IEICE TRANS. INF. & SYST., VOL.E97–D, NO.12 DECEMBER 2014

PAPER

Predicting Vectorization Profitability Using Binary Classification

Antoine TROUVÉ[†a)], **Arnaldo J. CRUZ**[††], **Dhouha BEN BRAHIM**[†††], **Hiroki FUKUYAMA**[††], ***Nonmembers***, **Kazuaki J. MURAKAMI**[††], ***Member***, **Hadrien CLARKE**[††], **Masaki ARAI**[††††], **Tadashi NAKAHIRA**[††††], ***and*** **Eiji YAMANAKA**[†††††], ***Nonmembers***

SUMMARY Basic block vectorization consists in realizing instruction-level parallelism inside basic blocks in order to generate SIMD instructions and thus speedup data processing. It is however problematic, because the vectorized program may actually be slower than the original one. Therefore, it would be useful to predict beforehand whether or not vectorization will actually produce any speedup. This paper proposes to do so by expressing vectorization profitability as a classification problem, and by predicting it using a machine learning technique called support vector machine (SVM). It considers three compilers (icc, gcc and llvm), and a benchmark suite made of 151 loops, unrolled with factors ranging from 1 to 20. The paper further proposes a technique that combines the results of two SVMs to reach 99% of accuracy for all three compilers. Moreover, by correctly predicting unprofitable vectorizations, the technique presented in this paper provides speedups of up to 2.16 times, 2.47 times and 3.83 times for icc, gcc and LLVM, respectively (9%, 18% and 56% on average). It also lowers to less than 1% the probability of the compiler generating a slower program with vectorization turned on (from more than 25% for the compilers alone).

key words: *machine learning, support vector machine, automatic vectorization, software characteristics*

1. Introduction

Single Instruction Multiple Data (SIMD) is an effective paradigm which can dramatically raise the peak performance of processors as well as their power efficiency. This is particularly important in high performance computing and embedded systems, in which higher performance is required inside a constant power envelop. However, it is also a double-edged technique and in some cases it is preferable to rather use traditional scalar instructions for performance reasons. This happens because SIMD instructions in modern processors often involve some overheads like data packing and unpacking or very slow access to unaligned data in memory subsystems. Users often rely on the compiler in order to automatically generate SIMD instructions. We call this process *automatic vectorization* and the compilers that perform this type of optimization are called *vectorizing compilers*. Such compilers use heuristics in the middle-end and the back-end in order to create vector instructions from scalar programs. In the case where the so-vectorized program would be actually slower, that is, if vectorization would not be profitable, the compiler should discard vectors and output the original scalar program. This last step is important; yet, modern compilers are not effective at doing so: according to our results detailed in Sect. 4.3, the vectorized program generated by Intel Compiler (icc), the GNU C Compiler (gcc) and LLVM is slower than the scalar one for 56%, 76% and 78% of our benchmark (26%, 26% and 75% if we consider a 5% margin, as explained in Sect. 5.3). In other words, compilers often fail to detect the situations where vectorization is not profitable. The reason is that the complexity of both compilers and architectures has reached a point at which it is very difficult to predict their behavior and to develop reliable optimization heuristics.

In this paper, we decide to acknowledge this situation and we regard the compiler and the target processor as a single black box. We merely consider its inputs and outputs, and try to determine if we could guess some properties from them. This corresponds to the problem of pattern recognition, very common on various domains like image recognition. As detailed by Chirsotpher M. Bisoph [11], the common way to solve such problem is to use machine learning: this has motivated our work.

This paper is organized as follows. First, we introduce the related work in Sect. 2 before explaining how we leverage machine learning in Sect. 3. Next, we explain our experimental setup and propose several experiments to predict vectorization profitability with high accuracy in Sects. 4 and 5. Then we assess in Sect. 6.1 the potential of our method for speeding up the programs generated by the compilers. We conclude by discussing the limitations of our method, and how it could be applied to real-world situations.

2. Related Work

Using machine learning to improve compilers is no breakthrough, and the literature already contains several works that apply support vector machine (SVM) [2], [13], nearest neighbor (NN) [12], [13], artificial neural networks (ANN) [1], [3], and logistic regression [9]. They however make different types of predictions: Stephenson et al [13],

Manuscript received June 10, 2014.
Manuscript publicized August 27, 2014.
†The author is with the Institute of Systems, Information Technologies and Nanotechnologies, Fukuoka-shi, 814–0001 Japan.
††The authors are with the Engineering Department, Kyushu University, Fukuoka-shi, 819–0395 Japan.
†††The author is with the Computing Department, ENSEIRB-MATMECA, Bordeaux, France.
††††The authors are with Fujitsu Laboratories Limited, Kawasaki-shi, 211–8688 Japan.
†††††The author is with Fujitsu Limited, Tokyo, 105–7123 Japan.
a) E-mail: trouve@isit.or.jp
DOI: 10.1587/transinf.2014EDP7190

Park et al. [2] and Agakov et al. [12] determine parameters of code optimizations; Kulkarni et al. [1] order optimization passes in the middle end; Pekhimenko et al. [9] use machine learning to focus search algorithms.

The work from Stephenson et al. [13] uses multiclass classification† to determine for each program the unroll factor that yields the best performance. Their benchmark consists of 2500 loops extracted from several well-known benchmarks targeted at high-performance computing as well as embedded computing. They leverage both SVM and NN, and manage to correctly predict the best unroll factor with an accuracy of 65% and 62%, respectively. This proves to be far better than their baseline, Open Research Compiler††. Our work is however orthogonal to this work for several reasons. First, we consider a very different prediction target: Stephenson et al. predict the unroll factor, and we predict the vectorization profitability. Second, our benchmark programs are different. Third, the set of software characteristics we consider is very different from theirs (their set is similar to the one of Fursin et al. [6], evaluated under the label *Milepost* in Sect. 5.5).

Kulkarni et al. [1] use machine learning to tackle the complex problem of the ordering of optimization techniques. They model an optimization scenario using a Markov process; then they construct optimization scenarios iteratively, one optimization technique at a time, using ANN at each step. They apply their method to the just-in-time compiler of a Java virtual machine, and achieve to reduce the execution time of compiled programs by up to 20%.

Our work, as well as all the previously detailed ones, endeavors to improve the quality of the output of the compiler by reducing the execution time of the compiled programs. From this point of view, the works from Agakov et al. [12] and Pekhimenko et al. [9] are original. They utilize machine learning in order to reduce the compilation time, that is, the execution time of the compiler itself. The objective of Agakov et al. is to reduce the time required to find the best optimization sequence to apply to a given program. They adopt a technique based on NN to bias an existing search algorithm (random or genetic) and they manage to reduce the search time by one order of magnitude. On the other hand, Pekhimenko et al. [9] use logistic regression to determine the parameters of optimization techniques inside a fixed optimization scenario, with the aim of leveraging the fast execution time of logistic regression compared to the heuristics implemented into a commercial vendor compiler. They manage to reduce the compilation time by two orders of magnitude while at the same time slightly improving the execution time. The originality of our work lies on its target: we aim at predicting the profitability of vectorization. We choose to express vectorization profitability as a classification problem instead of a regression problem in the other works. This is the first work to do so to best of our knowledge. It is therefore complementary to the related work, and can be use side-by-side in order to improve traditional compilers that do not use any machine learning. Stock et al. [3] also target the improvement of automatic vectorization, but in the classical context of regression and automatic tuning. They extract their own set of static software characteristics (SSC) from assembly in order to predict performance using various machine learning techniques, and choose the best optimization scenario as the one with the highest predicted performance, the same way as Park et al. [2]. Their technique shows high accuracy on a simple benchmark kernels made of perfectly nested, independent loops that contain one statement; it however performs only slightly better than random search on stencil kernels. Our technique reaches high accuracy for a benchmark suite made of 151 realistic, complex loops.

Similar to our work, the motivation behind the GCC Milepost project by Fursin et al. [6] is to add machine learning capabilities to mainstream compilers so that they can automatically chose and tune optimization sequences for heterogeneous reconfigurable processors. Our work is more specific, and we only focus on automatic basic-block vectorization (ABBV). Yet, the set of static software characteristics (SSC) used by GCC Milepost proved not to be relevant for our benchmark. Indeed, we detail in Sect. 5.5 that for SVM, not only it yields far lower accuracy than our own set of SSC (detailed in Sect. 3.2), but also that this accuracy is similar to the one obtained with random numbers as SSC.

There are several ways to classify software features. First, they may be measured statically from sources or dynamically at runtime. While the second makes it possible to gather far more information, they require to actually execute the program: this is not something we can afford inside a compiler for obvious time-related constraints [10]. Software features may further be hardware dependent and hardware independent. The former suffers from lack of portability: we may rely on some hardware counters on a given machine that are not available on another one, for example because of differences in micro-architecture. For this reason we favor the latter, and our results in Sect. 5 show that this is enough. Another approach is proposed by Park et al. [2]: they leverage graph mining techniques to directly feed the program's dataflow graph to a SVM, and predict the best optimization scenario. They compare this approach with Milepost GCC and yield better prediction accuracy. Still, in this paper, we adopt a more conservative approach, and consider traditional *static*, *hardware independent* software features.

3. Machine Learning

In this section, we detail the inputs and outputs of our predictive machine learning device. We use a common technique called support vector machine (SVM), described in most reference books such as the one by Christopher M. Bishop [11].

†As opposed to binary classification.
††Now called Open64. Online: http://www.open64.net/

Table 1 The software characteristics.

Identifier	Level	Range	Description
AST1	AST	$\mathbb{N}$	The increment of the innermost `for` loop
AST2	AST	{0, 1}	Will the address of the first access to the array be always aligned with the machine's vector size ?
AST3	AST	{0, 1}	In array accesses, is the induction variables involved in the last dimension the one of the innermost loop ?
AST4	AST	$\mathbb{N}$	Number of array accesses in the body of the loop
AST5	AST	$\mathbb{N}$	The number of arrays accessed inside the loop
AST6	AST	{0, 1}	Does the benchmark involve any restrict keyword ?
IR1	IR	$\mathbb{N}$	The size of the dataset
IR2	IR	$\mathbb{N}$	Estimation of the dynamic instruction count
IR3	IR	$\mathbb{N}$	The depth of the innermost loop
IR4	IR	$\mathbb{N}$	The estimated trip-count of the innermost loop
IR5	IR	$\mathbb{N}$	Number of IR statements in the innermost loop
IR6	IR	$\mathbb{N}$	The number of variables used in the innermost loop, for the code in single static assignment (SSA) representation

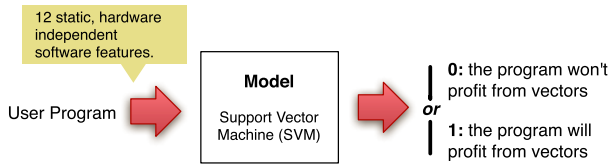

Fig. 1 The inputs and outputs of our model, implemented using a SVM.

3.1 Modeling

We express vectorization profitability prediction as a classification problem: we define a class labelled 1 for programs that profit from vectorization, and another class labelled 0 otherwise. Then we train a binary classifier for predicting the class of new programs. We use SVM as binary classifier. The inputs and outputs of our SVM are shown in Fig. 1. It takes as input the user program, and outputs its class, 0 or 1. The user program is expressed as a vector of floating point numbers called the software characteristics. They are described in the next section.

3.2 The Software Characteristics

Static software characteristics (SSC) consist in numbers that express some important characteristics of the input programs, and measured without executing it. We use the SSC as input features of our SVM in order to predict vectorization profitability. We use 12 software features; 6 of which are extracted at abstract-syntax-tree (AST) level using Clang, and 6 at LLVM's intermediate-representation (IR) level. We have selected them empirically, according to our observations of what may be important when considering basic block vectorization. We use custom SSC instead of GCC Milepost for the reasons explained in Sect. 5.5. Our SSC are detailed in Table 1. When more than one array is accessed in the innermost loop, AST2 and AST3 are estimated for each arrays access, then we consider as a software characteristics their arithmetic mean (a real number between 0 and 1). IR2 and IR4 rely on the prediction of the dynamic behavior of our program provided by LLVM. This is convenient as it uses some placeholder when the values can not be computed. IR6 should be understood as a rough estimation of the number of registers consumed by our loop. Finally, in order to determine AST1, we not only analyze the *for* statement, but also the array indexes for a consistent, constant multiplier of the induction variable.

4. Experimental Setup

4.1 Benchmark and Scope

We use a benchmark called TSVC, which stands for test suite for vectorizing compilers, in its version provided by Maleki et al. [4]. It consists of 151 simple computation loops, initially devised to assess how smart compilers are at vectorizing loops. This benchmark is representative of the portions of code that constitute hotspots of programs, namely the target of optimizing compilation.

We concentrate on one automatic vectorization technique called automatic-basic-block vectorization, referred to as ABBV in the rest of this paper. ABBV consists in leveraging the inherent instruction-level parallelism (ILP) inside basic blocks in order to generate SIMD instructions. This is different from loop vectorization and loop pipelining, which generate SIMD instructions across different iteration of the same basic block[†], not considered in this paper [14]. ABBV is carried out in the compiler's backend and mainly relies on pattern matching. The quality of the results greatly depends on the input to the backend, that is, basic blocks should exhibit enough ILP. The amount of ILP is not only an inherent property of the compiled program but also it is greatly affected by the front and middle ends of the compiler, for instance when unrolling loops.

For the forthcoming experiments, our test machine is an Intel Core2 Duo Merom Processor at 2.66GHz. We consider the following 3 compilers: Intel Compiler (icc) version 12.1.5, GNU C Compiler (gcc) version 4.6.3, and LLVM version 3.3. The options of each with and without ABBV are shown in Table 2.

4.2 Preparation of the Data and Experimental Flow

Our experimental flow is divided into two steps: the generation of known data and the training/validation of the SVM. The flow to generate the former consists of three steps: (1) we unroll, compile and execute each TSVC kernel with and without ABBV; (2) we determine the profitability of vectorization of each unrolled TSVC kernel; (3) we extract the

[†]The reader should further notice that automatic vectorization is equivalent to loop unrolling followed by ABBV.

Table 2 Compiler options that we use for our experiments.

	ABBV	Options
icc	With	-std=c99 -O2 -fno-alias -vec -xSSE3
icc	Without	-std=c99 -O2 -fno-alias -no-vec
gcc	With	-std=c99 -O2 -fivopts -funsafe-math-optimizations -fno-unroll-loops -flax-vector-conversions -msse3 -fno-tree-vectorize -fno-modulo-sched -ftree-slp-vectorize
gcc	Without	-std=c99 -O2 -fivopts -funsafe-math-optimizations -fno-unroll-loops -fno-tree-vectorize -fno-modulo-sched -fno-tree-slp-vectorize
LLVM	With	-std=c99 -O2 -fivopts -funsafe-math-optimizations -fno-unroll-loops -flax-vector-conversions -msse3 -fno-tree-vectorize -fno-modulo-sched -ftree-slp-vectorize
LLVM	Without	-std=c99 -O2 -fivopts -funsafe-math-optimizations -fno-unroll-loops -fno-tree-vectorize -fno-modulo-sched -fno-tree-slp-vectorize

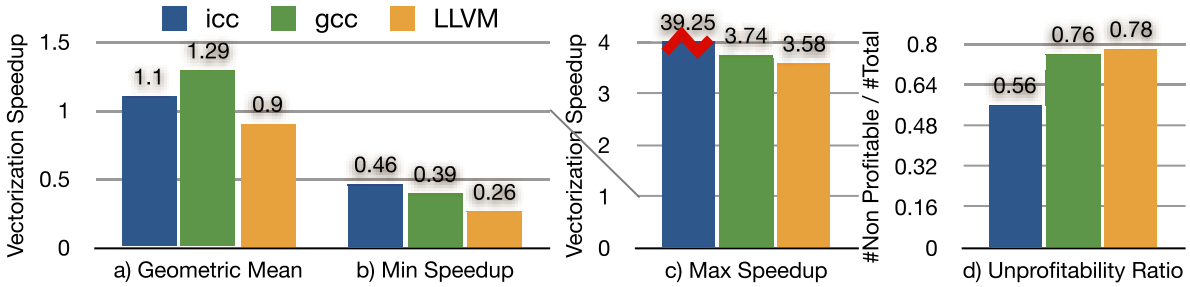

Fig. 2 Overview of the performances of the compilers.

software characteristics from the C source of each unrolled TSVC kernel. The flow relies on three tools: a innermost loop unroller at C-source level, a software-characteristics extractor, and the vendor compiler. For the first, we use a tool called PIPS[†] to unroll the TSVC benchmarks from factors 1 to 20 with the purpose of having enough training data. For the third, we use the icc, gcc, and LLVM compilers. For the second, we measure the software characteristics by means of a custom tool based on the LLVM framework, as explained in Sect. 3.2. In Sect. 5.5, we rather use a random-number generator and the tool GCC Milepost proposed by Fursin et al. [6]. After preparing the data, we assess the quality of our method by computing its LOOCV accuracy, where LOOCV stands for leave-one-out-cross-validation. It consists in training the SVM with all the data but one, predicting for the single left out data, and reiterating the process for all the data in the data set. Our LOOCV procedure is detailed in Sect. 5.1. In order to implement these experiments, we use libsvm[††], a stable and free library for SVM.

4.3 Early Results

We measure the data in the experimental conditions detailed in the two previous sections. First of all, we compute the geometric mean of the distribution of the vectorization speedups for all three compilers in Fig. 2 a. These numbers vary significantly among compilers. In the case of LLVM the mean is below 1, that is, the LLVM vectorizer tends to actually slow down programs in average; fortunately, our method corrects this bias. Figure 2 *b* and *c* further plots the minimum and maximum values of the vectorization speedups for each compiler. The numbers are similar among compilers, only the maximum vectorization for icc is very high: almost 40 times. More importantly, we notice that the minimum vectorization speedup is significantly smaller than 1 for the three compilers; in other words, all the compilers may actually generate programs slower with SIMD instructions than without. To better illustrate this phenomena, we displayed the probability for each compilers to slow down programs in Fig. 2 *d*. The ratios are all above 50%; gcc and LLVM even exceed 75%, that is, these two compilers generate unprofitable vectorization in almost 80% of the situations. This situation is obviously not tenable; we suggest compiler users to never use automatic vectorization for these compilers. Fortunately, our method introduced in next section reduces these probabilities to less than 1%.

5. Our Model

5.1 The Naive Model

Our data set consists of 151 TSVC kernels × 20 unroll factor, that is, 3020 lines. Each line consists of the tuple $\{pid, SSC, vect.\ prof.\}$, where *pid* is an integer between 1 and 151 that uniquely identifies the TSVC kernel, SSC the software characteristics described in Sect. 3.2, $vect.\ prof.$ the integer identifier of the class we are trying to predict (0 or 1). This last entry is ignored during predictions; it is only useful for training as well as calculating accuracy and recall figures. Our LOOCV procedure is shown in pseudo-code in algorithm 1. It takes into input not only the whole data set, but also the vector of all the program ids in our data set, (the sequence of integers from 1 to 151 for TSVC). This vector is used at line 3 to make sure that data lines from a same program do not span across the training and test sets. This is to prevent the presence of oracle in the training set, so that

[†]Online: http://pips4u.org/
[††]Online: http://www.csie.ntu.edu.tw/˜cjlin/libsvm/

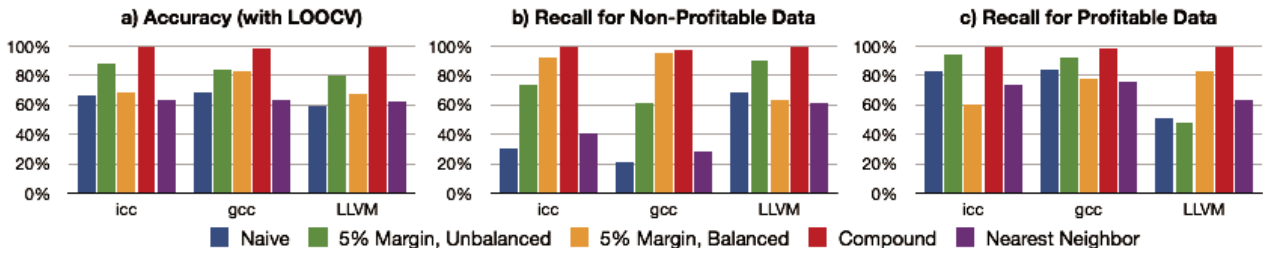

Fig. 3 Prediction accuracies for all methods and compilers: (a) the LOOCV total accuracies (b) the LOOCV accuracies per class.

Algorithm 1: Our LOOCV procedure.

```
   Input: data (array of pid, SSC, vect. prof.), pids (array of pid)
1  predictions ← ∅;
2  for test.pid ∈ pids do
3  |   test.set ← { d ∈ data, such as its pid is test.pid };
4  |   training.set ← data \ test.set;
5  |   model ← train.svm(training.set);
6  |   predictions ← predictions ∪ model(test.set);
7  end
8  return predictions
```

our LOOCV procedure reproduces precisely real-world situations. The predictions of our classifier, 0 or 1, are stored in the one-dimension vector *prediction*. The SVM model is trained at line 5, and stored in the variable *model* as a function: $SSC \mapsto vect.prof. \in \{0, 1\}$.

As a first approach, we try to predict vectorization profitability for all the data, using one model per compiler. We display the accuracy of this naive model using LOOCV in Fig. 3 a under the label *naive*. It ranges from 60% to 70%, which is rather low, and further hides large discrepancies between the recall for each classes as shown in Figs. 3 b and 3 c. The recall for class 0 is indeed under 30% for icc and gcc, and the recall for class 1 under 50% for LLVM. In other words, our predictor usually mis-predicts non-profitable examples for icc and gcc, and mis-predicts profitable ones for LLVM. On the other hand, NN exhibits lower overall LOOCV accuracy, but the recall is better balanced for each class. In overall, our predictor is not reliable and needs improvement.

5.2 A Better Modeling

Most of the mis-predictions for the naïve model occur when the vectorization speedup is close to 1. These are the programs for which vectorization has no significant effect. From the point of view of compilation, we do not care about correctly predicting these points; our focus is on situations for which vectorization is likely to bring significant speedup or slowdown. In other words, these mis-predictions are a consequence of a weakness in the way we model the problem: vectorization profitability should be defined with respect to a performance threshold above which performance difference is considered as significant. Starting from this section, we arbitrary set this threshold at 5%, that is, speedup and speed-down are only considered significant above 1.05 and under 0.95 respectively. Moreover, we classify the programs into the following three groups, noted 0,1, and 2: *profitable* ($speedup \geq 1.05$), *non-profitable* ($speedup \leq 0.95$) and *do-not-care* ($0.95 < speedup < 1.05$). The proportions of data in the class do-not-care for icc, gcc and LLVM are respectively 72.58%, 94.6% and 58.25%; the number of remaining data is therefore 762, 146 and 1128 respectively. With this definition, we consider that our predictor mis-predicts if and only if its prediction leads to a significant slow-down. In this context, we can recalculate the LOOCV accuracy as

$$accuracy = 1 - p(1|0) - p(0|1) \quad (1)$$
$$= p(0|0) + p(1|1) + p(0|2) + p(1|2) \quad (2)$$

where $p(a|b)$ is the probability to predict group a for a data in group b[†].

If we apply this definition with the data of the previous section, the new accuracies become 88.58%, 99% and 99% for icc, gcc and LLVM respectively. These figures are however artificially pushed up by the data in group 2 (do-not-care), that can never be mis-predicted. This may lead us to overlook some weaknesses of our predictors; we therefore decide to remove these data altogether when calculating the accuracy. It is important to notice that it does not affect the generality of our predictor, as mentioned in Sect. 6.1. Our new classification of the data further affect the training of the SVM. Indeed, we still use the same SVM as in Fig. 1. This SVM merely predicts group 0 and 1, but not 2 (do-not-care): mechanically, all the data from group 2 are not used for training anymore.

The new LOOCV procedure is similar as for the naïve predictor: we only need to remove all the elements of group 2 from the training set after the line 6 of algorithm 1. Moreover, the way we calculate accuracy is also different as explained in the previous paragraph. The accuracy and recalls for each class are shown in Fig. 3 under the label *5% margin, Unbalanced*. The accuracy has raised to between 80% and 90% for all the compilers. In particular, it is 84.25% with gcc despite the small size of its dataset[††]. The recalls for both class raise similarly; it however appears that all the

[†]There is no $p(2|x)$ because our machine learning device can only output 0 or 1, as explained in Sect. 3.1.

[††]Readers knowledgable in machine learning may further note that we have validated the correctness of the model with its learning curve, not shown in this paper.

models remain skewed, toward class 1 for icc and gcc, and toward class 0 for LLVM. We endeavor to address this weakness in nest section.

5.3 Balance the Data

In this section, we start by noticing that the skew of the predictor of the previous section is correlated with the bias inside the dataset itself: 73% of the programs are in class 1 for icc, 73% for gcc, and only 25% for LLVM. In this context, we decide to force both classes to be balanced by removing some data of group 0 or 1 from the training set before calling the procedure of previous section. Like the data cleaning introduced in previous section, this method only affects the way we prepare the training set, and not the generality of the predictor. The accuracy and recalls for this predictor are shown in Fig. 3, under the label *5% margin, balanced*. First, the recall for the less likely class has raised for all compilers, as expected: 92.61% and 94.87% for class 0 with icc and gcc respectively, and 82.5% for class 1 and LLVM. The overall accuracy as well as the recall for the other class has however dropped significantly, under the naïve predictor. In other words, this prediction model is over-fitted toward the less likely class. This has motivated our new compound predictor, introduced in next section.

5.4 Compound Predictor

In this section, we propose a new predictor based on the two models introduced in Sects. 5.2 and 5.3. Because it is made of two prediction models, we call it the *compound predictor*. Our objective is to combine the advantages of both: high accuracy for the most likely class for the former, and adjusted-bias for the less likely class for the latter. The new procedure is described in the pseudo-code in algorithm 2, where *loocv.svm(data.set)* returns the one-dimension-vector of prediction for the specified set using the LOOCV procedure of algorithm 1, and *train.svm(training.set)* trains a SVM using the specified set thereby returning a prediction model, that is, a function that predicts the vectorization profitability (0 or 1) from the SSC (see Sect. 5.1). The inputs of the prediction procedure are: *proba*, the three-dimension matrix of the probabilities calculated in the training procedure, *model.u* and *model.b*, the predicting classification functions obtained from tain.svm in the training procedure, and *test.sample*, a single line from the test set, that is, a tuple $\{pid, SSC\}$. In a nutshell, the new procedure is the following:
(1) we calculate the LOOCV predictions for each element of the training set, using the prediction procedures of Sects. 5.2 and 5.3 respectively (line 6 and 8);
(2) we calculate the conditional probabilities to hit each class from the predictions of each SVM (line 12 to 17);
(3) we also train unbalanced and balanced models from the whole training sets (line 7 and 9);
(4) when testing, we predict the vectorization profitability using the models trained in *(3)* (line 21 and 22);
(5) finally, we deduce the final prediction as the most probable one, given the predictions from *(4)*, from the probabilities pre-calculated in *(2)* (line 24).

Algorithm 2: The compound predictor.

```
1  **** Training **********;
   Input: training.st
2  training.set ← elements of training.set from groups 0 or 1;
3  training.set.balanced ← balance groups 0 and 1 in training.set ;
4  // Gets LOOCV predictions;
5  // (within the training set);
6  loocv.u ← loocv.svm(training.set);
7  model.u ← train.svm(training.set);
8  loocv.b ← loocv.svm(training.set.balanced);
9  model.b ← train.svm(training.set.balanced);
10 proba ← empty 3D array;
11 // Extracts Probabilities;
12 for i∈ [0,1] and j∈ [0,1] and k∈ [0,1] do
13     data.u ← data in loocv.u predicted as group i;
14     data.b ← data in loocv.b predicted as group j;
15     data.ub ← data.u ∩ data.b;
16     proba[i,j,k] ← probability for data in data.ub to be in group k;
17 end
18 return proba, model.u, model.b;
19 **** Prediction **********;
   Input: proba, model.u, model.b, test.sample
20 // Predicts with each SVM;
21 p.u ← model.u(test.sample);
22 p.b ← model.b(test.sample);
23 // Deduces the final prediction;
24 pred ← argmax(i,proba[p.u,p.b,i]);
25 return pred;
```

Figure 3 (label *compound*) confirms the relevance of this approach: the models are not skewed anymore with all the compilers, and the LOOCV accuracies are now above 99%. This is the best prediction technique we propose in this article.

5.5 Comparison with GCC Milepost

GCC Milepost is a tool based on GCC to extract SSC from benchmark for further optimization space exploration [6][†]. From the intermediate representation of GCC, it extracts 56 features that consists in some properties of the control flow graph and the instruction mix. In this section, we assess the accuracy of our SVM using GCC Milepost's SSC instead of our own set. First, one should note that GCC Milepost merely measures SSC at the function level instead of loop level in our work. For research purpose, we circumvented this problem by isolating loops in independent functions. Second, we also consider for comparison purpose a set of SSC made of random numbers, that is, that does not describe at all our benchmark.

Figure 4 shows the accuracy on icc's timing information of the models obtained with both sets of SSC, using the same setup as in Sect. 5.3. First, we can see that these numbers are similar than our numbers from 3 a for the naïve predictor. This hides strong bias toward the most likely class for both prediction models based on random and GCC Milepost. In fact, they almost exclusively predict the most likely class (1 in the case of icc): this makes them useless for any prediction. Second, these figures are significantly lower than

[†]It is now part of the cMind framework, available online: http://ctuning.org

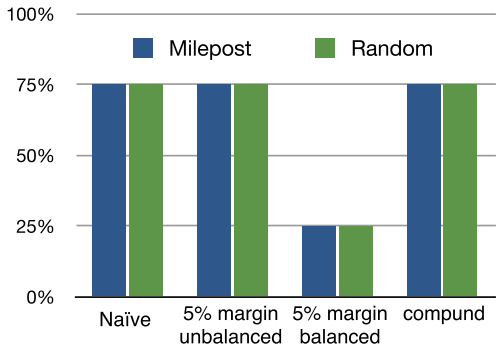

Fig. 4 The accuracy of the predictors for icc timing information, with GCC Milepost 2.1's SSC, as well as random numbers as SSC. This accuracy numbers hide strong bias in the recall per class.

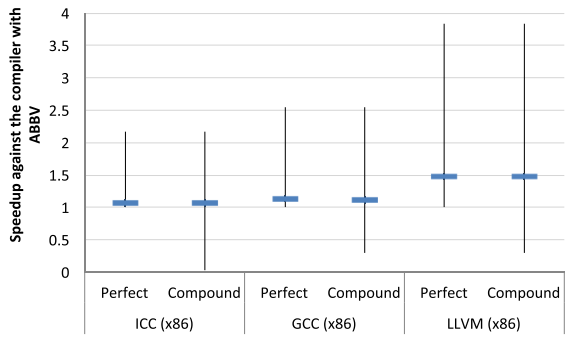

Fig. 5 Average, maximum and minimum speedups of our predictor (compound) and an hypothetical perfect predictor (perfect) against the compiler with vectorization always on.

the compound predictor with our own set of SSC. Third, we can see that GCC Milepost'SCC and random numbers yield the same accuracy: in other words the former does not provide any useful information about the benchmark to the machine learning models. We believe that this is because TSVC is not a natural fit for GCC Milepost, mostly tested using MiBench [6]. Indeed, TSVC kernels are very similar from the point of view of their control-flow graph and instruction mix. Their differences rather lie into their memory access patterns (especially alignment) and data-flow graph, which GCC Milepost does not measure at all.

6. Improvement of the Compilation Flow

6.1 Performance Improvement due to Our Compound Predictor

In this section, we assess the performance improvement we can achieve by using our compound predictor proposed in Sect. 5.4. To so, we predict the group for all the data with the compound predictor, 0 or 1, using an LOOCV procedure that wraps the procedures of algorithm 2. It is important to notice that the procedure predicts whether group 0 or 1 for all data, including those in group 2. That means that elements from group 2 are always predicted in class 0 or 1, but never in 2. Then, we compile programs using ABBV if the predicted group is 1, or without if it is 0, using options detailed in Table 2. The LOOCV procedure used in this section is similar to the one described in Sect. 5.1; in particular it makes sure that the data lines related to a given program do not span across both the training and test sets. This procedure finely reproduces real-world prediction situations. We measure the execution time of each so-compiled programs, and calculate the speedup compared to the case where ABBV is activated as follows:

$$speedup = \frac{exec.\ time\ with\ ABBV}{exec.\ time\ (ABBV\ as\ predicted)} \quad (3)$$

The speedup numbers calculated under this setup are displayed in Fig. 5. We consider three metrics: the average speedup, the maximum speedup and the minimum speedup, respectively symbolized on the figure by the horizontal bars and both ends of the vertical bars. Values under 1 correspond to slowdown. By correctly predicting situation where vectorization is not profitable, we manage to reach speedups of up to 3.8 times. On the other hand, we may incorrectly predict profitable vectorization, thereby slowing down the generated program. Still our predictor always provides speedup in average. It is particularly efficient for LLVM, where programs compiled with our predictor are in average 47.3% faster than without. We further display for each compiler the same numbers for a perfect predictor. Our predictor's numbers are always very close to the perfect predictor, that is, our method is close to the optimum.

Another interesting metrics is the unprofitability ratio of compilers when using our predictor, as introduced in Sect. 4.3. These numbers for icc, gcc and LLVM on the data without noise are respectively 26.64%, 26.71% and 75.18%†. When using our compound predictor, these numbers drop down to 0.13%, 0.68% and 0.09%.

6.2 About Mis-Predictions (Case of Intel Compiler)

The compound predictor merely mis-predicts two programs with icc. We can find several explanations for the predictor being mis-led. First, both mis-predictions occur at unroll factor 12. Indeed, whereas 12 is multiple of the width of our SIMD datapath in single precision floating point, our data show that the compiler sometimes decides not to vectorize at this level for some reasons we were not able to understand††. The second possible cause for our predictor to be mis-led can be found in the source code of each kernel. The first TSVC kernel to be mis-predicted is called *s3111*, shown below:

```
1 #define LEN 32000
2 void s3111() {
3   float sum;
4   for (int nl=0; nl<ntimes/2; nl++) {
```

†These numbers are different from the one of Fig. 2, which also consider the data inside the 5% margin.

††This may be a side-effect of the heuristics used internally by the compilers in order to predict the profitability of vectorization before actually applying ABBV.

```
5       sum = 0.;
6       for (int i=0; i<LEN; i++)
7         if (a[i]>(float)0.) sum+=a[i];
8     }
9   }
```

This program does not profit from vectorization regardless the unroll factor because it contains a branch at line 7. In fact, this program is in class 2 for all unroll factors but 12: this data line is a singularity. Moreover, our SSCs do not measure the control flow graph although it would have been important in this case. The second TSVC kernel to be mis-predicted is called *vpvts*, shown below:

```
1  #define LEN 32000
2  void vpvts( float s) {
3    for (int nl=0; nl<ntimes; nl++)
4      for (int i=0; i<LEN; i++)
5        a[i]+=b[i]*s;
6  }
```

This kernel profits from vectorization regardless the unroll factor, whereas we predict the opposite for an unroll factor of 12. It involves a variable inside the calculations, which might have influenced the prediction toward the wrong class. Yet, we notice that we correctly predict it in class 1 for other unroll factor: 12 is a singularity of our model. Therefore, it appears that this mis-prediction is a glitch of our SVM.

7. Concluding Discussion

In this paper, we use support vector machines (SVM) to predict the profitability of automatic basic-block vectorization for Intel Compiler, the GNU C Compiler and LLVM. As input programs, we consider TSVC, a benchmark made of 151 simple yet representative loops, unrolled by a factor ranging from 1 to 20. Our key innovation is to formulate vectorization profitability in terms of a classification problem, and we use SVM to solve it. We first obtain 70% cross-validation accuracy by naively applying SVM. After some improvements including combining the predictions of two different SVMs, we manage to reach 99% cross-validation accuracy.

This technique is very useful because it allows the compiler to avoid, in most cases, applying vectorization when it is not profitable to do so. We show that by doing so, it is possible to speedup compiled programs by up to 2.16 times for Intel Compiler, 2.5 times for the GNU C Compiler, and 3.82 times for LLVM. Moreover, it enables to reduce the probability for the compiler to generate slower programs with vectorization to less that 1%, from more than 25% without. Moreover, the overhead on the compilation time is very low: the predictions with SVMs required less than 1 second on our test machine[†]. Our method however requires the compiler to be trained once on some benchmark programs before the first prediction to be made; this may take from 10 minutes with a traditional SVM, up to several hours with our compound predictor. Still, if we consider a compilation flow in which the compiler is trained on the user's machine upon installation in order to tune itself to its environment, this training is to be done only once; therefore we believe it is not an obstacle to its use on real-world compilers.

Some important challenges remain to be solved before our work can be applied to more complex programs. First, it is challenging to measure most of our software characteristics on programs which loop boundaries involve non-linear expressions. Second, we have seen in Sect. 6.2 that it might be relevant for those characteristics to better express the control flow-graph of programs. This will constitute our main focus for further research.

[†]An Intel Core2 Duo Merom at 2.6GHz.

References

[1] S. Kulkarni and J. Cavazos, "Mitigating the compiler optimization phase-ordering problem using machine learning," OOPSLA, pp.147–162, 2012.
[2] E. Park, J. Cavazos, and M.A. Alvarez, "Using graph-based program characterization for predictive modeling," International Symposium on Code Generation and Optimization (CGO), pp.196–206, 2012.
[3] K. Stock, L.N. Pouchet, and P. Sadayappan, "Using machine learning to improve automatic vectorization," ACM Transaction on Architecture and Code Optimization (TACO), vol.8, no.4, pp.1–23, 2012.
[4] S. Maleki, G. Yaoqing, M.J. Garzaran, T. Wong, and D.A. Padua, "An evaluation of vectorizing compilers," Parallel Architecture and Compilation Techniques (PACT), pp.372–382, 2011.
[5] S. Kamil and A. Fox, "Bringing parallel performance to Python with domain-specific selective embedded just-in-time specialization," 10th Python for Scientific Computing Conference, 2011.
[6] G. Fursin, Y. Kashnikov, A.W. Memon, Z. Chamski, O. Temam, M. Namolaru, E. Yom-Tov, B. Mendelson, A. Zaks, E. Courtois, F. Bodin, P. Barnard, E. Ashton, E. Bonilla, J. Thomson, C.K.I. Williams, and M. O'Boyle, "Milepost GCC: Machine learning enabled self-tuning compiler," International Journal of Parallel Programming, vol.39, pp.296–327, 2011.
[7] L. Van Ertvelde and L. Eeckhout, "Benchmark synthesis for architecture and compiler exploration," International Symposium on Workload Characterization, pp.1–11, 2010.
[8] M.-W. Benabderrahmane, L.-N. Pouchet, and A. Cohen, "The polyhedral model is more widely applicable than you think," ETAPS International Conference on Compiler Construction (CC'2010), pp.283–303, 2010.
[9] G, Pekhimenko and A.D. Brown, "Efficient program compilation through machine learning techniques," International Workshop on Automatic Performance Tuning (iWAPT), Oct. 2009.
[10] K. Hoste and L. Eeckhout, "Microarchitecture-independant workload characterization," MICRO, vol.27, no.3, pp.63–72, 2007.
[11] C.M. Bishop, Pattern Recognition and Machine Learning, Springer, 2006.
[12] F. Agakov, E. Bonilla, J. Cavazos, B. Franke, G. Fursin, M.F.P. O'Boyle, J. Thomson, M. Toussaint, and C.K.I. Williams, "Using machine learning to focus iterative optimization," International Symposium on Code Generation and Optimization (CGO), pp.295–305, March 2006.
[13] M. Stephenson and S. Amarasinghe, "Predicting unroll factors using supervised classification," International symposium on code generation and optimization, pp.123–134, 2005.
[14] R. Allen and K. Kennedy, Optimizing Compilers for Modern Architectures, Morgan Kaufmann, 2002.

Antoine Trouvé received his master degree in 2006 from the Computer Department of ENSEIRB-MATMECA, Bordeaux (France). He later received his Dr. Eng. degree in 2011, from the Department of Informatics of Kyushu University, Fukuoka (Japan). He is now working as a researcher at the Institute of Systems, Information Technologies and Nanotechnologies, Fukuoka (Japan). His research interests include embedded systems, reconfigurable processors, high-performance computing, processor architecture, optimizing compilers and workload characterization.

Arnaldo J. Cruz is a PhD candidate at the Department of Informatics of Kyushu University, Fukuoka (Japan). His research interests include embedded systems, high-performance computing, optimizing compilers and workload characterization.

Dhouha Ben Brahim is a graduate student at the Computer Department of ENSEIRB-MATMECA, Bordeaux (France). Her research interests include high-performance computing and optimizing compilers.

Hiroki Fukuyama received his master degree in 2012 from the Department of Informatics of Kyushu University, Fukuoka (Japan). He is now a PhD candidate in the same laboratory. His research interests include embedded systems, high-performance computing, processor architecture and workload characterization.

Kazuaki J. Murakami received the B.E., M.E., and Ph.D. degrees in computer science and engineering from Kyoto University in 1982, 1984, and 1994, respectively. From 1984 to 1987, he worked for the Fujitsu Limited, where he was a Computer Architect of the mainframe computers. In 1987, he joined the Department of Information Systems of Kyushu University, Japan. He is currently a Professor of the Department of Informatics, and also the Director of the Computing and Communications Center. He is a member of the ACM, the IEEE, the IEEE Computer Society, the IPSJ, and the JSIAM.

Hadrien Clarke received his master degree in 2009 from the Electronic Department of ENSEIRB-MATMECA, Bordeaux (France). He is now a PhD candidate at the Department of Informatics of Kyushu University, Fukuoka (Japan). He is also working as a researcher at the Institute of Systems, Information Technologies and Nanotechnologies, Fukuoka (Japan). His research interests include embedded systems, high-performance computing and processor architecture.

Masaki Arai recieved the B.S. degree in information engineering from Tokyo University of Technology in 1990 and the M.S. degree in information science from Tokyo University of Science in 1992. In 1992, He joined Fujitsu Laboratories Ltd. His research interests are in the area of compiler optimizations and computer architectures.

Tadashi Nakahira is research manager at the systems laboratories of Fujitsu Laboratory Limited. His research interests include compiler optimizations and high-performance computing.

Eiji Yamanaka is director at the next-generation-technical-computing unit of the softawre-developpment division at Fujitsu Limited. His research interests include compiler optimizations and high-performance computing.