaf and with a odd number of leaves has at least ambiguity one. For larger trees (with at least some hundreds of leaves), this additional ambiguity may be irrelevant when comparing the reliability of clustering trees. However, it must be avoided when evaluating small trees, like those used in our environment.

We bypass this problem using the concept of strong ambiguity. Basically, two leaf nodes are strongly ambiguous if there are three or more internal nodes (ancestors) between them. Our clustering approach employs the strong ambiguity to separate the functions into groups. Therefore, the target function is joined to all reference functions separated by a couple of intermediate nodes. The number of internal nodes allowed (threshold of ambiguity) can also be modified and, as consequence, the strength of a relationship to establish a cluster can be defined according to the target domain. Fig. 5 shows the clustering over the phylogenetic tree built from the five DNA examples of Fig. 3 and considering the example 5 as the new code to be clustered.

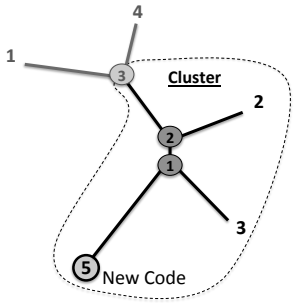

Fig. 5. Strong Ambiguity-based Clustering.

In the algorithm of Fig. 2, the ambiguity-based clustering step begins with an empty cluster (*line 22*) and for each leaf node (reference function), the path from it to the new function, i.e., the number of internal nodes between them, is computed (*line 24*). When the path length is less than the threshold of ambiguity (T_{Amb}), the node is included in the cluster (*lines 25* to *27*). Path length (*line 29*) is used to sort the clustered functions and the compiler passes are included in the search space according to the order of the functions and the passes in their corresponding sequences (*lines 30* to *38*).

IV. Experimental Results

Our experiments were performed on top of an integrated compilation environment developed in the context of the REFLECT FP7 project [29]. Moreover, the DSE schemes were programmed in LARA [29], a domain-specific aspect-oriented language, able to control and guide LARA-aware compilation and synthesis toolchains. LARA strategies define specific design patterns mainly consisting of code transformations and compiler optimizations that better suit the mapping of an application to the target architecture. In this paper, we explore a number of compiler passes provided by a CoSy (COmpiler SYstem) distribution [6].

The DSE strategies were performed without and with compiler passes pre-selection (i.e. clusted approach). The DSE without compiler-passes pre-selection (named Full DSE) was performed using a total of 49 CoSy passes in search space. Some included passes (but not limited to): constant propagation, loop invariant code motion and scalar replacement. More details about the CoSy passes can be found in [6].

The DSE carried out using the new clustering-based selection approach proposed here was called Reduced DSE. It was performed using a subset of compilation passes (13 on average). During the clustering we considered 11 reference functions: adpcm (coder and decoder), autocorrelation, bubble sort, dotprod, fdct, fibonacci, maximum value, mininum value (DSP version), pop count and sobel. This reference group is an adapted set of the benchmarks used in [3].

In this paper, two DSE strategies were developed using LARA. The objective was to investigate sequences of optimizations targeting a Xilinx MicroBlaze processor for 41 functions from image [30] and DSP [31] benchmark repositories of Texas Instruments (TI). These experiments aim to validate the efficiency and accuracy of the clustering-based compiler passes reduction approach. These strategies basically differ with respect to the technique employed in the search of the best optimization sequence for a given function. A brief description of each DSE strategy is given as follows.

A. Genetic Algorithm and Random Sampling

A DSE approach based on a standard Genetic Algorithm (GA) [11] was used to find the optimal sequence of compiler passes for the 41 functions analyzed. Basically, a GA consists in generating an initial population of random solutions with a subsequent iterative evolution of their individuals, based on evaluation and ranking. Each evolutionary step, called generation, involves the selection of the parents (pairs of individuals); the application of the genetic operators (crossover and mutation); evaluation of children (new solutions generated by operators); and the reinsertion operator (to decide the survivors). This iterative process was performed until achieving the maximum number of generations or 15 subsequent generations without improvement.

In this paper, all tests used the same GA configuration, which was empirically chosen. The GA runs over 100 generations, each one with a population size of 300 individuals. Each individual (chromosome) is a point in search space (compilation optimization sequence) represented as an array with variable size corresponding to the sequence length. Each position of the array (gene) may store an optimization pass, indicating its order of employment in the compiler. During the generation of the initial population, we applied an uniform distribution to the sequence length in order to guarantee similar quantities for all possible sizes (1 up to 16 points). The individuals generated are thus evaluated and ranked according to their fitness function. Here, the latencies (number of clock cycles) associated to the resultant code generated after applying the optimization sequence represented by individual coding were used as fitness. In each generation, a uniform crossover operator was defined with 60% of probability for applying crossover to a pair of individuals. A simple tournament selection ($Tour$ = 3) was used for the selection of the parent compilation optimization sequences. A mutation rate of 40% was applied over the population of new solutions generated by crossover. Three types of mutation operators were developed: including a new pass to a point randomly chosen of the sequence; removing the pass placed in any point of the sequence; and changing the order of two passes of the sequence. All operators have the same probability of occurrence (i.e. the choice between them is totally random) and each individual suffers a single mutation. The elitism reinsertion strategy keeps the best optimization sequences (parents and children) in population at the end of each generation.

The initial idea was to perform the GA-based DSE using both full and reduced spaces. However, in our exploratory experiments we could observe that the generation of initial population was sufficient to reach the better compiler sequences for the reduced search space. In fact, the number of compiler passes available on pruned space is relatively small (13 passes on average), thus 300 samples, even randomly generated, are able to achieve good results from that compiler passes set. In addition, it is a filtered subset of compiler passes usually covering only the passes relevant to the program under analysis. For our experiments, there was no improvement in latency during evolution and DSE was finalized at the 15th generation (stopping criterion). Thus, we chose a random sampling for the reduced space exploration. This process has exactly the same steps used in the generation of the initial population of the GA approach.

Fig. 6 presents the best execution speedups achieved for each benchmark considering the GA-based DSE over full space, and random sampling DSE over reduced space. As expected, the GA-based DSE using full compiler passes space achieved the best values when compared with random sampling. However the speedups achieved by the random sampling approach using selected passes are close to the ones achieved by the GA approach. This observation is corroborated through geometric means of 1.44 and 1.36 for GA-based and random sampling-based DSE strategies, respectively, i.e., a difference of about 0.08. This reduction in terms of speedup may be acceptable given the gains in execution time, as shown in Fig. 7. Considerable improvements were achieved with respect to the execution time when comparing DSE in full and reduced spaces. The GA-based exploration over full spaces took on average 6,502 seconds, whilst the random sampling approach over selected sequences spaces took about 458 seconds, i.e., a gain of 14× on average. In fact, the clustering provides the reduction of the search space, filtering the potential compiler passes to improve the speedup of the target function based on sequences of the reference functions. DSE with the reduced set requires fewer compilations/simulations than the full DSE, decreasing significantly the search time.

In order to evaluate the effectiveness of the clustering, we perform the reduced DSE from each cluster used during the experiments (6 clusters). The grouped functions and the number of distinct passes contained in each cluster are presented in Table II. In addition, we also use a reference cluster composed by all compiler passes of the optimization sequences associated with the reference functions, totalizing 20 passes. The passes used by the functions in each cluster were adopted as search space by the DSE strategy during the exploration.

TABLE II. CLUSTERS USED IN THE EXPERIMENTS

Clusters	Reference Functions	# Passes
1	adcpm (coder and decoder), fdct and sobel	17
2	autocorrelation and pop count	4
3	min. value and bubble sort	7
4	min. value, dotprod, max. value and fibonacci	10
5	fdct and sobel	11
6	dotprod	4
Reference	all reference functions	20

Fig. 8 presents the geometric means of the speedups achieved for all 41 TI's functions considering each cluster. As can be observed, our approach obtained the best result between the clusters (1.357), achieving speedups very close to the reference values (1.364). The DSE using the reference cluster as search space is able to find the best accessible speedups.

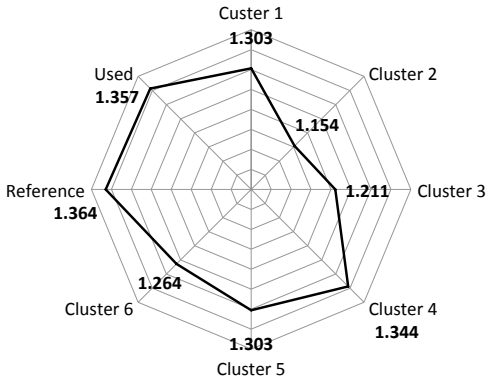

Fig. 8. Geometric mean of the speedups according to the cluster used.

Considering each function individually, our clustering-based approach did not achieve the reference speedup for only 5 (out of 41) functions. Fig. 9 presents the speedups achieved by DSE for these functions, considering the reduced search space provided by each cluster. Except for *dsp_mat_tr*, our approach achieved values very close to the reference ones. For *dsp_mat_tr*, the speedup achieved using the reduced space provided by our clustering-based approach was 1.236. The clusters more appropriate for *dsp_mat_tr* are clusters 1 and 4. In both cases the results achieve the reference speedup (1.513).

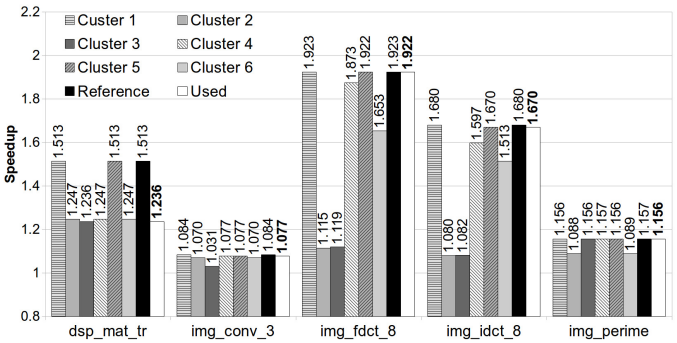

Fig. 9. Speedups for the functions which our clustering-based approach achieved code with higher latencies than when using the compilation engines of the reference set.

B. Insertion Algorithm

Insertion algorithm is an iterative greedy algorithm presented in [32] that starts with the simulation of the target function without optimizations for obtaining the reference latency. Then, the algorithm traverses the compiler passes space sequentially, in a predefined order, constructing a solution by inserting just a compiler pass at a time and verifying the resultant latency of the optimized code. If an improvement occurs, the pass is integrated in the current solution. Otherwise, it is removed and the next pass in the sequence is tested. In full DSE, the sequence order is arbitrarily defined. In reduced DSE, the sequence order is defined in relation to the proximity among the functions in the phylogenetic tree.

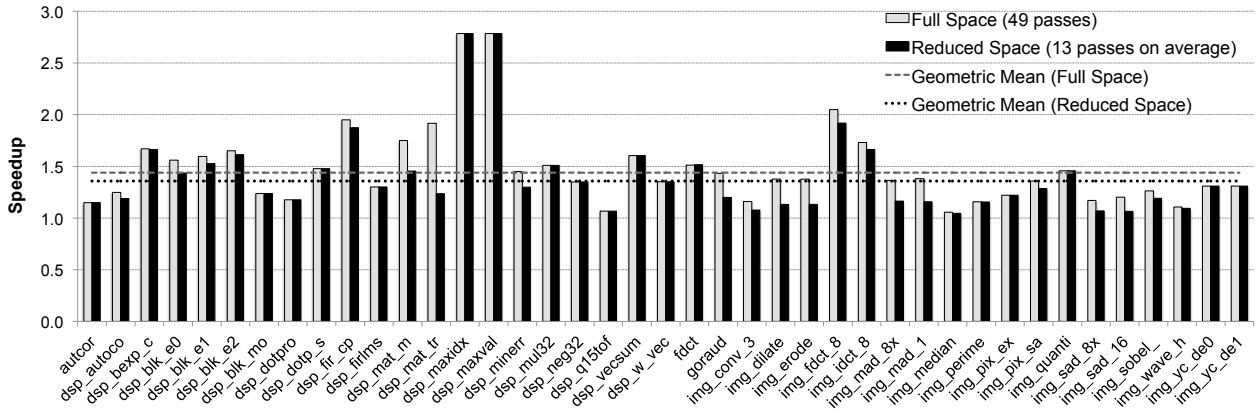

Fig. 6. Speedups obtained using GA-based DSE with full space and Random Sampling-based DSE with pruned space.

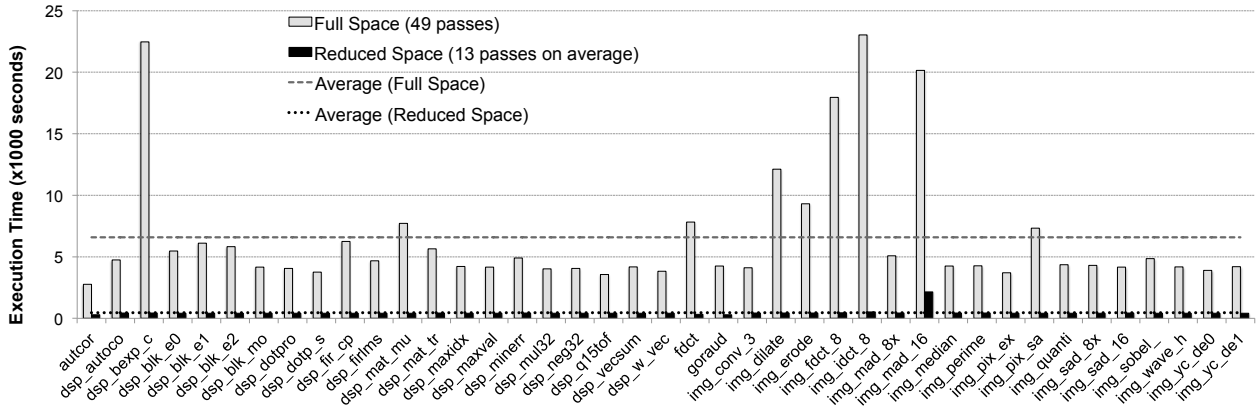

Fig. 7. Execution time of the GA-based DSE with full space and Random Sampling-based DSE with pruned space.

In both approaches (full and reduced DSE), when testing the inclusion of a new compiler pass in the current solution, the algorithm must evaluate the pass in all possible positions of the current solution, aiming at achieving the best sequence. Since the compiler passes are processed in the same order they appear in the search space, the approach results can be influenced by the arrangement used. For minimizing this problem, it is necessary to traverse the passes space a number of times. The number of iterations is an input of the algorithm. The employment of a small number of iterations may increase the dependence between sequence order and results, limiting the coverage of the space. On the other hand, a high number of iterations increase the execution time of DSE. Here, we adopted 3 iterations for both approaches (full and reduced), since higher values did not allow better results. The algorithm also stops when there is no improvement between iterations.

The goal of the DSE executions based on *insertion* algorithm is to collate the behavior of the exploration in both scenarios (full and reduced space) and confirm if the employment of the passes selection approach really reduces the execution time. The use of pruned space resulted in speedups close to those obtained for the full space in almost programs. Full DSE geometric mean obtained was 1.36 against 1.28 of the Reduced DSE, representing a difference of 0.08. The biggest difference occurred with dsp_mat_tr function. Analyzing the sequences resulted, we observed that the Full DSE returned a sequence of 6 compiler passes, while the Reduced DSE employed 2 passes, being only the *strength* passes present in both sequences.

The DSE using the reduced space was very fast. As expected, the prune of search space provided a significant speedup in the DSE execution time (around 7×) in relation to full space. This is due to the amount of passes used and the sequences length tested during the execution. The length of sequences generated from the Reduced DSE is in average smaller than the sequence-length than Full DSE (3 passes against 5 passes).

C. Analysis of Results

When we compare the four approaches (GA with full space, random sampling with reduced space, and *insertion* algorithm with full and reduced space), as expected, the full space GA-based DSE is the best method in terms of achieved performance (44% of performance improvement), but is also the most time consuming (6,502 seconds). On the other hand, the *insertion* algorithm with reduced space is the faster one (127 seconds), sacrificing the achieved performance (increase 28%). The other two methods (*insertion* algorithm with full space and random sampling based on reduced space) are equivalent in terms of performance (around 36% of improvement), but the last approach is the fastest (14× against 9×).

V. Conclusions

The choice of the sequences of compiler passes can have a significant impact on performance and is platform and

application dependent. Therefore, the adoption of a Design Space Exploration (DSE) scheme that aids embedded system developers is of paramount importance. This paper presented a clustering-based selection approach for reducing the number of passes used during the exploration of optimization sequences.

The experiments show that our clustering-based approach is able to search software similarities at code level. Its use in DSE allows to efficiently explore the design space points considering distinct compilation sequences and addressing performance improvements. When compared with the results achieved by a DSE strategy using a Genetic Algorithm, our approach using random sampling and *insertion* algorithms allowed significant reductions of execution time of the DSE (around 14× and 51×) still achieving significant performance improvements (36% and 28%) over the results without optimizations. Considering that a well-setting Genetic Algorithm (GA) generally results on close to optimal results [13] [20], it is expected that a GA-based DSE using full space provides close to optimal sequences of optimizations and the achieved speedups can thus be used as goals for the other strategies. In this context, the performance of a random sampling-based DSE using reduced space achieved results close to a GA-based approach and spending much less execution time.

Ongoing work is focused on evaluating the impact of the number and type of reference functions and of the DNA transformation rules in the clustering. For future work we intend to explore sequences of compiler optimizations in the context of hardware accelerators targeting FPGAs.

ACKNOWLEDGMENTS

This work has been partially supported by FCT (Portuguese Science Foundation) under research grants SFRH/BD/82606/2011, and FEDER/ON2 and FCT project NORTE-07-124-FEDER-000062; and FAPEMIG (PEE-00443-14). LGAM has a scholarship granted by CAPES (process: 0352/13-6) which made possible a 1-year long visiting period to FEUP and his contribution to the work presented in this paper. The FEUP authors also acknowledge the CoSy license and technical support granted by ACE Associated Compiler Experts bv, The Netherlands.

REFERENCES

[1] L. Almagor, et al., "Finding effective compilation sequences", *ACM conf. on Languages, Compilers, and Tools for Embedded Systems (LCTES'04)*, vol. 39, pp. 231-239, 2004.

[2] A. Sanches and J. M. P. Cardoso, "On Identifying Patterns in Code Repositories to Assist the Generation of Hardware Templates", *20th Int. conf. on Field Programmable Logic and Applications (FPL'10)*, pp.267-270, 2010.

[3] A. Sanches, J. M. P. Cardoso and A. C. B. Delbem, "Identifying Merge-Beneficial Software Kernels for Hardware Implementation", *Int. conf. on Reconfigurable Computing and FPGAs (Reconfig'2011)*, pp.74-79, 2011.

[4] A. R. Cilibrasi and A. P. Vitanyi, "Clustering by compression", *IEEE Trans. Information Theory.*, vol. 51, no. 4, pp. 1523-1545, 2005.

[5] J. Felsenstein, *Inferring phylogenies.* Sinauer Associates, Inc, 2003.

[6] "ACE CoSy Compiler Development System", Available: http://www.ace.nl/compiler/cosy.html, (accessed in 18/10/2012).

[7] J. M. P. Cardoso, P. Diniz, and M. Weinhardt, "Compiling for Reconfigurable Computing: A Survey", *ACM Computing Surveys*, vol. 42, no.4, pp.1-65, 2010.

[8] B. Buyukkurt, et al., "Impact of high-level transformations within the ROCCC framework", *ACM Trans. on Architecture Code Optimization (TACO)*, vol. 7, no. 4, pp.1-36, 2010.

[9] K. Hoste and L. Eeckhout, "Cole: compiler optimization level exploration", *6th Int. Symp. on Code Generation and Optimization (CGO'08)*, pp.165-174, 2008.

[10] G. Fursin, et al., "Milepost GCC: machine learning enabled self-tuning compiler", *Int. J. of Parallel Programming*, vol. 39, pp.296-327, 2011.

[11] D. E. Goldberg, "Genetic Algorithms in Search, Optimization and Machine Learning", 1st ed., Addison-Wesley Longman, MA, USA, 1989.

[12] K. D. Cooper, et al., "Exploring the structure of the space of compilation sequences using randomized search algorithms", *The Journal of Supercomputing*, vol. 36, no. 2, pp.135-151, 2006.

[13] P. A. Kulkarni, D. B. Whalley and G. S. Tyson, "Evaluating heuristic optimization phase order search algorithms", *IEEE Int. Symp. on Code Generation and Optimization (CGO'07)*, pp.157-169, 2007.

[14] P. A. Kulkarni, M. R. Jantz, and D. B. Whalley, "Improving both the performance benefits and speed of optimization phase sequence searches", *ACM conf. on Languages, Compilers, and Tools for Embedded Systems (LCTES'10)*, pp.95-104, 2010.

[15] J. J. Willcock, A. Lumsdaine and D. J. Quinlan, "Reusable, generic program analyses and transformations", *8th Int. conf. on Generative Programming and Component Engineering (GPCE'09)*, pp.5-14, 2009.

[16] O.S. Dragomir, "K-loops: Loop Transformations for Reconfigurable Architectures", *PhD thesis*, TU Delft, Faculty of Elektrotechniek, Wiskunde en Informatica, Delft, Netherlands, 2011.

[17] R. J. Meeuws, C. Galuzzi and K. L. M. Bertels, "High Level Quantitative Hardware Prediction Modeling using Statistical methods", *IEEE Int. conf. on Embedded Computer Systems*, pp.140-149, 2011.

[18] J. Cavazos and M. F. P. O'Boyle, "Method-specific dynamic compilation using logistic regression", *ACM Int. conf. on Object-Oriented Programming Systems, Languages, and Applications*, pp.229-240, 2006.

[19] R. Sanchez, et al., "Using machines to learn method-specific compilation strategies", *Int. Symp. on Code Generation and Optimization (CGO'11)*, pp. 257-266, 2011.

[20] Michael R. Jantz and Prasad A. Kulkarni, "Performance potential of optimization phase selection during dynamic JIT compilation", *9th ACM Int. conf. on Virtual Execution Environments(VEE'13)*, pp.131-142, 2013.

[21] M. Haneda, P. M. W. Knijnenburg and H. A. G. Wijshoff, "Optimizing general purpose compiler optimization", *2nd conf. on Computing frontiers (CF'05)*, pp.180-188, 2005.

[22] F. Agakov, et al., "Using Machine Learning to Focus Iterative Optimization", *Int. Symp. on Code Generation and Optimization, (CGO'06)*, pp. 295-305, 2006.

[23] Z. Pan and R. Eigenmann, "PEAK: a fast and effective performance tuning system via compiler optimization orchestration", *ACM Trans. on Programming Languages and Systems*, vol. 30, no. 3, pp.1-17, 2008.

[24] L. G. A. Martins, et al., "Exploration of Compiler Optimization Sequences using Clustering-Based Selection", *ACM conf. on Languages, Compilers, and Tools for Embedded Systems (LCTES'14)*, 2014.

[25] T. Wheeler, and J. Kececioglu, "Multiple alignment by aligning alignments", *15th ISCB conf. on Intelligent Systems for Molecular Biology, Bioinformatics*, vol. 23, no. 13, pp. i559-i568, 2007.

[26] C. K. Roy, J. R. Cordy, and R. Koschke, "Comparison and Evaluation of Code Clone Detection Techniques and Tools: A Qualitative Approach", *Science of Computer Programming.*, vol. 74, no. 7, pp. 470-495, 2009.

[27] M. Li and P. M. B. Vitanyi, *An introduction to Kolmogorov complexity and its applications.* 2nd ed. Springer-Verlag, 1997.

[28] M. Newman, *Networks: An Introduction.* Oxford Univ. Press, Inc, 2010.

[29] J. M. P. Cardoso, et al. (eds.), *Compilation and Synthesis for Embedded Reconfigurable Systems: An Aspect-Oriented Approach.* Springer, 2013.

[30] Texas Instruments, "TMS320C64x Image/Video Processing Library", 2003.

[31] Texas Instruments, "TMS320C64x DSP Library: Programmer's Reference", 2003.

[32] Q. Huang, et al., "The Effect of Compiler Optimizations on High-Level Synthesis for FPGAs", *IEEE 21st Int. Symp. on Field-Programmable Custom Computing Machines (FCCM'13)*, pp. 89-96, 2013.