CECS

CENTER FOR EMBEDDED & CYBER-PHYSICAL SYSTEMS

Using Hardware Counters to Predict Vectorization

Neftali Watkinson, Aniket Shivam, Zhi Chen, Alexander Veidenbaum and Alexandru Nicolau

Center for Embedded and Cyber-Physical Systems

University of California, Irvine

Irvine, CA 92697-2620, USA

{watkinso, aniketsh, zhic2, alexv, anicolau}@uci.edu

CECS Technical Report 17-01

May 19, 2017

Using Hardware Counters to Predict Vectorization

Neftali Watkinson, Aniket Shivam, Zhi Chen, Alexander Veidenbaum and Alexandru Nicolau
Department of Computer Science, University of California, Irvine
{watkinso, aniketsh, zhic2, alexv, anicolau}@uci.edu

***Abstract*— Vectorization is the process of transforming the scalar implementation of an algorithm into vector form. This transformation aims to benefit from parallelism through the generation of microprocessor vector instructions. Using abstract models and source level information, compilers can identify opportunities for auto-vectorization. However, compilers do not always predict the runtime effects accurately or completely fail to identify vectorization opportunities. This ultimately results in no performance improvement.**

This paper takes on a new perspective by leveraging the use of runtime hardware counters to predict the potential for loop vectorization. Using supervised machine learning models, we can detect instances where vectorization can be applied (but the compilers fail to) with 80% validation accuracy. We also predict profitability and performance in different architectures.

Our experiments evaluate a wide range of hardware counters across different machine learning models. We show that dynamic features, extracted from performance data, implicitly include useful information about the host machine and runtime program behavior.

Keywords*— *Machine Learning, Compilers, Auto-vectorization, Profitability

I. INTRODUCTION

During the last decades, numerous techniques have been proposed for manually or automatically transforming code to improve performance and optimize the use of resources. When a compiler is performing optimizations, most of the transformations are applied early in the compilation process relying on an abstract model of the host machine configuration. Because of this, it is expected that only a few optimizations will take full advantage of the host's architecture and hardware configuration.

Among the different transformations that the compiler can use, there are some designed to take advantage of different levels of parallelism found in modern architectures. These may range from parallel functional units in a CPU core, to using special instruction sets and parallel thread execution. Such is the case for Single Instruction Multiple Data (SIMD) instructions. They apply an operation on "vectors" of data simultaneously. Hence the transformation of code to take advantage of these instructions is called "vectorization".

Since obstacles like data dependency, function calls, or complex memory access patterns could prevent vectorization, most compilers rely on static information (gathered at the source level and/or at the intermediate representation) to decide when to vectorize. Part of this information is the use of a cost model to predict if vectorization is profitable, i.e. will lead to performance improvement.

However, the compiler's analysis faces challenges as well. The two most common ones are: lack of running-time information (e.g. the behavior depends on the input data) and inaccurate profitability prediction due to an incomplete model of the host architecture. In [10] we can see that three of the widely used (and arguably best) C compilers (IBM XL, Intel's compiler, GCC) failed to vectorize more than 30% of loops inside a benchmark. Extensive manual analysis showed that many of these loops were clearly vectorizable.

With the increasingly complex modern architectures with highly constrained vector instructions (i.e. with a specific purpose for the architecture), multiple levels of parallelism exploitation (e.g. vector instructions, instruction-level parallelism (ILP), multi-core processing), and complex memory hierarchies, it is also increasingly hard for compilers to create an accurate model.

Recent approaches have tried to design a model that is self-adaptable (i.e. can be applied to different machine configurations) and capable of predicting vectorization and its profitability.

The work from Fursin et al. [16] shows that creating models from dynamic and static data can be used to identify parallelism and profitable scheduling policies for loops inside a code. Kennedy et al. [8] use dependence information extracted at the source and intermediate level to optimize a compiler. However, these approaches are not specific to vectorization and use only static information. However, in the work of Cammarota et al. [3] they use hardware counters and unsupervised learning to find similarity among different applications that translate into performance improvement through the application of similar optimizations among clusters. This shows that dynamic information extracted at run time contains a different representation than the one provided by static models.

In this work, we focus on evaluating dynamic data (hardware counters) in supervised machine learning models for the prediction of vectorization. We predict whether a loop-nest can be vectorized, manually or by a compiler, and if vectorization will be profitable. The prediction is done for two major compilers, the Intel C Compiler (ICC) and the GNU Compiler Collection (GCC), using their respective auto-vectorization options.

To fully evaluate the performance of dynamic data, we make use of six machine learning algorithms: Support Vector Machines, Naïve Bayes, K-Nearest Neighbors, Random Forest Trees and Logistic Regression. In our experiments, we could

predict when vectorization is successful with up to 94% accuracy on cross-validation, and profitability with up to 92% accuracy. In the last experiment, we extend our dataset to include loops that can be manually vectorized but the compiler fails to do so. We created a validation set using these loops and we achieved 80% accuracy in predicting whether the loops could be manually vectorized.

The rest of this paper is organized as follows: The section Approach explains the tools and the methodology that we used, the key concepts needed to understand the problem and the solution. In Experiments, we present the tests we performed and the results obtained with six different sets of classification models and discuss the applicability of hardware event features. The section Related work gives a brief overview of other projects and a comparison of their approaches with ours, including a similar approach that used only static information. Finally, in Conclusion we analyze the implication of our results and explain why this approach to the problem has potential to grow further and be used in other areas of compiler optimization as well.

II. Approach

In most of supervised Machine Learning workflows, there is a dataset from which one can extract features, then build a classifying model using these features and one of the available algorithms, and finally validate that model. In this work, our dataset is built from a set of loops that we compiled, executed and profiled to collect runtime data from the hardware event counters. That data is then used to build the feature vector from where we can select the features that are more relevant for the classifier. We evaluate the features using several machine learning algorithms and select the ones that generate better results at validation. Fig. 1 illustrates the workflow.

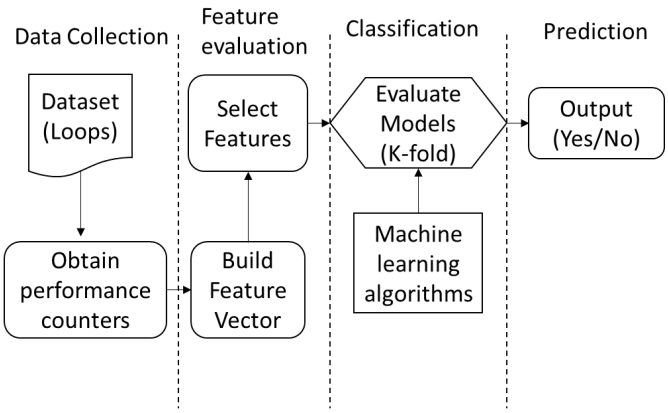

Figure 1 Flow diagram for our Machine Learning Approach.

A. Data Collection

For our training dataset, we used the TSVC (Test Suite for Vectorizing Compilers) benchmark [10]. It contains 151 loops written in C. It was designed to evaluate auto-vectorization capabilities in different compilers. Each loop is also nested inside a control loop that repeats the run to record execution time accurately.

We reevaluated the work in [10] to update the data for the new versions of the GCC (5.6) and ICC (15) compilers. By using –O3, there are 105 loops that are vectorized by ICC and 57 by GCC (114 in the union set of both compilers). However, by disabling the cost estimation model and force the compiler to vectorize (using the -vec-threshold0 flag on ICC and -fvect-cost-model=unlimited on GCC), the report yielded 116 from Intel and 78 from GCC (with 123 total). For our first experiment, we disregard the profitability of vectorization and focus only on whether the loop can or can't be vectorized.

Adding the information about manual vectorization and IBM compiler in [10] we end up with 137 loops that are vectorizable (compared to the 124 originally reported in [10]). Note that for our approach we consider a loop vectorizable if either of the compilers report some level of vector transformations applied to the loop or there was a way to vectorize it manually.

To create a separate validation data for our last experiment, we extracted loops from the following 13 benchmark applications: ALPBench [22], ASC-llnl [21], Cortexsuite [23], Fhourstones [24], FreeBench [25], Kernels [26], Livermore [27], MediaBench [28], Netlib [29], NPB [30], Polybench [31], Scimark2 [32], SPEC [33].

We isolated the loops that are not vectorized by ICC and then performed **manual vectorization** to identify the ones that could be vectorized. There was a total of 490 loops that could not be vectorized by the compilers. Out of those, 123 could be vectorized manually. We profiled these loops using the same method described for TSVC.

B. Hardware Performance Counters

The features that we input into the machine learning model are built using performance data gathered with Linux Perf [20] and Intel Vtune Amplifier XE 2016 [12]. Since we want to predict vectorization only, the data comes from non-vectorized code (with O3 enabled but vectorization turned off). Some of the hardware events that we obtained include branching, micro operations, cycles, memory flow, and time-based events, among others.

C. Feature Evaluation

It is very important when working with a new set of features, to evaluate and discard those that - related to the model's output - are irrelevant or introduce noise to the dataset. This list of features may be different for each ML model and algorithm depending on the problem being solved. Since we use every hardware counter available to build our feature vector, we use filter based feature selection [11].

In this study, we selected features using Information Gain (IG), a ratio useful for identifying those features that reduce bias in decision trees. It builds a decision tree for each feature and measures the difference in entropy before and after splitting the instances. The higher the IG value the better the feature is in dividing the dataset. In section III we discuss the features selected in each experiment, as well as in general terms the ones that we found to be most useful across the experiments.

Since our features are obtained from running the application, and will change with different machine configurations and runtime environments, they are considered as dynamic features. The main difference between dynamic and static features is that static features are obtained at either the source code level or the intermediate representation and thus doesn't require the application to be run [1]. A model that uses static features will most likely have the same result regardless of the running environment, whereas a model with dynamic features has the environment implicitly represented in the performance counters.

D. Classification

There are several machine learning techniques and algorithms for classification. We focused on applying the most common ones. We used Orange 3.2 [7] to build and evaluate the models.

1) Majority Classifier

This feature-agnostic classifier determines the output for every instance to be equal to the class or target value that has most instances in the training set (e.g. if most of the instances in the training data are part of the yes class, then the output will be yes for every other instance). Since this is technically the worse a model can do, it is used to establish a baseline for the other classifiers. For a dataset that has the same number of instances for either class (in binary classification) the baseline is 50% in training accuracy, but since our data is not equally distributed keep comparing our results to this theoretical classifier and select the models that surpass it.

2) Support Vector Machine (SVM)

This is one of the most popular classification algorithms. It creates a virtual space with multiple dimensions (hyperspace) and then represents each instance as a point in that space using features as coordinates. It then divides the space by using a hyperplane via support vectors that define the limits for each class. When a new data point is evaluated, depending on which side of the decision hyperplane it falls, the model will define the predicted class. Along with feature filtering, it is a very powerful classifier for discriminative classification, and it is also very versatile since the user can define the number of dimensions the kernel that the SVM will use.

3) K-nearest Neighbors (KNN)

Similar, to SVM, this algorithm also deals with a vector space in the sense that it represents the instances as points in it using features as coordinates. However, instead of dividing the data into two spaces, it calculates the distance between the data point to be predicted and the ones nearest to it. The predicted value will be equal to the majority of the *k* neighbors nearest to it. The distance can be measured with different metrics, we use Manhattan distance (distance following a strictly horizontal and vertical axis [11]).

4) Naïve Bayes

This is a generative model that makes a statistical assumption: the features are conditionally independent to the predicted value. Once trained, the probability of each class is computed given the new example's feature. It then classifies the example as having the most likely class label [11].

5) Random Decision Forest

This algorithm builds a set of classification trees using the training data and outputs the mode for all the trees built. This is done to countermeasure the tendency of classification trees to overfit. Each tree is built using a subset of the features chosen stochastically [19]. The predicted value comes from the average of the trees selected in training.

6) Logistic Regression

This algorithm assigns weights to each of the features and fits a sigmoid function where all the instances are mapped and assigned a probability of being part of a class or not [11].

E. Validation

We validate most of our models using leave-one-out cross validation, which is a variance of K-fold Cross validation [35]. We use all the instances but one for training, and validate on the remaining one, iterating until all the instances have been used once for validation. No classifier is ever tested on examples used in training it, which maintains the statistical integrity of the procedure. For the last experiment, we have previous knowledge of the features that are good predictors and we include more loops in the dataset so we can create a separate validation set by randomly splitting the data 70/30 (70% for training and 30% for validation). We do leave-one-out cross validation on the training set to select the features that perform the best and then use the validation set for accuracy.

III. Experiments

The first four experiments are designed to identify a relationship between hardware counters (dynamic features) and auto-vectorizability (whether a compiler can or cannot vectorize the given loop). The first and second experiment consider Harpertown and Haswell architecture respectively but each with a different set of hardware counters. The third experiment analyzes the use of hardware counters with a different compiler. The fourth experiment uses the data from [10] to predict vectorization in general (vectorization with any of the compilers and manual vectorization). The fifth experiment analyzes profitability, and compares our results to the Intel Compiler model. The sixth and final experiment tests the models using an extended dataset consisting of non-vectorized loops.

A. Hardware Setup

Since our approach is designed to be independent of the architecture, or more accurately, adaptable to it, we performed our experiments on three different Intel processors. Table I shows the hardware configuration for our experiments, we used Harpertown for the first and sixth experiment, and Sandy Bridge and Haswell systems for the rest.

For every architecture, we compiled and ran the TSVC dataset using ICC 15.0.4 and GCC 5.3.0 with O3 optimization minus vectorization (-fno-tree-vectorize option for GCC and –no-vec for ICC) to obtain the hardware counters without vector transformations. Next, we compiled and ran the dataset again but enabling vectorization and using the default cost model for each compiler (the default flags are "-fvect-cost-model=dynamic" for GCC and "-vec-threshold100" for ICC),

which will vectorize only when the compiler estimates that there is going to be favorable speedup (over 1.0). Finally, we performed the same experiment but now ignoring the cost model ("-fvect-cost-model=unlimited" for GCC and "-vec-threshold0") to obtain runtimes for the loops that the compiler's cost models predict to have no speedup. We consider that a compiler is capable of vectorizing a loop, if for any of the two cost models the compiler report one or more vector transformations.

TABLE I Hardware Configuration of the Three Systems

Hardware Setup			
Microarchitecture	**Harper-town**	**Sandy Bridge**	**Haswell**
Operating System	CentOS Linux 6.6 (x64)	Linux Ubuntu 14.04 (x64)	Linux Ubuntu 14.04 (x64)
Processor	Xeon E5450	Intel Core i7-2600	Intel Core i7-4770
Vector Instruction Set Extensions	SSE 4.1	SSE 4.1/4.2, AVX	SSE 4.1/4.2, AVX 2.0
Processor Frequency	3.00 Ghz (4 cores)	3.40 Ghz (4 cores)	3.40 Ghz (4 cores)
RAM	32 GB	8 GB	8 GB
Compiler version			
Intel C Compiler		ICC 15.0.4	
GNU Compiler Collection		GCC 5.3.0	

B. EXP1: Predicting Vectorization on Harpertown

Our Harpertown architecture has vector instructions up to SSE 4.1, so we expected the evaluation to find the loops that are vectorized by ICC to be different than the other two systems. Due to compatibility issues with Intel Vtune Amplifier, for the Harpertown System we used Linux Perf to obtain the hardware counters. For this experiment, our machine learning models are predicting whether ICC can vectorize the loops with any of the profitability models. In this case the significant features used were: L1 Cache loads misses, LLC store misses, LLC stores, D-TLB load misses, cache-references and branch-misses. Table II shows the results for the top classification models. Of all the models, Random Forest had the best result for accuracy on leave-one-out cross validation.

Notice (Table II) that the majority classifier has 77% accuracy. This is because 77% of the loops in the dataset are vectorizable by ICC, therefore the baseline for the models accuracy is high. However, 83% accuracy on our first experiment is very impressive considering the lack of knowledge there is in the correlation between the features and the output.

TABLE II Accuracy for Xeon Processor (EXP I)

Harpertown	
Algorithm	Overall Accuracy
Majority	0.77
Random Forest	0.83

C. EXP2: Different Architecture

For this experiment, we evaluate the models for predicting vectorization by ICC in the Haswell system. To take full advantage of the hardware counters available, we used Intel Vtune Amplifier's analysis to obtain our data for the feature vector. Due to the availability of AVX instructions, ICC yields different results when vectorizing the code.

The features selected by the filter were instructions retired, Micro-operations issued, Retired load micro-operations, Stalls pending (cycle activity), micro-operations executed, micro-operations dispatched by port, and load misses in D-TLB levels.

TABLE III Accuracy for ICC Auto-Vectorization (EXP 2)

Vectorizability	
Algorithm	Overall Accuracy
Majority	0.77
SVM	0.80
Naïve Bayes	0.73
KNN	0.81

Table III shows that the three best classification models for this dataset are Naïve Bayes, SVM and KNN. Results are similar to our previous experiment in the increase of accuracy over the baseline. While Naïve Bayes has a lower classification accuracy than the majority classifier, we included it because we observed that the minority class (not vectorized) has an increase in true negatives (lower index of false positives). Depending on the model's application, the user may sometimes want to maximize the number of true positives (classes correctly classified as positive), and other times minimizing false positives (classes incorrectly classified as positive). In a separate experiment, Haswell's results were very similar to Sandy Bridge, so for the sake of space we will not discuss them.

D. EXP3: Different Compiler

For this experiment, we turn our focus to the GCC compiler and used its vectorization report to feed the model. For the feature vector, we profiled a GCC's non-vectorized version of the benchmark.

In Table IV, the majority classifier has 0.5 accuracy for both Harpertown and Haswell, which sets a good baseline for comparison. SVM and Naïve Bayes are the best models with up to 75% accuracy. For Haswell, the filter chose Instructions retired, Micro-operations issued, Micro-operations dispatched by port (8 ports total), and Micro-operations retired. For Harpertown, the features selected were CPU Cycles, Cache References, L1 D-Cache Load Misses, L1 D-Cache prefetches, L1 I-Cache loads and bus cycles. The models had very similar

results, with up to 25 percentage point improvement over the baseline. Because we have information about the loops that are vectorized by ICC and manually, we know that the model is predicting that other loops are vectorizable, but GCC would not vectorize them.

Since GCC uses a vectorization model that doesn't change much with the host architecture (vectorization results are the same regardless of the architecture, while ICC produces different results based on the cost estimation), we believe that the loops vectorized by it will have some degree of similarity between them, forming clusters that are easily detectable by the classifier. This also explains the difference in features from the previous experiment, as the models are looking for different patterns, the filter will dim a different subset of features to be useful for the given problem.

While we considered testing with LLVM compiler, the results in the vectorization report (number of loops vectorized) for TSVC were not better than GCC's.

TABLE IV Accuracy for GCC's Auto-Vectorization (EXP 3)

Vectorizable with GCC in Harpertown	
Algorithm	Overall Accuracy
Majority	0.5
SVM	0.75
Naïve Bayes	0.67
Vectorizable with GCC in Haswell	
Algorithm	Overall Accuracy
Majority	0.5
SVM	0.73
Naïve Bayes	0.76

E. EXP4: Testing for Manual Vectorization

As mentioned in section II, TSVC was used in [10] to test vectorization capabilities of different compilers. For this experiment, we used their data to build a separate set of models to predict vectorizability by either GCC, ICC, IBM XL or manual vectorization. The output class is whether a loop can or can't be vectorized by any of the mentioned compilers, and/or manually. We used the feature data from the ICC non-vectorized version ran on the Haswell system. Table V shows the result from this experiment.

TABLE V Accuracy for All Auto-Vectorization (EXP 4)

Vectorizability	
Algorithm	Overall Accuracy
Majority	0.92
SVM	0.94
Naïve Bayes	0.89
Logistic Regression	0.91

Just like the previous experiments, Naïve Bayes tends to sacrifice overall accuracy but with lower false positives than other classifiers. SVM obtained the highest overall accuracy with 94%. A 92% for majority class is a high value, and there may not be enough instances of the negative class (non-vectorized) for the model to detect a pattern or cluster. However, the results seem to be consistent with the other experiments. The features selected were: Memory Micro-operations retired, Cycle stalls, Micro-operations dispatched by port, and L1 D-Cache replacements.

F. EXP5: Testing for profitability

Since on Haswell and Sandy Bridge systems, ICC uses a profitability model to decide when to vectorize a loop, we designed this experiment to analyze the performance of hardware counters to predict the profitability of vectorization. The main challenge for this model is that profitability will change per the architecture.

We collected data by running a vectorized version and comparing the runtime for each loop. We removed the loops that cannot be vectorized by ICC, and then defined every loop with at least 1x speedup as profitable, which is the same threshold the compiler uses.

In these models, the significant features selected for Haswell were Instructions Retired, Micro-operations retired, Micro-operations dispatched by port, and Memory Micro-operations retired. For the Sandy Bridge model, the features selected where Instructions Retired, Micro-operations issued, Micro-operations retired, Memory Load Micro-operations, L2 Cache hits, and IDQ Micro-operations not delivered.

Table VI shows the results for the experiment. Interestingly, Sandy Bridge's models have highest accuracy when predicting profitability than Haswell's models do.

TABLE VI Accuracy for Profitability Classification (EXP 5)

Profitability	
Haswell	
Algorithm	Overall Accuracy
Majority	0.84
SVM	0.83
Naïve Bayes	0.88
KNN	0.85
Logistic Regression	0.85
Intel Compiler	0.86
Sandy Bridge	
Algorithm	Overall Accuracy
Majority	0.90
SVM	0.93
Naïve Bayes	0.92
KNN	0.91
Logistic Regression	0.91
Intel Compiler	0.91

While in previous models, comparing to the compiler was somewhat impossible (the compiler vectorizes what it can without making any prediction), we can compare to the compiler's Cost Estimation Model. We don't know how this

model is computed, but we can get the output value by using the -vec-report=9 flag which lets us calculate an accuracy for their model. Note that the compiler uses information obtained very early in the compilation process. Through informal experiments we identified that it will predict different values depending on the architecture, which leads us to think that the model includes some abstract representation of the host system.

The compiler correctly predicts profitability in 86% of the loops ran on Haswell and 91% on Sandy Bridge. Note that we only consider profitability, however the value obtained from ICC's cost model is not always close to the actual speedup. It is worth noticing that with the newer architecture (Haswell), a lower number of loops have profitable vectorization and the compiler has lower accuracy in predicting it. This could be because the cost model is not keeping up with the changes of architecture, or the newer architectures have other optimizations that make vectorization less critical, depending on the loop.

Our best models are 2% more accurate than Intel's Model and are completely agnostic to the information the compiler uses to estimate speedup. These results may be used in future work to fine tune the compiler's model.

G. *EXP6: Finding opportunities for Vectorization with a Validation Set*

TSVC is biased towards already vectorizable loops (most the loops are vectorized by ICC). Therefore, for this experiment we combined the dataset with the loops from 13 benchmarks (as mentioned in section II) that ICC couldn't vectorize, but some of them could be manually vectorized. We partitioned the data so that 70% of this extended dataset would be used for training, and the rest of the loops could be used as a validation set. We first performed cross validation on the training set to select the best features using an Information Gain filter. In this case the features chosen were LLC loads, LLC stores, D-TLB load misses, Branch loads, L1 Cache prefetches and Instructions per cycle. We then tested our model using the validation dataset.

TABLE VII Accuracy for Validation Set (EXP 6)

Cross Validation for 70%	
Algorithm	Overall Accuracy
Majority	0.62
SVM	0.72
Naïve Bayes	0.70
KNN	0.71
Random Forest	0.78
Validating on 30%	
Algorithm	Overall Accuracy
Majority	0.61
SVM	0.75
Naïve Bayes	0.70
KNN	0.71
Random Forest	0.80

Table VII shows that the improvement over the majority classifier is much greater. This gives us a better sense of the potential that hardware counters have in detecting vectorization. When testing on the validation set, Random Forest performs best with 80% accuracy in detecting loops with opportunities for vectorization - recall that these loops were not originally vectorized by the compiler. This prediction could be used to detect loops that require a deeper analysis for improvement.

H. *Result Analysis*

The results on vectorizability from the first four experiments give us an insight on the effect vectorization has on the loop nests. Table VIII shows the different groups of features and how they were used in each of the 6 experiments. While Vtune and Perf don't have the same hardware events, we can group them and correlate them by the type of hardware counter.

TABLE VIII Features used by experiment

Features used by Experiment						
Feature class	Exp1	Exp2	Exp3	Exp4	Exp5	Exp6
Instructions related counters	✓	✓	✓	✓	✓	✓
Last Level Cache (LLC) counters	✓		✓			✓
L1 Data Cache counters	✓	✓	✓			✓
Others	✓	✓	✓	✓	✓	✓

The experiments show evidence that there is information about the opportunities for vectorization hidden in the runtime data. In this experiment, all the features were extracted from non-vectorized implementations of the code. While some experiments are designed to show that it is possible to predict vectorization using dynamic features, the output of those models is easily obtained by running the compiler. However, for our last experiment the output is not so trivial. By using the validation set, we are finding loops that the compiler is not able to vectorize but with manual analysis could be vectorized. This can be applied to detect optimization opportunities where today's tools fail.

Going back to the dynamic features, there were some that may seem counterintuitive but they still yielded good results in the classification models. Possible explanations behind some features being prominent across the different models are:

• **IPC (Instruction per Cycle):** The loops with higher IPC values seem to be well-suited for vectorization. The higher IPC correlates to higher parallelism at instruction level, to computation-bound loops and low cache miss rates, and predictable branches. The high scalar IPC also indicates that

multiple operations available in same or near cycles, which indicates a potential to vectorize.

• **Branch Instructions:** Number of branch instructions had significant impact on predicting vectorization. Branching leads to inefficiency in generating vector instructions since even after several compiler optimizations (Flattening-IFs, Index Set Splitting, etc.) it may be not possible to eliminate branches. Also, branch misprediction introduce stalls and reduce IPC.

• **L1 Data Cache:** Misses in L1 D-Cache may indicate that access to data inside the loop is not consecutive across iterations and hence vector instructions may not profitable. The fetching or storing of each data element needs to be performed as a scalar access and moved to/from an SIMD register separately. While the computation is now a vector instruction, but scalar moves are costly.

• **LLC Loads and Stores:** The LLC accesses are costly and LLC misses even more so. Both can significantly lower the IPC, ultimately making the program memory bound. SIMD instructions may not give speedup when the memory hierarchy dominates performance.

• **TLB:** A TLB miss also introduces significant delays and has the same effect on IPC as cache misses.

The results from our last experiment demonstrate that the accuracy of the models is not random. Being able to predict vectorization in a "blind" validation test, implies that the correlation between runtime data and vectorization potential is strong. The static information that compilers use to vectorize the code is insufficient.

IV. Related Work

The use of static models and dependence analysis for vectorization has been studied for at least three decades ([2] and [8]). However, the use of machine learning for compiler optimization is new and growing. In [3] a feature-agnostic model is used to predict performance, it deals mainly with finding parallelism and it doesn't need any previous knowledge of the program because it uses unsupervised learning. [1] uses a set of hand written rules guided by a combined set of static and dynamic features to generate suggestions for the programmer on where to apply transformations that will help to vectorize code using SIMD instructions. Their decision system is designed to work at the source level and it only deals with possible vectorization opportunities that need to be validated by the user. They detect 123 vectorizable loops in the TSVC benchmark. Their focus is on the GCC compiler and profitability is not discussed. Our experiments show that only 105 loop-nests had profitable vectorization for Sandy Bridge (98 for Haswell).

The work presented in [13] applies an SVM model to detect basic block vectorization specific to unroll factors (they populate a dataset using TSVC with different unroll factors from 0 to 20, ending with a dataset of 151 x 20 loops) and get a final classification accuracy of about 70% in binary (yes/no) classification to determine whether unrolling would be profitable or not. They use static features only.

The justification behind using static features is that the application doesn't need to be run. However, in [17] they make a good case about using profiling information by showing that it improves the classification model considerably. In their specific case, it is applied towards identifying auto-parallelization, which represents a different challenge than auto-vectorization, the latter being a more constrained problem. They use an SVM predictor to identify if a parallel execution would be profitable and identify which scheduling policy to choose and rely on the user to approve the cases where static correctness cannot be proved. The dynamic features they rely on (aside from the static ones) are Data Access count, Instruction count, and Branch count.

Considering the success Machine Learning has had through the years in other areas such as text mining, image recognition, among others; there is still much work to be done. Our approach stands out from the others mentioned because it evaluates a wide array of dynamic features and machine learning models, and compares it with the state of the art which is what the compiler uses today. In addition, we analyze our results against the baseline, i.e., Majority Classifier to strengthen the validity of our results, something that is not considered in any of the prior works. The reason behind dynamic features yielding better results than static ones may be because they implicitly include information about the host's architecture and the effect that other optimizations have. The cost estimation models used by compilers are applied very early in the compilation process so they disregard the effects of other optimization transformations and the differences in architectures, therefore the use of only static features will likely have the same outcome. However, static features are not to be discarded, in future work we are going to explore dependence analysis information to our models.

V. Conclusion

In this paper, we evaluated the use of hardware counters in machine learning to predict if vectorization can be applied and how profitable it will be. We predicted loops that contain opportunities for manual vectorization that the compilers miss with 80% accuracy. As of now, this model can already be applied to find potential candidates for vectorization, without having to manually analyze each loop that the compiler can't vectorize.

In other experiments, we built different models that predict the vectorizability of a loop-nest when using GCC (with 76% accuracy), ICC (80% accuracy), and manual vectorization (94% accuracy), as well as vectorization profitability (93% accuracy). This shows viability towards the use of runtime data to identify optimization opportunities. This gives new insight into code optimization. We can identify optimization opportunities that a commercial compiler misses.

Dynamic information can be used to further optimize the compilers. Since our approach is not constrained to a specific compiler or architecture, it can be further implemented to predict how the combination of the two will produce different gains in performance.

REFERENCES

[1] Aumage, O., Barthou, D., Haine, C., & Meunier, T. (2014). Detecting SIMDization Opportunities through Static/Dynamic Dependence Analysis. In Euro-Par 2013: Parallel Processing Workshops (Vol. 8374, pp. 637–646).

[2] Banerjee, U. (1988). An introduction to a formal theory of dependence analysis. The Journal of Supercomputing, 2(2), 133–149.

[3] Cammarota, R., Beni, L. A., Nicolau, A., & Veidenbaum, A. V. (2013). Optimizing program performance via similarity, using a feature-agnostic approach. In Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics) (Vol. 8299 LNCS, pp. 199–213).

[4] Castro, P. D. O., Akel, C., Petit, E., Popov, M., & Jalby, W. (2015). CERE: LLVM-Based Codelet Extractor and REplayer for Piecewise Benchmarking and Optimization. ACM Transactions on Architecture and Code Optimization, 12(1), 1–24.

[5] Chandy, K. M., & Kesselman, C. (2005). Compositional C++: Compositional parallel programming. In Languages and Compilers for Parallel Computing (Vol. 757, pp. 124–144).

[6] Corporation, I. (2012). A Guide to Vectorization with Intel ® C ++ Compilers. Intel Corporation.

[7] Demšar, J., Curk, T., Erjavec, A., Gorup, Č., Hočevar, T., Milutinovič, M., Zupan, B. (2013). Orange: data mining toolbox in python. The Journal of Machine Learning Research, 14(1), 2349–2353.

[8] Kennedy, K., & Allen, J. R. (2001). Optimizing compilers for modern architectures: a dependence-based approach.

[9] Kulkarni, S., & Cavazos, J. (2012). Mitigating the compiler optimization phase-ordering problem using machine learning. Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications - OOPSLA '12, 147.

[10] Maleki, S., Gao, Y., Garzarán, M. J., Wong, T., & Padua, D. A. (2011). An evaluation of vectorizing compilers. In Parallel Architectures and Compilation Techniques - Conference Proceedings, PACT (pp. 372–382).

[11] Manning, C. D., Raghavan, P. & Schütze, H. (2008). Introduction to Information Retrieval. In Americas (Vol. 32, pp. 10013–2473). Delhi Cambridge University Press.

[12] Reinders, J. (2005). VTuneTM Performance Analyzer Essentials VTuneTM Performance Analyzer Essentials Measurement and Tuning Techniques for Software Developers (First.). Intel Press.

[13] Trouvé, A., Cruz, A., Fukuyama, H., Maki, J., Clarke, H., Murakami, K., Yamanaka, E. (2013). Using Machine Learning in Order to Improve Automatic SIMD Instruction Generation. Procedia Computer Science, 18, 1292–1301.

[14] Trouvé, A., Cruz, A. J., Brahim, D. BEN, Fukuyama, H., Murakami, K. J., Clarke, H., … Yamanaka, E. (2014). Predicting Vectorization Profitability Using Binary Classification. IEICE TRANSACTIONS on Information and Systems, E97-D (12), 3124–3132.

[15] Tape, T. G. (2006). Interpreting diagnostic tests. University of Nebraska Medical Center, http://gim.unmc.edu/dxtests.

[16] Fursin, G., Kashnikov, Y., Memon, A. W., Chamski, Z., Temam, O., Namolaru, M., O'Boyle, M. (2011). Milepost GCC: Machine learning enabled self-tuning compiler. International Journal of Parallel Programming, 39(3), 296–327.

[17] Tournavitis, G., Wang, Z., Franke, B., & O'Boyle, M. F. M. (2009). Towards a holistic approach to auto-parallelization: integrating profile-driven parallelism detection and machine-learning based mapping. ACM Sigplan Notices, 177–187.

[18] Hosmer Jr, D. W., Lemeshow, S., & Sturdivant, R. X. (2013). Applied logistic regression (Vol. 398). John Wiley & Sons.

[19] Breiman, L. (2001). Random forests. Machine learning, 45(1), 5-32. Chicago

[20] Weaver, V. M. (2013, April). Linux perf_event features and overhead. In The 2nd International Workshop on Performance Analysis of Workload Optimized Systems, FastPath (p. 80).

[21] ASC Benchmarks. https://asc.llnl.gov/sequoia/benchmarks/.

[22] Li, M. L., Sasanka, R., Adve, S. V., Chen, Y. K., & Debes, E. (2005, October). The ALPBench benchmark suite for complex multimedia applications. In IEEE, International. 2005 Proceedings of the IEEE Workload Characterization Symposium, 2005. (pp. 34-45). IEEE.

[23] Thomas, S., Gohkale, C., Tanuwidjaja, E., Chong, T., Lau, D., Garcia, S., & Taylor, M. B. (2014, October). CortexSuite: A synthetic brain benchmark suite. In Workload Characterization (IISWC), 2014 IEEE International Symposium on (pp. 76-79). IEEE.

[24] Fhourstones benchmarks. https://github.com/llvm-mirror/test-suite/tree/master/MultiSource/Benchmarks.

[25] Rundberg, P., & Warg, F. (2002). The FreeBench v1. 0 Benchmark Suite. URL: http://www.freebench.org.

[26] Van der Wijngaart, R. F., & Mattson, T. G. (2014, September). The Parallel Research Kernels. In HPEC (pp. 1-6).

[27] Livermoore loops. http://www.roylongbottom.org.uk/livermore loops results.htm.

[28] Media Bench II http://mathstat.slu.edu/~fritts/mediabench.

[29] Browne, S., Dongarra, J., Grosse, E., & Rowan, T. (1995). The Netlib mathematical software repository. D-Lib Magazine, Sep.

[30] Bailey, D. H., Dagum, L., Barszcz, E., & Simon, H. D. (1992, December). NAS parallel benchmark results. In Proceedings of the 1992 ACM/IEEE conference on Supercomputing (pp. 386-393). IEEE Computer Society Press.

[31] Pouchet, L. N. (2012). Polybench: The polyhedral benchmark suite. URL: http://www. cs. ucla. edu/pouchet/software/polybench.

[32] SciMark 2.0. http://math.nist.gov/scimark2.

[33] SPEC benchmarks. http://www.spec.org

[34] Kohavi, R. (1995, August). A study of cross-validation and bootstrap for accuracy estimation and model selection. In Ijcai (Vol. 14, No. 2, pp. 1137-1145)