2010 Ninth International Conference on Machine Learning and Applications

Spatial Based Feature Generation for Machine Learning Based Optimization Compilation

Abid M. Malik
Institute of High Performance Computing, Singapore
abidmuslim@gmail.com

***Abstract*—Modern compilers provide optimization options to obtain better performance for a given program. Effective selection of optimization options is a challenging task. Recent work has shown that machine learning can be used to select the best compiler optimization options for a given program. Machine learning techniques rely upon selecting features which represent a program in the best way. The quality of these features is critical to the performance of machine learning techniques. Previous work on feature selection for program representation is based on code size, mostly executed parts, parallelism and memory access patterns with-in a program. Spatial based information–** *how instructions are distributed with-in a program* **–has never been studied to generate features for the best compiler options selection using machine learning techniques. In this paper, we present a framework that address how to capture the spatial information with-in a program and transform it to features for machine learning techniques. An extensive experimentation is done using the SPEC2006 and MiBench benchmark applications. We compare our work with the IBM Milepost-gcc framework. The Milepost work gives a comprehensive set of features for using machine learning techniques for the best compiler options selection problem. Results show that the performance of machine learning techniques using spatial based features is better than the performance using the Milepost framework. With 66 available compiler options, we are also able to achieve 70% of the potential speed up obtained through an iterative compilation.**

***Keywords*-compiler option selection, compiler optimization, machine learning, auto- tuning, graph similarity**

I. Introduction

Modern compilers provide various optimization options for users to choose in order to obtain better running time for a given program. Proper selection of optimization is not easy for average users. Modern compilers provide various levels of optimizations. For example, three levels, O1, O2 and O3, are provided by the GNU Compiler Collection (GCC) [1]. A higher optimization level will give better binary code than a lower optimization level. These optimization levels combine various compiler options using some heuristic. However, these optimization levels exploit only a portion of the available options. There is still a large potential that a better efficiency can be obtained by exploiting the rest of the available optimization options.

With $\mathbf{k}$ compiler options, there exist $\mathbf{2^k}$ different ways compiler options can be selected for compiling a program. In order to get an optimal solution, an exhaustive search is needed. Studies have been done on searching algorithms to select a set of compiler options for optimal performance. Various strategies have been adopted that include iterative compilation searching [3] and predictive modeling [12]. Iterative compilation is a random searching of a compiler optimization space for a particular program, evaluating as many points as possible within a constrained time. The evaluation consists of simply compiling the code with a given set of optimizations, executing the binary and recording the run-time. The optimizations which provide the fastest run-time are considered the best for the particular program being compiled.

Predictive modeling techniques use features to characterize an optimization space. Using known correct points, a model is built. The model is used to predict the best optimization options in the search space. Machine learning techniques have been used to build a predictive model for the best compiler options selection problem [9]. Machine learning techniques rely upon selecting relevant features of the search space. The quality of these features is critical to performance of machine learning techniques. Previous work on feature selection only captures characteristic of a program which is based on code size, mostly executed part, parallelism and memory access [8] patterns with-in a given program. However, no work has been done on feature generation which captures spatial information with-in a program. By spatial information, we mean how different instructions are distributed with-in a program. This information is important for many compiler optimizations e.g. instruction scheduling and register allocation. This information is stored in a compiler in the form of a data structure known as Data Flow Graph (DFG). DFG is a directed labeled graph. Each node represents an instruction and each edge represents data dependency between two instructions in a DFG. The distribution of instructions in a DFG is an important information which tells how data flows with-in a program.

In this paper, we present a framework which generates features based on the spatial information of a DFG. These features can be used for selecting the best compiler options using machine learning techniques. We test our framework against the recently announced IBM Milepost-gcc framework. The Milepost framework gives a comprehensive set

978-0-7695-4300-0/10 $26.00 © 2010 IEEE
DOI 10.1109/ICMLA.2010.147

of features for representing a program that can be used to automate the compiler option selection process using machine learning techniques. Experimental results show that our framework outperforms the IBM Milepost-gcc feature framework on most of the applications using Decision Tree (DT) and Support Vector Machines (SVMs) learning. With our framework, using 66 compiler options in the GNU GCC, we are also able to gain 70% of the maximum speedup obtainable by an iterative compilation search using 1000 iterations.

The paper is organized as follows: Section II of the paper gives related work. Section III shows how machine learning is applied to compiler option selection problem. Section III-A presents a motivating example, showing why an approach like ours is needed. Section IV gives our framework for generating spatial based features. Section V gives experimentation. Section VI discusses the results and Section VII gives conclusion and future work in this area.

II. Related Work

One of the first researchers to incorporate machine learning into compiler for optimization were McGovern and Moss [10] who used

reinforcement learning for scheduling of straight-line code. Cavazos et al. [6] extended this idea by learning whether or not to apply instruction scheduling. Stephenson and Amarasinghe [16] looked at tuning the unroll factor using supervised classification techniques such as K-Nearest Neighbor (KNN) and SVMs.

Subsequent researchers have considered predictive models using machine learning techniques to automatically tune a compiler for an existing micro- architecture. These models use program's features to focus the search of optimization space in promising areas. Agakov et al. [2] use code features to characterize a given program while Cavazos et al. [5] investigate the use of performance counters. Leather et al. [13] give grammar to select the features to represent a program to tune unrolling optimization. Christophe et al. [8] use hardware features for selecting the best compiler options for a given architecture.

Recently the Milepost-gcc framework has been developed by the IBM Haifa to drive the compiler optimization process based on machine learning. The framework is very comprehensive in terms of capturing all important characteristics of DFG for a given program. Interested readers can consult the work by Fursin et al. [9] for complete list of features. However, the work fails to capture the spatial information of DFGs. In this work, we use the IBM Milepost-gcc framework as a reference to compare the quality and performance of our framework.

III. Use of Machine Learning in Compiler

In this section, we briefly describe how machine learning is deployed with-in a compiler for selecting good options for a better performance. The state of a compiler is examined before any optimization is applied. The Static Single Assignment (SSA) state with-in a compiler is that point [9]. For more detail on SSA state, please see [11]. Data which is important for various optimizations is collected at SSA level. This data includes data structures like abstract syntax tree, control flow graph and data flow graph. The data is transferred into a set of features which is used by machine learning tools. The quality of features is important for the quality of machine learning tool. A training set is generated using a number of benchmark applications. These applications are transferred into feature vectors using the feature set. The benchmark applications are compiled multiple times; each time with different compiler options and by running the newly compiled program and discovers which options are the best for a given application. The training set consists of tuples. Each tuple consists of a feature vector of a program and the best compiler options for the program. The training set is given to a machine learning tool to build a model. The model's job is to predict the best compiler options for new feature vectors from unseen applications.

The advantage of using a machine learning technique is that a compiler writer is not required to develop a new heuristic for selecting the best compiler options for every new architecture. The process can easily be repeated whenever there is a change in the underlying architecture. However, effort has now been transferred from tuning the heuristic by hand to creating the right features for the machine learning techniques.

A. Motivation Example

In this section, we demonstrate that features based on the spatial information of data flow graphs is important for the performance of machine learning techniques for the best compiler options selection problem. Figure 1 shows two data flow graphs with different structure layouts. The number besides an edge is latency which is the minimum number of cycles by which two instructions should be separated in any legal schedule for a given program. Assume, we have three types of instructions; A,B and C. For simplicity, consider six features: f_1) **number of instructions** f_2) **number of edges** f_3) **critical path distance–the minimum number of cycles by which a root node and a sink node**[1] **in a DFG should be separated in any legal schedule for the DFG when there are unlimited physical resources in a processor–** f_4) **number of instructions of type** A f_5) **number of instructions of type** B f_6) **number of instructions of type** C. These features are also included in the IBM Milepost-gcc framework [9]. With this set of features, the two graphs will have same feature vector ,i.e., $< f_1 = 5, f_2 = 4, f_3 = 4, f_4 = 3, f_5 = 1, f_6 = 1 >$.

[1]A node with-out successor nodes is a sink node. A node with no predecessor nodes is a root node.

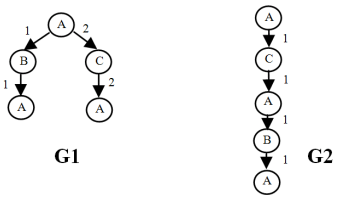

Figure 1. Data Flow Graphs with different structure layouts. Graph G_1 has parallelisam while graph G_2 is a serial code. We have three types of instructions; A, B and C. The number beside each edge is latency.

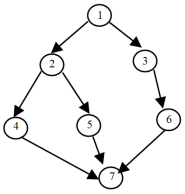

Figure 2. Data Flow Graph G_3. Each node is an instruction of same type. Each edge has latency of unit cycle. Node 1 is the root node at the level 0. Node 7 is the sink node at the level 3.

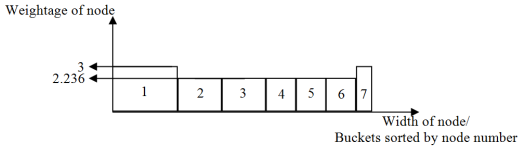

Figure 3. Histogram for the graph G3. The horizontal axis is node number and the vertical axis is weight-age of each node. In our case, its the resultant critical path distance from the sink and root nodes.

Suppose, X is a set of compiler options which is the best for G_2. If G_2 is used as a training set to build a model, the same set of compiler options will be predicted for G_1. However, we can see that G_2 is a serial code and can not be benefited from optimizations that are targeted towards multi-issue processor. G_1 has parallelisam and can be benefited from such compiler optimizations. Using the spatial information of each instruction type, one can easily differentiate the two graphs. Consider levels of each node. The level of a node is the maximum number of edges between a root node and that node. In G_1, instruction A is at level 0 and 2. Instruction B and instruction C are at level 1. In G_2, instruction A is at level $0, 2$ and 4. Instruction C is at level 1 and instruction B is at level 3.

IV. Our Framework

Program features which are used for selecting the best compiler options using machine learning can be classified into four classes: **1) code size based features 2) hot instructions based features 3) parallelisam based features 4) memory access based features.** These features are considered good indicators of a program's performance. Features based on code size, like number of instructions, control the behavior of cache. Features based on hot instructions–*instructions that are mostly executed*–control execution time of a program. Features based on parallelism control optimizations like instruction scheduling, register allocation, loop unrolling etc. Features like number of predecessors or successors of an instruction and critical path distance etc. come under this class. Memory access patterns in a program control behavior of cache. Features like number of load and store instructions come in this class. All this information can be determined at SSA level in a compiler. However, none of the afore-mentioned classes capture the spatial information with-in a DFG. In this work, we try to remove this gap of information. We use the concept of histogram to capture the spatial information in a DFG.

A histogram is a bar graph which captures weight-age of each element in a data set. People have used this data structure in image processing to capture spatial information [14]. In our case, DFG is a data set and each node of the graph is an element of the data set. The weight-age of each node is how it is located with-in a given DFG. Consider the graph G_3 in Figure 2. For simplicity, we assume that there is only one type of instruction in G_3 and each edge has latency of one cycle. To build a histogram for G_3, each node will have a bucket. The height of each bucket represents the weight-age of each node. For our work, the location of a node with respect to root and sink nodes of a DFG is a key information. We calculate the weight-age of each node using Equation 1.

$$W_i = \{{D_{ir}}^2 + {D_{is}}^2\}^{1/2} \quad (1)$$

Where D_{ir} is the critical path distance of Node i from the root node r and D_{is} is the critical path distance of Node i from the sink node s. Consider Node 2 in Figure 2. The node has critical path distance of 1 cycle from the root node ,i.e., Node 1 and has critical path distance of 2 cycles from the sink node, i.e., Node 7. Using Equation 1, the weight-age of Node 2 is 2.236 cycles. We consider **critical path distance to sink node + 1**[2] as the width of each bucket. Therefore, the bucket for Node 2 has width of 3 cycles. The bucket for Node 1 has weight-age of 3 cycles and width of 4 cycles. The bucket for Node 7 has weight-age of 3 cycles and width of 1 cycle. Bucket for Node 3 has weight-age of 2.236 cycles and width of 3 cycles. The buckets for Node $4, 5$ and 6 have weight-ages and widths of 2.236 cycles and 2 cycle respectively. Figure 3 gives a histogram for G_3. In order to give some uniqueness to the histogram, we sort it according to the weight-age of node. If two or more nodes have same weight-age then we sort them according to the width of bucket as shown in Figure 4. In **Engineering**

[2]We add 1 so a bucket for a sink node can have a width of at-least 1 cycle.

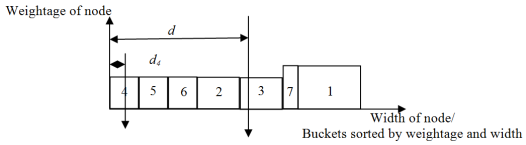

Figure 4. Sorted Histogram for the graph G3; according to the weight-age of each node.

Mechanics, concept of **Center of Area** is used to capture the point of concentration of cross- sectional areas of structures. We use the center of area of histogram to capture the node distribution with- in a DFG. This can be calculated using Equation 2.

$$d = \frac{\sum A_i \times d_i}{\sum A_i} \tag{2}$$

Where d is the distance of center of area of histogram from some reference axis and d_i is the distance of center of area of bucket i from the same reference axis. A_i is the area of bucket i and can be calculated by multiplying the weight-age of each bucket by its width. Using Equation 2, **d** in Figure 3 is 9.765 cycles. In Figure 3, d_4 is the distance of center of area for bucket 4.

Two isomorphic graphs will have same distribution of sorted histograms and hence will have same distances of center of areas from some reference axis. However, the reveres is not true ,i.e., two histograms having same distances of center of areas from some reference axis might not be isomorphic. The other important observation is that if two graphs are similar but not isomorphic then the distances of their center of areas will be very close to each other from some reference axis. This observation is the base of our technique. If two graphs are not isomorphic but are similar to some extent then this similarity can be quantified using the center of area of histogram.

Data flow graph of a program is a directed labeled graph. An important question is: how to capture distribution of different labels or instruction? This can be done by preparing a histogram for each label and then calculating the center of area of histograms for each label. Consider the two DFGs in Figure 5 . Both graphs have three types of instructions i.e. A, B and C. Node $A1$ means that Node 1 is of type A. As we have three types of instructions so we will have three histograms ,i.e., one for each instruction type.

Figure 6 (a) (b) and (c) show three sorted histogram distributions for instruction type A, B and C respectively for $G1$ and $G2$. We calculate the distance of center of area for each histogram from the vertical axis. The distance of center of area of each histogram will become a feature. Using these features, we will have a feature vector $< 3.731, 1, 1 >$ for G_1 and a feature vector $< 4.825, 2, 1 >$ for G_2. On the other hand, if we use the Milepost-gcc framework, feature set from Section III-A, then feature vectors for both the graphs are

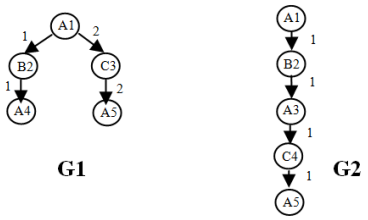

Figure 5. Two data flow graphs with different structural layouts. G1 has parallelisam while G1 is a serial code. In the graphs, we have three types of instructions A, B and C. Node $A1$ means that Node 1 is of type A.

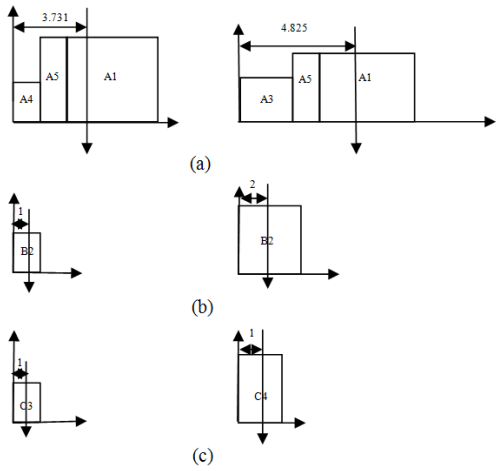

Figure 6. Histograms for for G1 and G2. (a) for Instruction A, (b) for Instruction B (c) for Instruction C. The histograms in the left column belong to G1. The histograms in the right column belong to G2.

same. Hence, the Milepost feature framework is misleading in this case.

The Milepost work gives 55 features ,i.e., the length of feature vector is 55. For our approach the length of feature vector is equal to the number of available labels or instructions in the data set. For this, we scanned the SPEC2006 benchmark applications for all possible operations in DFGs. We found 110 operations or labels. Therefore, we can have a feature vector of length 110. However, in order to avoid the **curse of dimensionality** phenomenon in the machine learning technique, we calculated the frequency of each operations in the data set and took the top 55 high frequency labels as features for our approach.

V. Experimentation

In this section, we briefly describe the experimental setup. We used GCC 4.4.1 to extract DFGs at SSA level. We used the IBM Milepost- gcc to implement the Milepost framework. We used 66 GCC optimization options used by the work [15]. We used C applications from the MiBench and SPEC2006 benchmark suites. These experiments were

run on the Intel dual core running at 2.0 GHz with 4.0 Gb of RAM. We used DFG of a hot function–*a function which is mostly executed*–to represent a program. We used the **gprof** tool to determine hot functions in the benchmark applications.

We build a training set using the SPEC2006 C applications by finding the best compiler options for each application using the Genetic Algorithm (GA) from the work [7]. We used 1000 evolutions for the GA approach. Each run was repeated five times so the speedups were not caused by cache priming etc.

In our experimentation we tried to keep the implementation similar to the work [9]. We used DT learning as it was also used by the work [9] as well. We used SVMs as they are considered the best tool for classification and regression problems. We implemented both linear and non-linear SVMs using the **SVMlight** freely available on the web [17]. We used the default values for various parameters set by the SVMlight. However, for the non-linear SVMs, we used radial basis function kernel with $\sigma = 1$ and the upper bound parameter (C) of the SVM equal to 10. We used the **C4.5 algorithm** for DT implementation [4]. The same was used by the Milepost work. Again, we used the default values given by the developer for various parameters.

We considered 66 compiler options. We assumed that they were independent of each other. The same assumption was adopted by the Milepost work. We considered each option as a class and tried to learn a model for it using DT and SVMs. We trained 66 models using DT and SVMs ,i.e., one for each compiler option.

VI. Results

In this section, we discuss our results. Figure 7 and 8 show the performance of DT and SVMs learning using our framework as compare to the Milepost framework. The vertical axes of graphs give the percentage improvement in the execution time of a program over the execution time when the same program is compiled with the GCC compiler using -O3 level. This can be calculated using Equation 3.

$$X = \frac{T_1 - T_2}{T_1} \times 100 \tag{3}$$

Where X is the percentage improvement in the execution time of a program, T_1 is the execution time when the same program is compiled using the - O3 level[3] and T_2 is the execution time of the same program when compiled with optimization options predicted by a model.

We used C applications in the SPEC2006 benchmark for building a training set. The Mibench C applications were used as a test set.

Figure 7 gives the comparison of our approach using the top 55 features against the Milepost approach. Again, our approach out performs the Milepost work on most of the applications. The maximum and minimum gain over the -O3 level by the Milepost-gcc is 2.4% and 1% respectively using DT and 2.7% and 1.2% respectively using linear SVMs. The maximum and minimum gain over the -O3 level by our approach is 2.9% and 1% respectively using DT and 3.2% and 1.2% respectively using linear SVMs. With the Milepost framework, the average gain in execution time is 1.6% and 1.91% using DT and linear SVMs respectively. With our approach, the average gain in execution time is 1.7% and 2.2% using DT and linear SVMs respectively. On can note that we are able to achieve 70% of iterative approach using our features with the SVMs.

[3]-O3 is the highest level of optimization in the GCC compiler.

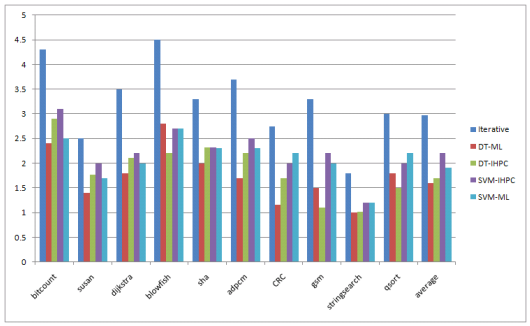

Figure 7. Experimental results using the top 55 spatial based features vs the Milepost features.The vertical axis shows the % improvement in execution time. The horizontal axis shows applications. DT-ML and DT-IHPC show the performance of DT using the Milepost and our framework respectively. SVM-ML and SVM-IHPC show the performance of linear SVMs using the Milepost and our framework respectively.

Figure 8 gives the comparison of our approach using the top 25 spatial features against the Milepost-gcc framework using DT and linear SVMs learning. Again, our approach out performs the Milepost-gcc on most of the applications. The maximum and minimum gain over the -O3 level by our approach using the top 25 features is 2.3% and 1% respectively using the DT learning and 2.75% and 1.2% respectively using the linear SVMs. On average, with only 25 features, our approach is gaining 1.5% and 2.0% improvement using DT and linear SVMs respectively.

We got similar results using non-linear SVM. However, due to the space constraint we are not presenting them here. We also combined the spatial based top 55 features with the Milepost features and gave the combined set of features to DT. We did not find much improvement in the execution time of programs. However, in more than 70% decision trees, spatial based features were the top features. This clearly shows that the features based on spatial information are more informative than the Milepost features for building better machine learning models.

tables

VII. Conclusion and Future Work

In this work, we showed that the features based on the spatial information of data flow graphs have significant

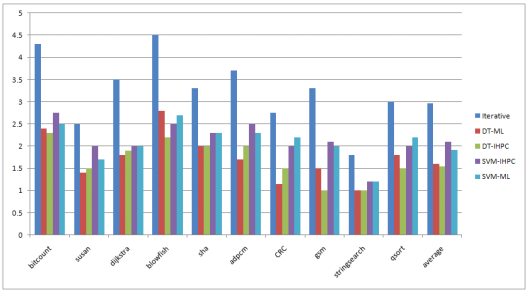

Figure 8. Experimental results using the top 25 spatial based features vs the Milepost features.

potential in the auto-tuning of compiler optimization. We presented in this paper a novel technique to represent this information as features for machine learning tools. We tested our approach extensively using the SPEC2006 and Mibench benchmarks and compared it against the IBM Milepost-gcc framework. The results showed that our framework clearly out-performed the Milepost framework on almost all benchmark applications using DT and SVMs learning.

In future, we plan to use the features based on the spatial information to investigate the automatic selection of better order of optimization passes and fine-grained tuning of transformation parameters for important optimization e.g. unrolling factor of loop unrolling optimization.

REFERENCES

[1] http://gcc.gnu.org/

[2] F. Agakov, E. Bonilla, J. Cavazos, B. Franke. *Using machine learning to focus iterative optimization*. pp. 295-305, 03 2006.

[3] F. Bodin, T. Kisuki, P. M. W. Knijnenburg, M. O'Boyle and E. Rohou. *Iterative compilation in a non-linear optimization space*. Workshop on Profile Directed Feedback-Compilation, PACT'98, October 1998.

[4] *http://www2.cs.uregina.ca/ dbd/cs831/notes/ml/dtrees/c4.5/*

[5] J. Cavazos and M. O'Boyle. *Method-specific dynamic compilation using logistic regression* SIGPLAN , vol. 41, no. 10,pp. 229-240, 2006.

[6] J. Cavazos and J. Moss. *Inducing heuristics to decide whether to schedule*. SIGPLAN , vol. 39, no. 6,pp. 183-194, 2004.

[7] K. Copper and P. Schielke. *Optimizing for reduced code space using genetic algorithms.* Proceedings of the ACM SIGPLAN 1999 workshop on Languages, compilers, and tools for embedded systems, 1999.

[8] C. Dubach, T. M. Jones, E. V. Bonilla, G. Fursin and M. F. O'Boyle. *Portable Compiler optimization across embedded programs and micro- architectures using machine learning*. In Proceedings of the 42^{nd} IEEE/ACM International Symposium on Micro- architecture, 2009.

[9] G. Fursin, C. Miranda, O. Temam , M. Namolaru, E. Yom-Tov, A. Zaks, B. Mendelson, P. Barnard, E. Ashton, E. Courtois, F. Bodin, E. Bonilla, J. Thomson, H. Leather, C. Williams, M. O'Boyle. *MILEPOST GCC: machine learning based research compiler.* In Proceedings of the GCC Developers' Summit, Ottawa, Canada, June 2008

[10] A. McGovern and E. Moss. *Scheduling straight-line code using reinforcement learning and rollouts*. In Proceedings of Neural Information Processing Symposium. MIT Press, 1998.

[11] S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, 2006.

[12] E. Ipek and S. A. Mckee. *Efficently exploring architectural design spaces via predictive modeling*. In Proceedings of Architectural Support for Programming Languages and Operating Systems(ASPLOS) 2006.

[13] H. Leather, E. Bonilla, M. O'Boyle. *Automatic feature generation for machine learning based optimizing compilation*. In Proceeding of Code Generation Optimization 2009.

[14] A. Papadopoulos and Y. Manolopoulos. *Structure based similarity search with graph histograms.* In Proceedings of the 10^{th} International Workshop on Database and Expert Systems Applications 1999.

[15] Z. Pan and R. Eigenmann. *PEAK: a fast and effective performance tuning system via compiler optimization orchestration.* In the ACM Transactions on Programming Languages and Systems (TOPLAS),Article No.: 17 ,Volume 30 , Issue 3, May 2008.

[16] M. Stephenson, S. Amarasinghe, M. Martin, and U.-M.O'Relly. *Meta optimization: Improving compiler heuristics with machine learning*. MIT-06-2003.

[17] *http://svmlight.joachims.org/*