Machine Learning Driven Compiler Tuning

Author: Arnaldo J. Cruz-Ayoroa
Supervisor: Prof. Kazuaki Murakami

A thesis submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy

Department of Advanced Information Technology of the Faculty of Information Science and Electrical Engineering

Kyushu University

December 2015

Abstract of Thesis Presented to the Graduate School
of the Kyushu University in Partial Fulfillment of the
Requirements for the Degree of Doctor of Philosophy

Machine Learning Driven
Compiler Tuning

By

Arnaldo J. Cruz-Ayoroa

December 2015

A compiler is a software tool that translates human-readable programs into executable machine code. As part of the translation process, compilers have the important task of automatically applying sequences of program optimizations in order to produce an executable that makes efficient use of hardware resources.

This work investigates methods for boosting the capacity of compilers at selecting beneficial optimizations for speeding up programs. The research is motivated by the observation that the conventional methods employed by production compilers for choosing program optimizations fall shot in achieving efficient use of hardware resources, due to the difficulty of modeling modern hardware, compiler and program behavior. To this end, the research focused on how to apply machine learning techniques for predicting from a high-level source program which optimizations would be beneficial in order improve, or *tune*, compiler performance.

Vectorization was selected as the first optimization technique to be predicted due to its potential for providing dramatic speedups, while also being detrimental if not used correctly. A predictor was trained with up to 98% of accuracy in predicting vectorization profitability from a high-level characterization of the experimental

benchmark. The methodology was also demonstrated to be useful in reducing power and energy consumption, resulting in an average of 64% decrease in consumption and only 5% increase in the case of mispredicitons. For predicting multiple optimizations, this work proposes an optimization space pruning technique that makes score-based predictors, commonly used in related works, feasible by bounding the prediction time. This was achieved without decreasing prediction accuracy on the experimental benchmark, but rather increasing it by up to 40% in speedup over the original score-based prediction scheme. Along with these promising results, this work also identifies several challenges that need to be addressed in order to bring machine learning driven compilation to the hands of programmers who want the compiler to make the best use of the available resources in an automated fashion.

Dedication and Acknowledgements

I dedicate this work to my dear parents, brother, family and girlfriend. A warm hug for them, for their selflessness and for their unconditional support.

My appreciation to Murakami-sensei for accepting me in the laboratory first as an intern, and then as a Ph.D. student, and for making sure all my needs were met. And to Antoine-sensei for leading the project, and for being not only a good friend but also a *senpai* I can look up to.

Thank you to the Kyushu University staff for their professionalism and for making the students feel at ease. To Fujitsu Laboratories Ltd. for sponsoring the project and for their advice. And to Monbukagakusho (MEXT) for providing me with a scholarship that covered all my living and studies expenses.

And thanks to my colleagues, new friends and old friends, some of which are far away, but only in space.

My deepest gratitude to all, for making this work and my wonderful experience of living in Japan possible. May this be one step towards many more amazing things to come.

Table of Contents

Chapter 1
Introduction

A compiler is a software tool that translates human-readable programs into executable machine code. As part of the translation process, compilers have the important task of automatically applying sequences of program optimizations in order to produce an executable that makes efficient use of hardware resources. We term an *optimization strategy* as a sequence of one or more program optimizations and their corresponding parameters, all of which are applied to a given input program. Modern-day compilers rely on a few predefined optimization strategies, otherwise known as *optimization levels*, and static performance models of the hardware architecture to estimate whether a given optimization would be beneficial. However, in this work we show that there is no single optimization strategy that will work well in the general case; that some will improve the program's performance, while others will have no significant impact, or may actually have a large detrimental effect.

Modern compilers fall short in selecting optimizations that yield the highest speedups because they apply an optimization level to any input program while mostly disregarding its specificities. As motivational example, Figure 1.1 shows the speedup results of an experiment we conducted to test Intel's ICC compiler ability at optimizing 1500 tensor contraction kernels. Tensor contraction kernels are generalized multiplication programs that are often used in computational chemistry suites, and are structurally simple but exhibit complex memory access patterns, which make them useful for benchmarking compiler optimization performance. Listing 1.1 shows

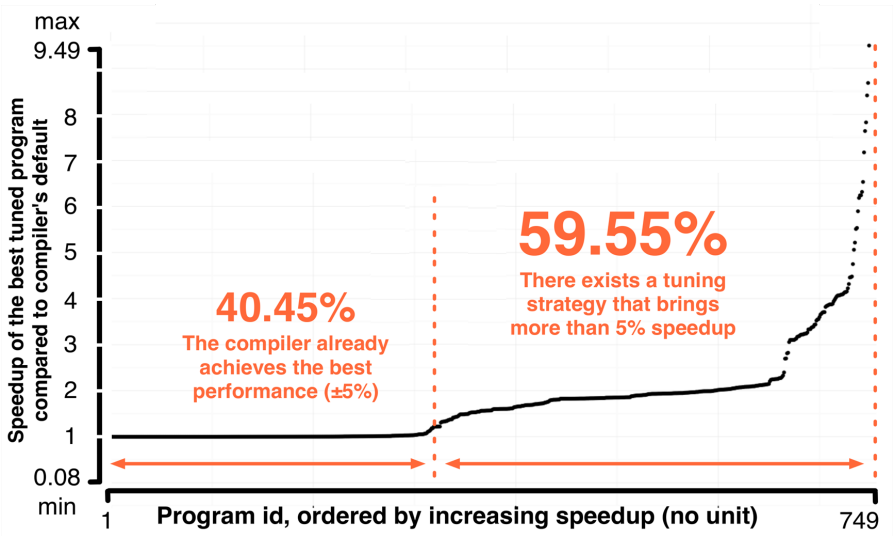


Figure 1.1 : Intel's ICC compiler's performance in optimizing tensor contraction kernels.

an example tensor contraction, such as the ones used in this experiment. We considered loops of depth 4 and array sizes of 100, optimized with a relatively small optimization space of 9505 strategies, for a total of 14.26 million data points. Figure 1.1 shows that for 59% of our test programs there was at least one optimization strategy that would provide more than 5% speedup than the one chosen by ICC, and in a significant number of cases resulting in more than twice the speedup.

```
float A[100][100]
float B[100][100][100];
float C[100][100][100];
for (int i = 0; i < 100; i++)
    for (int j = 0; j < 100; j++)
        for (int k = 0; i < 100; k++)
            for (int l = 0; l < 100; l++)
                A[i][j] += B[i][k][l] * C[j][k][l]
```

Listing 1.1: Tensor contraction benchmark example of depth 4

More recent compiler research has shown promising results in the application of machine learning techniques to improve compiler performance, an approach that we term Machine Learning Driven Compiler Tuning (MLDCT). In MLDCT the hardware and compiler are treated as a black box and a predictor is trained with benchmark programs to learn to recognize which types of programs benefit from which optimizations strategies. This is an attractive approach, because it eases the burden of developing complex optimization selection schemes with a flexible solution that is comparatively easy to adapt to different compilers and hardware, while at the same time improving the baseline compiler performance.

In this work we explore how MLDCT can improve a compiler's performance, starting with predicting the effectiveness of a single optimization and then progressing to predicting multiple optimization scenarios. In terms of performance, we considered not only reducing execution time, but also energy reduction mobile System-on-Chip devices. We compare our approach to the sate-of-the art works in MLDC, and highlight the need for more rigorous predictor evaluation criteria. Our study gives insight not only on the strong points of MLDCT, but also highlights its weaknesses. This work also describes the challenges faced in making MLDCT a practical solution, and proposes program characteristics visualization and optimization space reduction techniques to help ameliorate some of these challenges.

Chapter 2
Problem Definition

Programmers rely on compilers to transform their resource demanding high-level programs to an executable that makes best use of its available resources. In this work we address the question of how to assist the compiler in selecting beneficial optimization strategies in order to improve its optimization capacity at speeding up programs. Figure 2.1 depicts an overview of our proposed solution.

In a standard compilation flow, the programmer would input a high-level source program to the compiler. The experiments in this work were conducted on computational kernels written in the C programming language. The compiler would then apply a generic optimization strategy according to the optimization level option set by the programmer, and produce an equivalent but optimized machine executable program. *Optimization* in the context of compilers refers to a program transformation that is expected to improve some aspect of the program, such as execution time or

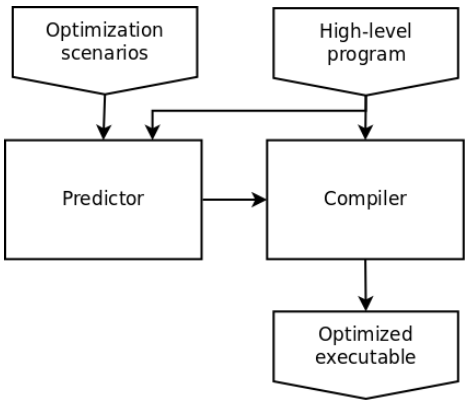


Figure 2.1 : Overview of the proposed compiler tuning strategy.

storage size, and does not refer to the mathematical definition of finding an optimum solution. In compilation, a sub-optimal solution is common and can be acceptable. We did, however, compare selected optimizations strategies to the best strategy to evaluate our approach. In our experiments, we considered the mainstream production compilers Intel's ICC, GNU's GCC, and Apple's LLVM. We experimented on a single core, to reduce the number of experimental factors. Parallel and heterogeneous systems also offer interesting research opportunities, but these were beyond the scope of this work.

Referring once more to Figure 2.1 , our task was to devise a predictor that would recognize the high-level input program, select from a set optimization strategies which optimizations would be beneficial for the particular program, and instruct the compiler as to which optimization strategy to apply. This predictor would model the hardware and the compiler's behavior, and would be accurate at predicting optimization strategies for an unknown input program to improve or *tune* a compiler's performance. Performance improvements were measured mainly as execution time speedup, although in Chapter 5 we also present experiments regarding energy consumption reduction, which is a less researched aspect of software performance. Speedup was defined as an improvement in a program's performance compared to the optimization strategy that a compiler would have made without the predictor's aid. Likewise, we defined a speed-down as a detriment in a program's performance caused by our predictor in comparison the optimization strategy the compiler would have chosen by itself.

The optimization strategy is presented to the compiler as a series of command line flags to drive a certain set of optimizations within the compiler. We studied the commonly used loop optimizations vectorization, loop nest interchange, tiling and unroll-and-jam. Still, the compiler would be free to apply other optimizations it estimates would be beneficial. In fact, although compilers provide an interface to

alter optimizations and their parameters, there are still some decisions that are taken by the compiler for which the programmer has no way to control.

We hypothesized that machine learning would be an effective way of predicting from high-level source program when to apply a given optimization scenario to boost compiler performance. The reasoning is that the nature of the problem involves recognizing complex input patterns and mapping them to an output as fast as possible, a task well suited for machine learning algorithms. Furthermore, there are several ways of modeling the optimization strategy prediction problem using machine learning, and we study several of them in this work.

This research entailed the following main questions, each of which we address in the following chapters of this document:

- At which stage of a programs lifetime should the optimization be performed? The alternatives were compile time, installation time, when the program is being executed or at idle times when it is not running
- At what code level should estimation and optimization be performed? The alternatives were high, intermediate, and low level.
- How to model the problem in terms of a predictor? Which prediction technique works best, how to encode the programs (input) and the optimization scenarios (output)?
- How do we assess the effectiveness of the predictor?
- How general is the predictor? That is, for which programs can we predict? What degree of portability can be achieved in terms of compiler and hardware architectures?
- How practical is the predictor for being used in a production environment?

Chapter 3
Predicting from Hardware Independent Metrics

This chapter describes our first approach at devising an optimization strategy predictor for compiler tuning. This was an intuitive approach, in which we studied the relationship between Hardware Independent Metrics (HIM) and how programs performed after applying different compiler optimization strategies. HIM are statical and dynamical metrics that describe a program's behavior while being architecture agnostic. Establishing a relationship between programs with similar HIM and how they perform for a set of optimization strategies would lay the groundwork for developing an optimization strategy predictor.

3.1 Methodology

Our methodology was based on the work of Hoste and Eckout, who researched the clustering of applications with respect to a set of 47 HIM [25]. The authors managed to predict the Instructions-Per-Cycle (IPC) of the SPEC CPU 2006 benchmark programs from the IPC of a single application per cluster of benchmarks with similar HIM, achieving under under 5% of prediction error. Finding a set of HIM that correlate with program performance could allow for the classification of programs that benefit from the same set of optimization strategies, in an architecture-agnostic fashion. Thus we set out to investigate a similar method that considered program speedup due to the application of optimization strategies, rather than IPC. If a relationship between HIM and speedup were to be found, this approach could be used for predicting beneficial optimization strategies for a given input program. That is,

the predicted optimization strategies for a new program would be those of the closest HIM vector cluster of previously profiled programs.

We instrumented the SPEC CPU 2006[a] benchmark suite using Intel's Pin binary instrumentation tool and the MICA[b] extension developed by Hoste and Eckout for HIM profiling. SPEC was chosen because it covers a large range of program behaviors [18]. A limitation of this technique was that the execution of the instrumented code was very slow, close to 1000 times slower than the original code.

Each benchmarks was compiled with LLVM compiler's optimization levels O1 to O3, and their execution time for each optimization level was recorded. We selected LLVM because it was designed as an open source compiler toolkit, which allows greater degree of the optimization process. Our test machine was an Intel XEON E5520 2.7 GHz (3 cores and 6 hardware threads) with 12GB of memory.

Next we describe the HIM profiled for our benchmark, as specified the MICA documentation.

Instruction level parallelism. The ILP for an idealized out-of-order processor that only depends on the instruction window size and the data dependences between instructions. The default window sizes are 32, 64, 128, and 256 instructions long.

Instruction mix. Executed instructions are counted according to these categories: memory read, memory write, control flow, arithmetic, floating-point, stack operation, shift, string, vector, nop.

Branch predictability. Branch predictability using the Prediction-by-Partial-Match (PPM) method. The total branches, total transitions, and total branch taken counts are also profiled.

---

[a] http://www.spec.org/

[b] http://boegel.kejo.be/ELIS/mica/

Register traffic. Three metrics are gathered concerning register usage: (1) the average number of input operands in the executed instructions, (2) the average degree of use which is the average number of times a register is read after being written, (3) the register dependency distance, or the number of executed instructions between a register being written and then read. These are measured in buckets of powers of two between 1 and 64.

Data stream strides. Two types of strides are counted; global and local. A global stride is the distance between memory addresses in consecutive accesses. The local stride is the distance between addresses for a single load or store instruction. This is profiled separately for load and store instructions. These are measured in buckets of powers of eight between 0 and 262144.

Memory footprint. The number of memory accesses to 64B blocks and 4kB pages, separated by data and instruction accesses.

Memory reuse distance. Given a read access to a 64B cache block, the memory reuse distance is the number of unique blocks accessed until a second access to the original block. This is done using an LRU stack and measured using buckets in the range [2n, 2 (n+1)] for $0 \leq n \leq 18$. The bucket 0 also counts the cold references, or the number of blocks which are only accessed once. The last bucket also includes all other accesses greater than 219.

3.2 Results

In the first experiment, preliminary results were obtain by arbitrarily selecting single-optimization strategies. The following is the list of strategies, as reported in the LLVM documentation[c] :

adce. Aggressive Dead Code Elimination

[c] http://llvm.org/docs/Passes.html

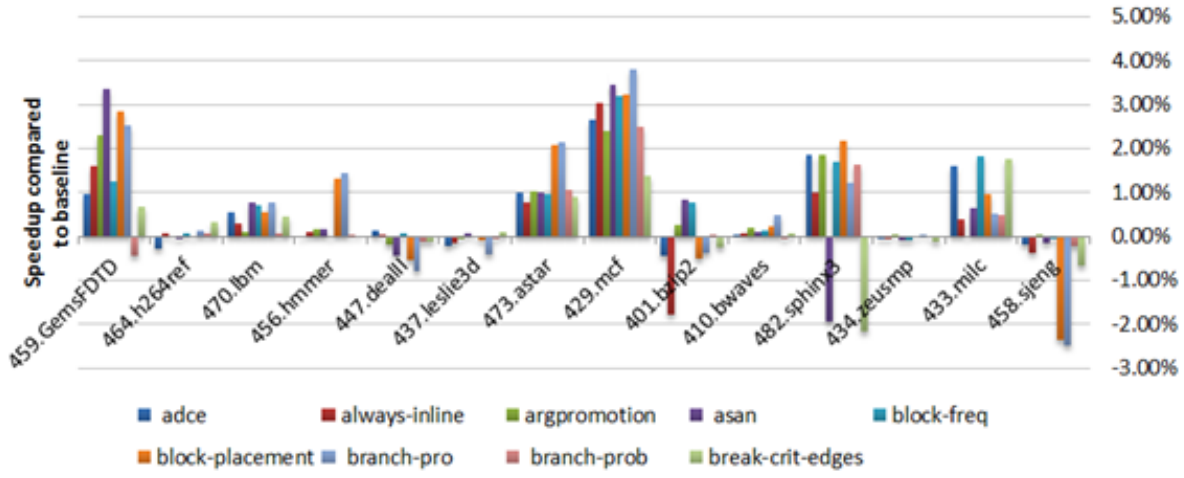


Figure 3.1 : SPEC speedup due to individual LLVM optimizations against O0

always-inline. Inliner for always-inline functions

argpromotion. Promote 'by reference' arguments to scalars

asan. Detect use-after-free and out-of-bounds patterns

block-freq. Block Frequency Analysis

block-placement. Profile Guided Basic Block Placement

branch-prob. Branch Probability Analysis

brak-crit-edges. Break critical edges in CFG

The speedup due to LLVM optimizations is shown in Figure 3.1 , for a subset of SPEC and LLVM optimization techniques. The numbers are expressed in terms of speedup against the non-optimized program, or LLVM optimization level O0. Results vary depending on the benchmark programs and the type of optimization, suggesting that there is no single optimization has the same effect across programs. The speedup range is limited, however; the maximum speedup is less than 4%.

Nevertheless, the HIM versus the speedup was plotted as shown Figure 3.2 . In this plot, one point represents the distance between a pair of programs. The vertical axis represents the distance with respect to HIM. That is, let us consider two

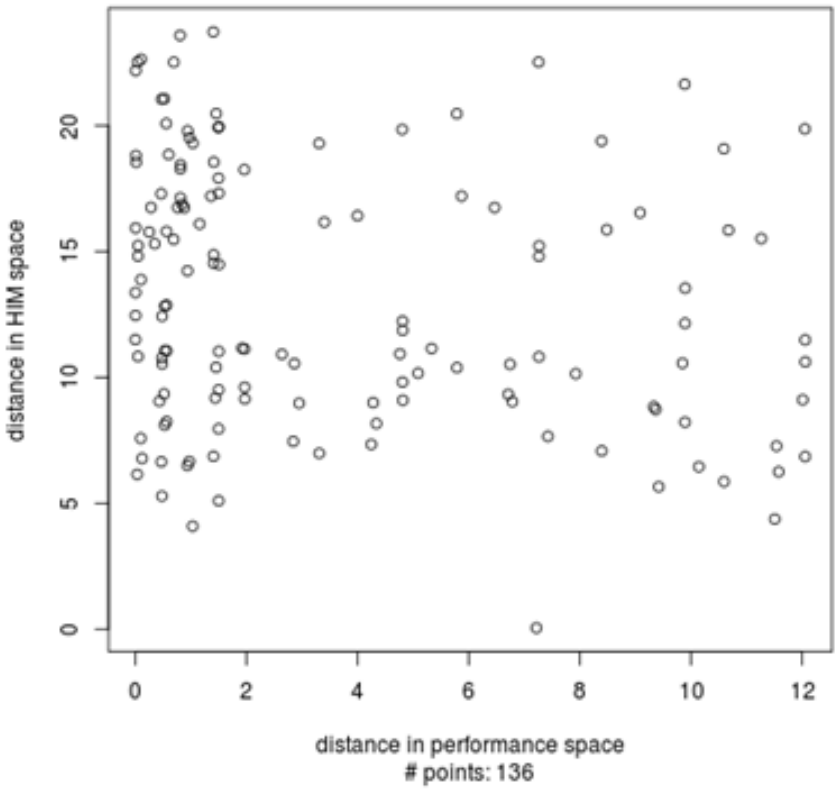


Figure 3.2 : Relationship between variations of HIM and speedup due to individual LLVM optimizations

benchmarks A and B and represent their HIM in n-dimension vectors for n HIMs, denoted as a and b. The value reported vertically is the Euclidean distance between a and b. The horizontal axis is plotted similarly but for speedup vectors resulting from the independent application of each selected LLVM optimization technique.

No pattern can be recognized in this plot, and two factors may explain these results. First, the SPEC benchmark was designed for hardware profiling and not for compiler profiling. Hence, the code may already be hand-optimized and it may not provide any more room for compiler optimizations. As a consequence, the performance results plotted in Figure 3.1 may correspond to experimental noise. Second, the optimization techniques of LLVM are not meant to be applied independently, but to be combined together instead. Some of these optimization techniques are mainly

enabling transformation, that is, they do not provide speedup by themselves but enable further optimizations downstream.

Next, we repeated the experiment but with optimization strategies of length greater than one. The strategies were determined by the list of optimizations carried out by LLVM compiler level O1, and by trusting our intuition. The following are the optimization strategies for the second experiment, whose explanation can be found in LLVM's documentation [d] :

z0. -scalarrepl -mem2reg -scalarrepl -mem2reg

z1. -early-cse

z2. -scalarrepl -mem2reg -scalarrepl -mem2reg -early-cse

y0. -simplify-libcalls

x0. -jump-threading -instcombine

x1. -correlated-propagation -instcombine

x2. -jump-threading -correlated-propagation -instcombine

w0. -tailcallelim -simplifycfg

a0. -reassociate -loop-rotate -instcombine

a1. -reassociate -loop-unswitch -instcombine

b0. -reassociate -loop-unroll -instcombine

b1. -reassociate -indvars -loop-deletion -instcombine

b2. -reassociate -loop-idiom -loop-deletion -instcombine

b3. -reassociate -indvars -loop-idiom -loop-deletion -instcombine

b4. -reassociate -indvars -loop-idiom -loop-deletion -loop-unroll -instcombine

c0. -reassociate -memcpyopt -dse -adce -simplifycfg -instcombine

c1. -reassociate -sccp -dse -adce -simplifycfg -instcombine

[d] http://llvm.org/docs/Passes.html

c2. -reassociate -memcpyopt -sccp -dse -adce -simplifycfg -instcombine

d0. -reassociate -jump-threading -dse -adce -simplifycfg -instcombine

d1. -reassociate -correlated-propagation -dse -adce -simplifycfg -instcombine

d2. -reassociate -jump-threading -correlated-propagation -dse -adce -simplifycfg -instcombine

All the optimizations strategies were prefixed by *-always-inline -globalopt -ipsccp -deadargelim -instcombine -prune-eh -mem2reg* and suffixed by *-strip-dead-prototypes*, as is done for optimization level O1. The prefix and suffix also constituted the baseline that was used for computing the speedup numbers. The most important optimization included into the baseline is *memtoreg*, which allocates variables and function parameters to registers instead of the stack when possible. Without this optimization, all read and write to variables are considered as memory accesses, and programs spend most of their time doing such operations, thereby canceling out the effect of other optimizations. After conduction the second experiment however, we obtain similar results to the first experiment.

For the third experiment, we tried the fixed optimization strategies provided by LLVM in its optimization levels O1 (less aggressive optimizations) to O3 (most aggressive optimizations). The speedup results are shown in Figure 3.3 , with the baseline being the same as in experiment two. The correlation between distances in the space of both HIM and optimization strategies is shown in Figure 3.4 . From these data, we can see that no obvious correlation between HIM and speedup due optimization can be found, and that SPEC still shows little response to the optimization strategies. Only two applications show a speedup of more than 50%; dealII and soplex benchmarks.

These experiment failed to find a relation between HIM and performance. They did, however, provide us with valuable insights into the optimization strategy prediction problem. We discovered that hardware performance benchmarks are not

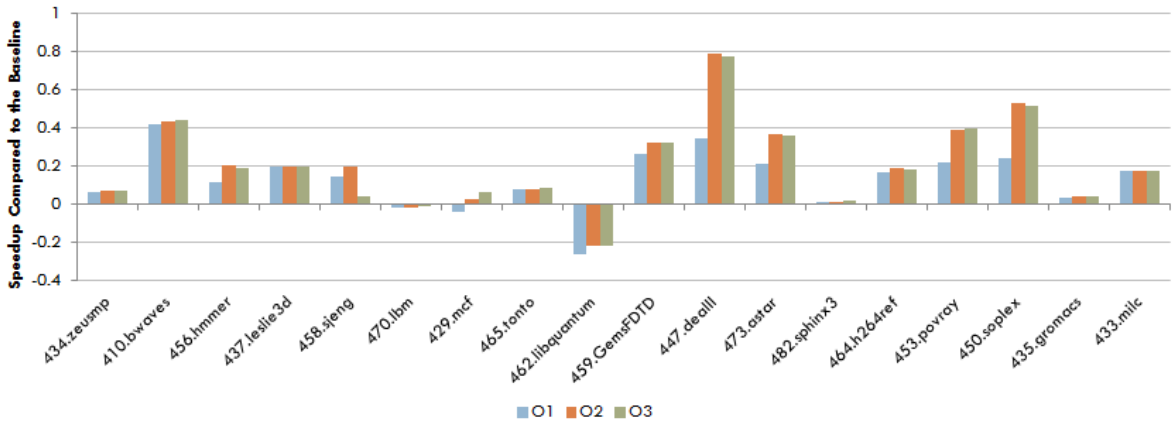


Figure 3.3 : SPEC speedup due to LLVM optimizations levels O1, O2, and O3 against O0

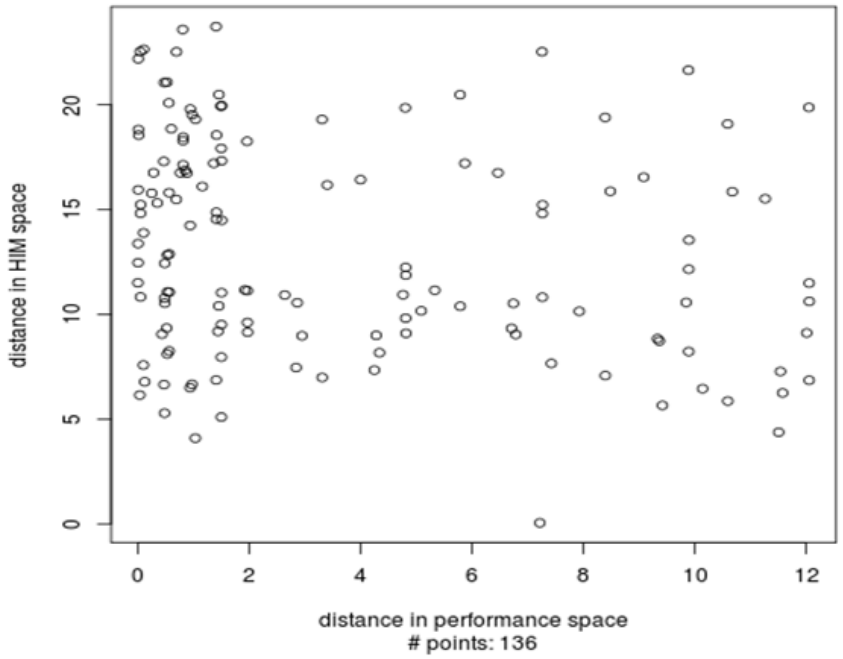


Figure 3.4 : Relationship between variations of HIM and speedup due to LLVM optimization levels O1, O2, and O3

appropriate for compiler tuning research, because they are already optimized. The questions remains whether there could a different HIM set that does relate to performance. Another question is what kind of relationship do we expect between HIM distances and performance distances, for example whether it's a linear or not linear relation. We address each of these issues in the next chapter.

Chapter 4
Predicting Vectorization for Execution Time Reduction

In Chapter 3 we presented our first approach at predicting optimization strategies by relating vector distances between high-level hardware independent characteristics and program performance. As it turns out, there is a branch of artificial intelligence called *machine learning* designed precisely for this type of problems, with algorithms that can find patterns within data and for recognizing input objects and associating them to expected outcomes. In this chapter we first give an overview of what is machine learning and how it can be adapted to the compilation process. This is followed by our experiments in applying machine learning techniques for the prediction of a widely used data parallelization technique called vectorization, with the objective of reducing execution time.

4.1 Machine learning driven compiler tuning

Machine learning (ML) is a field in computer science concerned with the development of algorithms that can automatically find structure within data and predict future outcomes. Although it employs statistical techniques, it differs from statistics in its objective. Whereas in statistics the focus is to model and understand the data, ML places emphasis on prediction. That is, the usefulness of a ML model is measured by how well it can predict new cases. It is important to understand that ML and statistics have different objectives, because there are data manipulation techniques

that are used in ML in order to increase prediction precision, or the predictor's usefulness, that are considered inappropriate in statistics. The difference between both fields is eloquently explained by Breiman in his work titled *Statistical Modeling: The Two Cultures* [30]. The ML concepts to be presented in this section are based on Bishop's book titled *Pattern Recognition and Machine Learning* [26].

ML is subdivided into unsupervised and supervised type of algorithms. In unsupervised learning, we have training examples for which the targets are not known and we want to give some label to each point. A common application is the use of clustering algorithms which label groups of points that are close together. For example, we may want to classify programs in terms of the instruction mix, which is the number of counts per instruction. This could be useful to classify programs as computation or memory bound. The feature vector for each program would be the instructions mix and the unknown target or label would be the cluster to which it belongs. Applying a clustering algorithm would reveal which programs are similar in terms of their instruction mix and allow classification as computation or memory bound. A well-known example of clustering algorithms is called *k-Means*. The objective of *k-Means* can be described as finding the coordinates of k points such that the Euclidean distance of each point to its neighbor is minimized. At each iteration step the k points are moved independently towards clusters of points until convergence, where each of the k points becomes the centroid of a cluster of training examples. After training, prediction can also be performed by finding the closest centroid to the new feature vector.

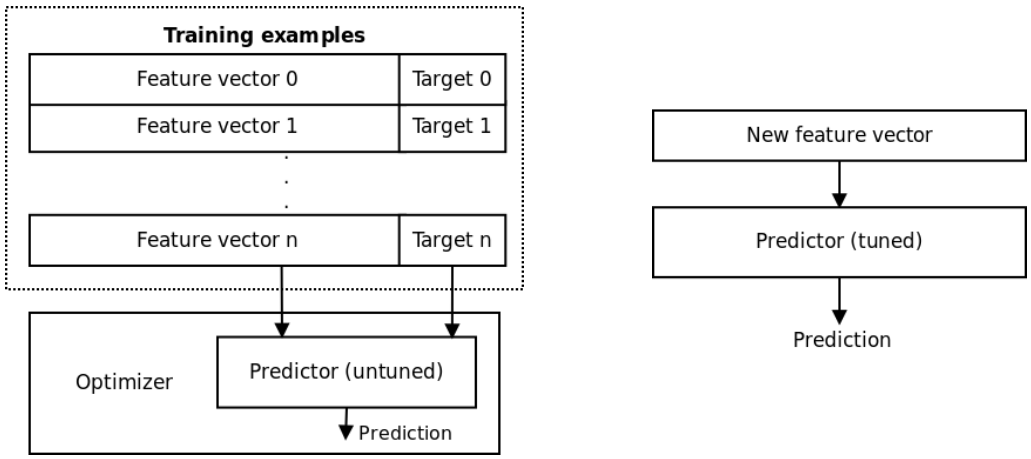
Supervised ML uses generic data models that are tuned based on previous experiences, otherwise known as *training examples*. These models treat the system under study as a black-box and are tuned using only experimental inputs and outputs. This is in contrast to analytical models which are constructed based on an understanding of the system. However, just like analytical models, the inputs and outputs consist of

numerical values. In ML terminology, the inputs to the model are called the *feature vector* and the output the *target*. The feature vector is a characterization of the type of object to be recognized. The target is the expected output for a given feature vector. Defining an appropriate feature vector and target is one of the main challenges in applying ML. Experience and understanding of the problem domain is required to be able to define these parameters and to use ML successfully.

There are two types of predictions depending on the ML model being used. Regression models predict continuous values whereas classification models predict discrete values, which is a class to which the new feature vector belongs to. Regression is useful, for example, if we want to predict the execution time of a program. Classification, just as was explained for clustering, could be useful for classifying types of programs. Being a supervised method, the difference to clustering is that the type of program for the training examples is known beforehand and we want to predict the type of program for new feature vectors.

Figure 6.2 shows how a supervised ML model is trained and tested. The objective of the training phase is to create a set of training examples and to use it to tune a ML model to perform predictions. Each learning example is composed of a feature vector and a target value. The task of the optimizer is to input each feature vector to the predictor and to compare the prediction with the corresponding target, which is the expected output. The optimizer will repeat this process while at the same time tuning the model coefficients until the prediction error is minimized. Once the model is tuned it can be used in a testing or prediction phase, where a new feature vector is input to the predictor and a prediction is output. Again, this prediction can be a real value in the case of regression or a discrete value for classification.

Having explained supervised learning is performed, we can now discuss two common supervised ML algorithms. Linear regression is the best known of the regression algorithms. The objective of linear regression can be stated as finding a line (or



(a) Training phase (b) Testing phase

Figure 4.1 : Training and testing phases for a supervised machine learning algorithm.

hyperplane in n-dimensions) that minimizes the Mean Squared Error (MSE) of the training examples. In other words, during the *training phase* the coefficients of the model are tuned using an optimizer to best align a line to the training examples. The model can now be evaluated during the *prediction phase* for a new feature vector to obtain a real value that lies within the line. In practice, some noise is added during the training phase so that the model does not overfit or is biased towards the training examples. A biased predictor in undesirable because it may not predict well for new feature vectors.

The Support Vector Machine (SVM) is a more advanced supervised ML technique because it can also account for non-linearity in the feature vector space. It was originally designed for classification but can be used for regression as well. In its most basic form it acts as a decision machine, that is, it outputs a single binary value which represents one of two classes. In an n-dimension feature vector space, let us consider a set of data that may be of two classes, 0 or 1. An SVM is a classifier, or a function that takes as input a feature vector and outputs a predicted class as shown in Formula 4.1.

$$\mathbb{R}^n \longmapsto \{0, 1\} \tag{4.1}$$

To do so, it determines the position of a new feature vector relatively to a hyperplane that separates one class of data from another. This hyperplane is known as the decision boundary. More precisely, the optimization algorithm tunes the coefficients of the decision boundary to maximize the margin or distance from the boundary to a selected group of points called the support vectors. In the case where the data is not linearly separable, the feature vectors can be projected to a different feature space. This is known as the *kernel trick* and there are certain functions, such as Gaussian, that can be used for this purpose.

Another important aspect about ML is how to evaluate a predictor's performance during the testing phase. In this work we utilize the concepts of precision and recall [26]. Precision is defined as the ratio of the number of times the prediction matched the target to the total number of targets (or predictions). Recall is defined as the ratio of the number of times the prediction for one of the outcomes matched the target over the total number of that particular outcome. For example, assume that we are predicting whether to vectorize a program or not. The precision would be the number of times *should vectorize* or *should not vectorize* were predicted correctly divided by the total number of targets. And the recall for *should vectorize* would be the number of times *should vectorize* was predicted correctly divided by the total number times *should vectorize* was a target. Recall would also be computed the same way for the *should not vectorize* outcome.

A naive testing approach would train a model with the training examples and test it with the same data, but this would of course produce biased results. For practical purposes we would like to evaluate whether the predictor is capable of predicting correctly for new inputs. Prior to the training phase, precision and recall evaluation is performed through a process called *n-fold cross-validation*. Cross-validation refers to partitioning the training examples into a training set and a test set, which are used respectively to train the model and test its precision and recall. When n-folds

are used, the training examples are partitioned into n chunks of data. Each chunk is used in progression as the test set, while the remaining *n - 1* chunks are are used as the training set. This process gives a better sense of the predictor performance than a single cross-validation because each point is evaluated in both training and testing phases.

The last topic to be discussed in this section is how ML can be used to drive a compiler optimization process, in what we term Machine Learning Driven Compiler Tuning (MLDCT). Figure 4.2 shows one possible configuration in which ML is performed as a module separate from the compiler. The block labeled *machine learning* contains one or more predictors that are used to inform the compiler as to which optimizations to apply and with which parameters, according to the feature vector extracted from the input source code. This we call the *optimization strategy*. The ML component will not take over the compiler entirely but will assist in the optimization process. Since the predictors treat the compiler as a black box, any internal transformations that the compiler performs will be captured during the training phase. The optimization strategy can be communicated to the compiler through command line flags, code annotations, and special programming interfaces provided by compilers such as GCC or LLVM.

An initial set of training examples is generated to train the predictor. This can be done using benchmark source codes which are representative of the types of programs that will be compiled during the testing phase. For each benchmark, a feature vector is extracted, the program is compiled, and its performance is measured. The predictors can then be trained off-line with the training examples for a particular compiler and target system. Thus when the predictors are trained, all the system complexity is hidden within tuning coefficients.

On the testing phase, the compiler transforms and outputs the executable based on the ML module predicted optimization strategy. When the program is executed,

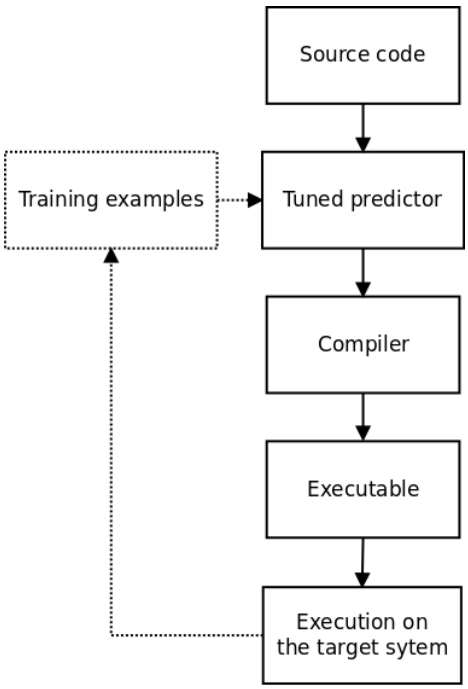


Figure 4.2 : Machine Learning Driven Compiler Tuning scheme.

the feature vector and target performance can be fed-back into the training examples. In this way, the machine learning module can perform continuous tuning of the models to predict for new types of programs. This last point, however, is an open research topic and was not considered as part of this work [16].

4.2 Single instruction, multiple data and vectorization

A vector instruction, or Single Instruction Multiple Data (SIMD), is a type of instruction that operates over fixed-size vectors of data of the same type. This is in contrast to the more commonly used sequential instruction paradigm, which in the context of vector instructions is called Single Instruction Single Data (SISD). SIMD instructions, just like SISD, can perform computation and data transfer operations.

Vector processors have existed for several decades but only recently have they started to become popular in consumer devices. The reason is that modern processors have been designed to operate on scalar data and to rely on instruction-level parallelism to increase their throughput. The result is increasingly more complex

architectures. Vector processor architectures take a different approach because they exploit data-level parallelism. Instead of executing multiple instructions in parallel, a vector processor can execute single instructions that operate on vectors of data. This simpler design can yield a much higher throughput in programs that have high data-level parallelism, such as scientific and multimedia applications which usually deal with matrix and streaming data. Some modern processor architectures, however, combine both aspects of parallel execution and parallel data processing. We now examine a generic vector processor architecture as described by Hennessy and Patterson [8].

Vector processors can be classified as memory-memory or vector-register. In memory-memory vector processors all operations are carried in memory while vector-register processors require operands to be loaded from memory into registers before carrying out an operation. Vector-register processors are more common and thus will be assumed in the rest of the discussion. The basic components of a vector processor are similar to a SISD processor, and include vector and scalar register files, the vector functional units, the vector load-store unit, and banked memory.

Figure 4.3 shows a vector addition operation as an example. There are two input vector registers and an output vector register. Registers in the vector register file are divided into sub-registers. Each sub-register can hold a vector element, and the total number of sub-registers is called the vectorization factor. The vectorization factor in Figure 4.3 is 4. The vector element data type may be fixed or variable, but every element in the sub-register must have the same data type. Element data types may include signed and unsigned integer, floating-point, and polynomial.

Vector functional units are similar to their scalar counterparts. There are functional units for arithmetic and logic, multiply and accumulate, and shift operations. The main difference to scalar processors is that besides providing vertical parallelism through pipelining, they also provide horizontal parallelism in the form of lanes. Each

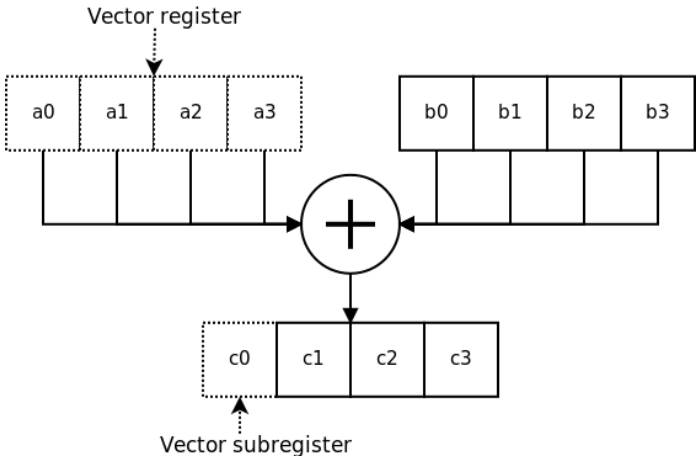


Figure 4.3 : Vector addition with vectorization factor of 4.

additional lane duplicates a functional unit pipeline to process one more vector element per clock cycle. Since vector operations presuppose that there are no data dependencies between vector elements, lanes can be added to increase the throughput without the risk of data hazards. Vector processors can be single-issue or may issue more than one instruction at a time.

The vector load-store unit handles transfers between memory and the vector register file. Memory is banked and the data is interleaved between banks to be able to retrieve multiple vector elements at the same time while reducing the memory access latency. Caches may or may not be present. To support access to data structures, memory may be accessed with a non-unit stride. A stride is the number of data units between vector elements. The structure elements may be de-interleaved during loads or interleaved during stores either by software or hardware. Software approaches load the stridden data into vector registers and use instructions to extract the elements to a different set of vector registers. Hardware approaches makes the process transparent to the programmer or compiler but still require more time than unit stride access.

Besides the stride, a second characteristic of memory access is data alignment. Similar to scalar processors, vectors need to be stored and loaded from memory addresses that are multiples of the vectorization factor. Some vector processors may

not allow unaligned memory access, and must load the adjacent aligned vectors and reassemble the desired vector in another register. The process of loading adjacent vectors from memory to extract the desired data and work around the unaligned access limitation is called unpacking, while the reverse operation to store an unaligned vector is called packing. Packing and unpacking introduce an overhead because additional instructions have to be added to the program for this purpose.

There are several overheads that can be avoided by using vectorization. A single vector instruction executes the equivalent of many sequential operations, reducing the number of fetched instructions. Replacing a loop by a vector instruction guarantees that there are no loop branch control hazards and no data hazards. In addition, the instructions to compare, update the counter variable, and branch in the original loop can be eliminated. Although no cache may be present, a single load or store instruction transfers a large amount of data from multiple memory banks, reducing the memory access latency. The loaded data has good spatial locality because most, if not all, of the data will be processed as a vector.

There are two ways in which programmers can employ vector instructions. The first is through the use of compiler intrinsics which resemble function calls but are directly translated by the compiler into a SIMD instruction. A second approach is to let the compiler transform sequential code into vector code, a process known as automatic vectorization. Vectorization is a well known optimization technique which, if used correctly, can speedup loops several-times fold. It can be applied to loops that have no loop-carried dependencies, such as the code in Listing 4.1.

```
float a[110], b[110], c[110];
for (int i = 0; i < 110; i++) {
    c[i] = a[i] + b[i];
}
```

Listing 4.1: Vectorizable code; no loop-carried dependencies.

In order to vectorize this loop to perform four additions in parallel as shown in Figure 4.3 , the first step is to apply a code transformation called loop unrolling. In loop unrolling, the loop's body is replicated by an unroll factor and the index variable's upper boundary and increment are adjusted to have an equivalent computation. Modern compilers can perform this procedure automatically. As a preparation for vectorization, the loop is unrolled the same number of times as the vectorization factor (VF) as shown in Listing 4.2.

```
float a[110], b[110], c[110];
for (int i = 0; i < 27; i += 4) {
    c[i+0] = a[i+0] + b[i+0];
    c[i+1] = a[i+1] + b[i+1];
    c[i+2] = a[i+2] + b[i+2];
    c[i+3] = a[i+3] + b[i+3];
}
for (int i = 108; i < 110; i++) {
    c[i] = a[i] + b[i];
}
```

Listing 4.2: Unrolled loop for a vectorization factor of 4.

Note that the upper boundary for the index variable is also divided by the VF and is incremented by VF. Since the iteration range is not a multiple of the VF, a second

loop had to be created to perform the remaining operations. This transformation of creating a new loop to perform the first or last iterations of the original loop is called loop peeling. Now the body of the first loop can be vectorized by substituting these SISD operations by a single SIMD operation, called *add4* in Listing 4.3. However, the remaining peeled loop remains as vectorization overhead.

```
float a[110], b[110], c[110];
for (int i = 0; i < 27; i += 4) {
    add4(c, a, b);
}
for (int i = 108; i < 110; i++) {
    c[i] = a[i] + b[i];
}
```

Listing 4.3: Vectorized loop.

The reason vectorization can yield high speedups, as explained in this section, is that operations can be performed in parallel up to the VF and more importantly, memory accesses can be done more efficiently [9]. At the same time, underutilizing the vector registers and performing non-aligned memory access can make the vectorized code perform worse than sequential code. Compiler support is one of the main obstacles to taking advantage of the potential of vector processors. However, modern production compilers have limited support for automatic vectorization [10]. The reason is that compilers currently utilize heuristics and hand-built machine models to estimate optimization profitability which do not capture well enough the complexity of modern systems (in the case of vectorization, see for example the work of Trifunovic et al. [11]). In this work we study a more advanced technique based in machine learning that can aid the compiler in choosing the correct optimization. In conclusion, we selected vectorization as the candidate optimization since it has been

used for decades to dramatically reduce the execution time through data parallelization and more efficient memory access patterns.

4.2.1 Methodology

In this experiment we focus on predicting whether to apply automatic-basic-block vectorization, referred to as ABBV in the rest of this section [12]. ABBV consists in leveraging the inherent instruction-level parallelism (ILP) inside basic blocks in order to generate SIMD instructions. ABBV however, is part of the vectorization process previously explained, and software pipelining which is not considered in this experiment [29]. Automatic vectorization is equivalent to loop unrolling followed by ABBV. 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 by unrolling loops. Still, sometimes no benefit can be obtained by using vector instructions and it may be more advantageous to only use scalar instructions. This may happen because vector instructions in modern processors often involve some overheads like data packing and unpacking or very slow access to unaligned data in the memory subsystems. Hence, vectorizing compilers should be able to: (1) not vectorize if it is not profitable, that is, if the execution time of a program compiled with ABBV is higher than the same without it, and (2) transform the code ahead of the backend in order to feed the best possible input to the ABBV optimizer. We refer to these properties as **P1** and **P2** in the remainder of this section.

However, modern compilers are far from optimal with respect to both P1 and P2. Let us consider Intel's ICC compiler, which is well known for the high quality of its output. As much as 44% of the programs vectorized by Intel Compiler using only ABBV are slower than without vector instructions. This blatantly shows its failure

with respect to P1. Moreover, works from various teams at University of Illinois, USA [10], or INRIA, France [19], have shown that it is also weak with respect to P2 even on simple examples[a] . In this experiment, we focus on one code transformation: loop unrolling. We have measured that if we do not assist Intel Compiler into unrolling the program upstream, it fails to generate profitable vectors for 90% of our benchmarks. This occurs despite the fact that there exists an unroll factor that allows profitable vectorization for 45% of it.

We propose to determine ahead of the backend whether or not a program can be profitably vectorized downstream by the ABBV optimizer, the Intel Compiler in our case. To do so, we utilize the Support Vector Machine (SVM) learning algorithm. We carry out two experiments that differ in their inputs and that address P1 and P2 respectively. First, we consider a program *after* being transformed in order to reproduce the situation encountered in the backend of a vectorizing compiler, when it decides whether or not to vectorize. Second, we consider the program *before* being transformed, as a middle-end would do before deciding which code transformation to apply. In the remaining of this section we will often refer to these experiments simply as **Exp1** and **Exp2**, respectively. We were able to obtain an accuracy of approximatively 70% for both, which is far above the current capabilities of the Intel Compiler. Our success not only comes from the power of SVM, but also from the software characteristics we consider in order to describe the input programs.

In this experiment we further graphically assess the quality of our model by means of a graph called the *learning curve*. Two examples of learning curves are given in Figure 4.8 . It shows the evolution of the accuracy of the model for a growing number of example data, for several different choices of training and test sets. We train the SVM with the training set and compute the accuracy for both

[a] These works consider all kinds of vectorization, not only ABBV

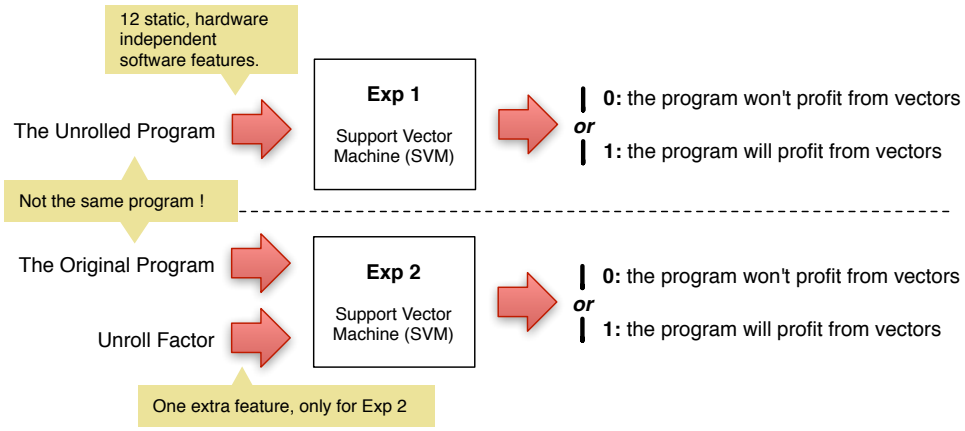


Figure 4.4 : The inputs and outputs of the SVM used in both Exp1 and Exp2. They only differ by one extra input feature in the case of Exp2: the unroll factor to apply on the program.

the training and the test sets. For each horizontal value, we consider 10 randomly generated sets. The lines show the average of the accuracies related to both sets in order to show their respective tendencies. The line is expected to raise for the test set as the size of the datasets increases: this is because the SVM is more trained and therefore smarter. On the other hand, the accuracy for the training set should drop if the SVM does not overfit the data. Finally, both lines are expected to converge toward the same value: this is the *asymptotic accuracy* of our model.

We leverage SVM in order to determine the profitability of vectorization when only considering ABBV. We define that the output *0* corresponds to a program for which vectors are not profitable, and *1* otherwise. We consider two situations for which we want to determine the profitability of vectorization. The inputs and outputs of both experiments are shown in Figure 4.4 . In both cases, the output of the SVM is the same; they however differ from their inputs. Exp1 only accepts as feature the transformed input program, while Exp2 also accepts the parameters of the transformation to apply. In this experiment we only consider loop unrolling to transform the program, hence the later boils down to the unroll factor.

In this experiment we select a set of 12 static, hardware-independent features to characterize the input programs. There are several ways to classify the types of software characteristics. First, they may be measured statically from sources, or dynamically at runtime. While the second makes it possible to gather far more information, it requires to actually execute the program: this is not something we can afford inside a compiler for obvious time-related constraints. Software characteristics 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 is not available on another one, for example because of differences in micro-architecture. For this reason we favor the latter, and our results in the next section show that this is enough. In conclusion, we consider only *static*, *hardware independent* software characteristics.

The characteristics were selected empirically, while observing the behavior of Intel Compiler with our benchmark. In the end we have retained 12 of them; 6 of which are extracted at AST level, and 6 at IR level. These characteristics constitute features of the SVM, together with the unroll factor for Exp2 and are detailed in Table 4.1 . When more than one array is accessed in the innermost loop, AST2 and AST3 are measured 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.

In order to further assess the amount of redundancy in our set of features, we have run principal component analysis on the set of known data and the results showed that the amount of variance is loosely concentrated, and that it requires half of the

Table 4.1 : The selected software characteristics for predicting vectorization profitability

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 is 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	$\{0, 1\}$	Do 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 SSA variables used in the innermost loop

components to express 80% of it. This indicates that the information is relatively well spread into our original set of features. In other words, our features are relevant in helping the SVM to carry out accurate predictions.

Compiler optimizations are more efficient when applied to programs' hotspots, that is, innermost loop nests. That is why we consider a benchmark suite made of simple loops, representative of how an innermost calculation loop may look like. The selected benchmark is called Test Suite for vectorizing Compilers (TSVC), in its version provided by Maleki et al. [10]. It is composed of 151 simple loops and has been devised to assess the quality of compilers for vectorizing loops. In our work however, we do not consider loop vectorization; instead, we expect to mimic this transformation by expanding it into (1) loop unrolling and (2) ABBV. That is why we consider not only the 151 loops that constitute TSVC, but also the same with the innermost nest unrolled with a factor ranging from 2 to 20. The resulting benchmark counts 3020 programs (151×20).

Table 4.2 : The options fed to Intel Compilers. We use SSE3 as supported by our test machine (Intel Core2Duo Extreme Merom@2.66GHz).

With ABBV	-std=c99 -O3 -fno-alias -opt-report 3 -V -vec -vec-report5 -xSSE3
Without ABBV	-std=c99 -O3 -fno-alias -no-vec

The programs are compiled into executables using Intel Compiler with and without ABBV, using the compilation options shown in Table 4.2 . For Intel Compiler not to unroll, vectorize or pipeline loops, we annotate the innermost loops with the corresponding pragmas: `nounroll`, `novector` and `noswp`. The host machine is an Intel Core2Duo (Merom) at 2.66GHz, that supports up to SSE3 instruction set extension.

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 is detailed in Figure 4.5 . It consists of 3 steps: (1) we prepare, compile and execute our benchmark with and without ABBV; (2) we determine the profitability of vectorization (3) we measure the software characteristics for the benchmark. The flow relies on three tools: an innermost loop unroller, a feature extractor, and the vendor compiler. For the first, we use a tool called PIPS[b] . For the third, we use the Intel Compiler. For the second, we measure by means of custom tools the characteristics of programs at two abstraction levels: LLVM's intermediate representation and Clang's abstract syntax trees[c] . We refer to both as **IR** and **AST** respectively.

After preparing the data, we use the SVM as explained in section 4.2.1: (1) we divide the examples dataset into the training set and the test set (respectively 80% and 20% of the initial set); (2) we train the SVM with the training set; (3) we assess the quality of our model by computing the cross-validated accuracy and by plotting

[b] Online: http://pips4u.org/

[c] Online: respectively http://www.llvm.org and http://clang.llvm.org

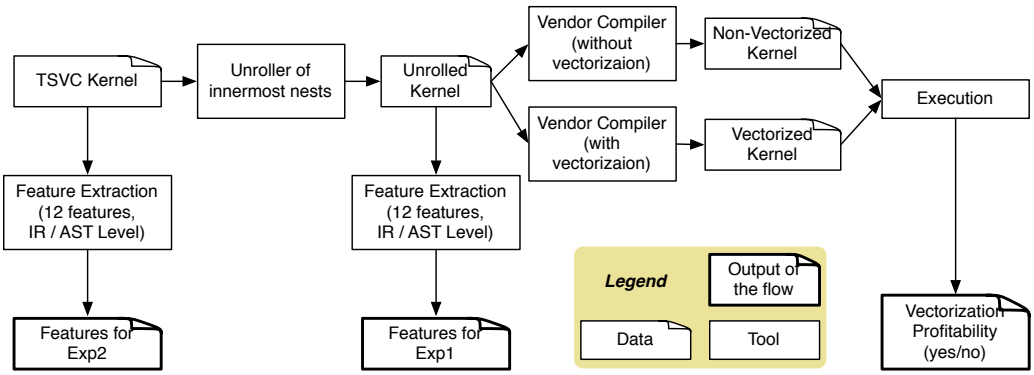


Figure 4.5 : Experimental flow for building ther training and test sets.

the learning curve. To implement these experiments, we use libsvm[d] , a stable and free library for SVM.

We compare our results against two baselines. The first one is Intel Compiler alone, as explained below. The second one is the exact same experiments, but using nearest neighbor (NN) instead of SVM. NN is a simple machine learning technique that considers for prediction the vectorization profitability of the closest training data with respect to the Euclidean distance in the space of features.

We compare against Intel compiler alone because it is considered by the community as one of the most capable vendor compilers. With respect to P1, it has a success rate of 56%. This number is the ratio of profitable vectorization among the examples dataset; we consider all the kernels and all the unroll factors, and Intel Compiler has unrolling, loop pipelining and loop vectorization off.

With respect to P2, it is impossible to reproduce Exp2 with Intel Compiler. Instead, we consider its success ratio to profitably generate SIMD instructions when it is possible. We consider the 151 non-unrolled programs, and we activate unrolling in

[d] Online: http://www.csie.ntu.edu.tw/ cjlin/libsvm/

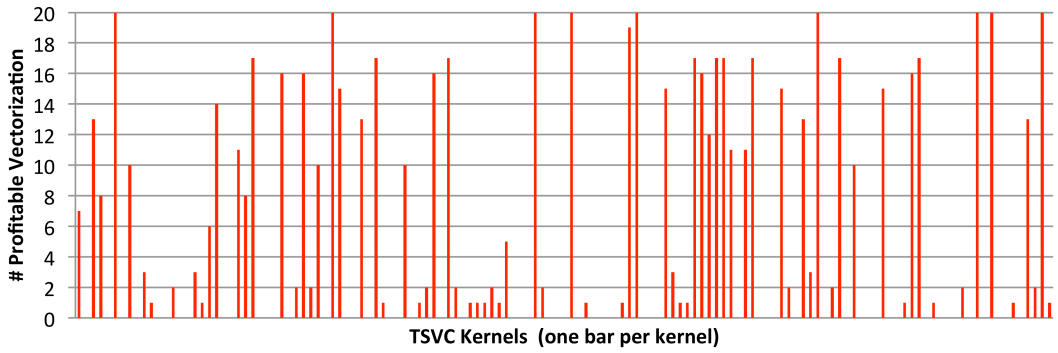


Figure 4.6 : Number of times the vectorization was profitable for each program.

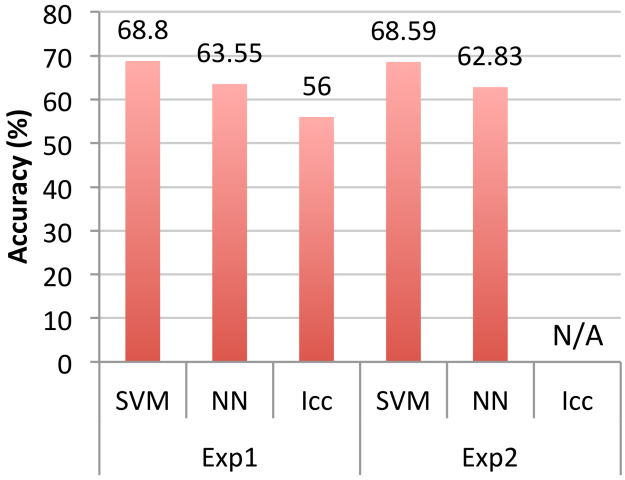


Figure 4.7 : Prediction accuracy.

Intel Compiler (by removing the `nounroll` pragmas). By doing so, we ask Intel Compiler to Figure out by itself a correct transformation to apply (including unrolling) in order to generate profitable vectors. It succeeds for 13 programs. However, as explained in the next section, it is possible to find an unroll factor that enables profitable vectorization for 63 programs. Therefore, its success rate is 20.63% (13/63).

4.2.2 Results

First of all, let us take a look at Figure 4.6 . It plots the number of times that vectorization was profitable for each benchmark. This graph confirms that our data are not skewed. Further analysis showed that the ratio of the execution time

without vectors and the same with vectors ranges from 0.46 to 39.25. In other words, ABBV may provide up to 39.25 time speedups, but failing to recognize non-profitable ABBV may slow down the compiled program up to 2.2 times ($0.46^{-1} = 2.2$). The latter occurs for the program *s116* with an unroll factor of 2.

We plot the accuracy of our method against the baselines in Figure 4.7 . We use cross-validation as explained in Section 4.1. The prediction accuracy for Exp1 by using SVM is almost 70%. This is significantly better than the 56% of Intel Compiler (labeled *icc* in the plot). In particular, the SVM manages to correctly decide against vectorizing for the program *s116* with an unroll factor of 2, thereby providing a speedup of 2.2. Moreover, we achieve a similar, high precision accuracy in the case of Exp2. This is remarkable because we solve in Exp2 a problem more complex than in Exp1. No number is given for Intel Compiler because it is not possible to carry this experiment on it. Still, we can compute the success of SVM at finding at least one profitable vectorization for each program when possible: it is far above Intel Compiler at 73.01% (against 20.63%).

Figure 4.7 further plots the prediction accuracy using NN: 63.55% and 62.83% respectively. These numbers are under those of SVM, but very close. This is coherent with the results published by Stephenson et al. [28]. Moreover, NN has the advantage over SVM of being a simpler algorithm. Therefore we believe there may be situations where it is preferable over SVM. Still, SVM achieves more than 5% better accuracy and it is the most successful technique in our experiments.

Figure 4.8 plots the learning curves for both Exp1 and Exp2 (from left to right). The profile of the curve is ideal, as the accuracy for the testing and the training data are actually converging towards 70.5% and 68.5%, respectively. The maximum average accuracies measured for the test sets are respectively 68% and

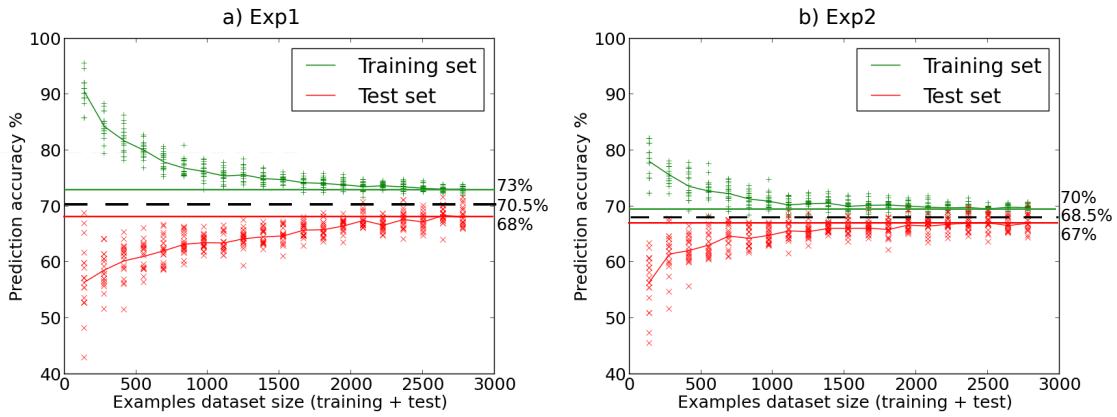


Figure 4.8 : Learning curves for Exp1 and Exp2 (resp. a and b, from left to right).

67%[e] . These learning curves illustrate the convergence of our model and its high accuracy, that is, its high quality. In other words, we have shown that SVM can be successfully used to predict the profitability of vectorization on TSVC, and can be useful to address both P1 and P2.

In conclusion, we use SVM to predict the profitability of automatic basic-block vectorization for Intel Compiler on TSVC, a benchmark made of 151 simple yet representative loops, unrolled by a factor ranging from 1 to 20. We achieved prediction accuracy of about 70%, even before actually unrolling the loops. This technique may be useful in compilers in two ways. First, it can enable the compiler to avoid generating vectors that actually slow down the program; this often happens with the Intel Compiler (44% of our benchmarks). Second, it can allow the compiler to better predict the existence of profitable unroll factors in the middle-end, as Intel Compiler fails to do in 79.37% of the programs of our benchmark. This is important because depending on the unroll factor, vectorization may provide up to 39 times speedup according to our measurements (benchmark *vpvts* with an unroll factor of 20).

[e] These accuracy numbers are different (yet very close) from the ones computed using cross-validation, because the method to obtain them is different.

We however see several limitations in our approach. First, SVM require fine tuning to be accurate. Second, the choice of the set of software characteristics is critical and depends on the problem we are trying to solve as well as the benchmark. Still, our results show that machine learning makes it possible to significantly improve the quality of the code generated by Intel Compiler: we have measured speedups up to 2.2 times by successfully preventing it to carry out unprofitable vectorization.

4.3 Improving prediction performance

In the next experiment we investigate how to increase the accuracy of the vectorization profitability predictor from the first experiment. With this objective, a new benchmark was selected and two new sets of software characteristics were developed.

4.3.1 Methodology

The dataset of this experiment consists of Tensor Contraction (TC) kernels. A TC is a generalized matrix multiplication that is heavily used in chemistry suites such as NWChem[f] for computing electronic configurations. They consist of k perfectly nested loop with a single multiplication-accumulation statement inside the innermost loop, and an example is illustrated in Figure 4.9 . This innermost statement features three memory loads and one memory store that may result in irregular memory access patterns: this makes such kernels challenging to optimize. There is no specific limitation in the depth of tensor contraction kernels, but we only consider the ones of depth 4 to maintain a reasonable experimentation time. Still, this is a common loop depth in programs that feature tensor contraction calculations. We refer to these kernels with the abbreviation TC4. TC4 kernels operate on 3 matrices A, B and C, two of them being of dimension 3, and one of dimension 2. Each dimension has the

[f] http://www.nwchem-sw.org

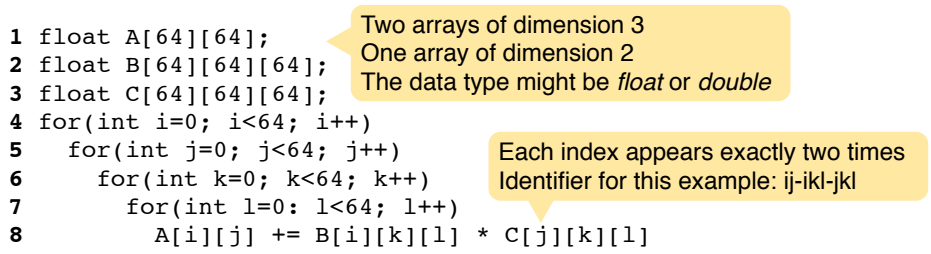


Figure 4.9 : Example of tensor contraction benchmark of depth 4 for $N = 64$. The workload identifier (wid) of this example is **ij-ikl-jkl**

same size, which we call N. In the case of TC4, the innermost statement is executed N^4 times. An example of such kernel for $N = 64$ is given in Figure 4.9 .

One can generate many TC4 kernels by varying: (1) the dimensions of A,B and C, 3 or 2; (2) the order of the array indices at line 8, providing that each induction variable appears exactly 2 times, at most once per access; (3) N, the size of each dimension of the arrays. For the kernel of Figure 4.9 those parameters as set as follows: A, B and C are of dimensions 2, 3 and 3 respectively; the arrays are accessed with the indices i,j for A, i,k,l for B and j,k,l for C in this order; $N = 64$.

It is possible to fully characterize a TC4 kernel with a short description string with the following format: AAA-BBB-CCC, where AAA, BBB and CCC are the indices used to access arrays A, B and C respectively. We call the string the workload id, or *wid* for short. For example, the wid of the kernel in Figure 4.9 is ij-ikl-jkl. We always consider that the induction variables corresponding to the loop nests are i,j,k and l in this order. Therefore, the wid characterizes the source code of a kernel, independently from the size of the arrays.

We have measured that Intel Compiler version 12.1.5 applies vectorization differently depending on the alignment of the memory accesses with the SIMD width[g] .

[g] This only applies when the dimension size of the input arrays is known statically, which is the case in our experiments.

Indeed, in the SSE3 instruction set, the SIMD arithmetic instructions only operate from SIMD registers, which width is 128 bits or 4 single-precision-floating-point scalars. One has to load and store data between memory and these registers using SIMD load and store operations. SIMD load/store are more efficient when the locations of scalars in memory are consecutive and aligned with 128 bits. In this case it is possible to move the 4 single-precision-floating-point scalars at once using an aligned packed instruction. When locations are consecutive but not aligned, it is still possible to use a single different non-aligned packed instructions, but its execution is however far slower. In other cases, one has to load/store scalars independently and pack/unpack them into the SIMD registers using dedicated instructions. This is the slowest situation that we want to avoid as much as possible. However, TC kernels feature at least one such case along one of the four loop indexes i,j,k,l.

Figure 4.10 shows the speedup of the TC kernels forcibly vectorized, compared to when compiled with vectorization forced off. As expected, the fastest performance are achieved for N multiple of 4, and the slowest when N is an odd number. For $N = 4k+2$ Intel Compiler generates two versions of the inner loop body, one vectorized and one scalar, and uses a if statement to execute the appropriate one. In the vectorized loop body, it uses exclusively aligned SIMD load/store. We further notice that the TC kernels can be divided into two groups of wid that always yield similar performance for any N. The ownership of a wid to a given category is decided by the ratio of SIMD to serial load/store operations. We were however not able to determine the category of the wid without actually compiling the TC kernels. Intuitively, this is because it is hard to predict the behavior of the compiler.

In fact, most of the TC kernels that benefit from our technique introduced in this section fall in one of following two categories: the slow wid group and $N = 4k$, both wid group and $N = 4k + 2$.

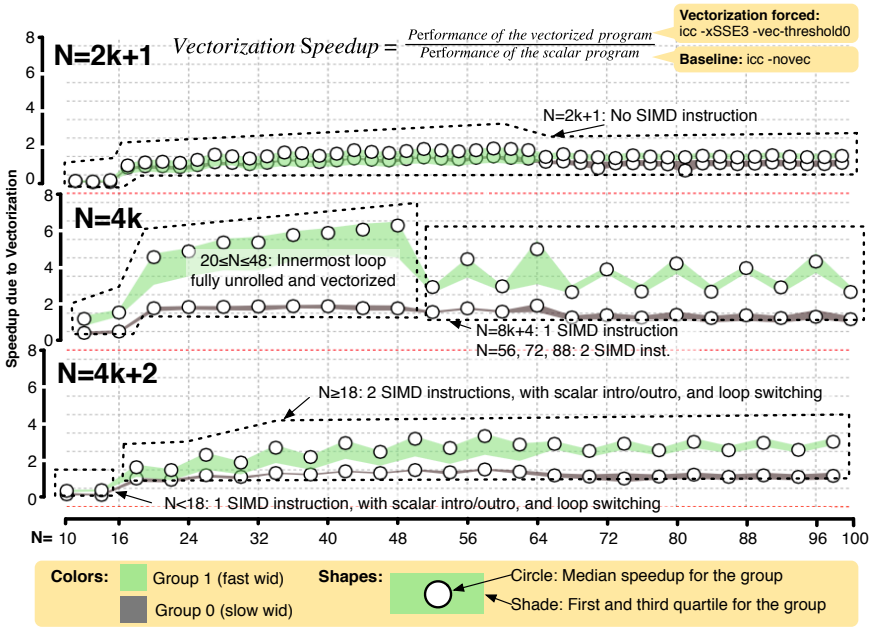


Figure 4.10 : Performance of the TC kernels when compiled with default options with Intel Compiler version 12.1.5. N stands for the size of the input array's dimensions. We show separately the numbers depending on the value of N. We further manually divide the wid into two groups with similar performances.

In order to be integrated before the compiler, we only consider software characteristics that are measured statically, that is, without actually executing, and before compiling. We consider the three sets below from the related work or as baseline:

Milepost. It is representative of the methods that rely on information on the control flow graph or the static instruction count including, which also include the software characteristics from Park et al. [4].

Assembly. This corresponds to the static instruction count of the vectorized assembly, as proposed by Stock et al. [5]. Intuitively, predicting from this level is less challenging than from software characteristics measured before optimization; indeed the machine learning device does not need to predict the behavior of the compiler anymore.

Random. Additionally, we consider a set of software characteristics made of 3 random numbers; it constitutes an important baseline for two reasons. First, if a given

set of software characteristics yields a similar accuracy as random, it means that this software characteristics does not characterize the TC4 kernel. Second, if random yields high accuracy, it means that our experimental setup is flawed.

Next, we discuss the our two software characteristics sets designed for the tensor contraction kernels to capture their dynamic behavior. **Proposal1.** For our first set of software characteristics, *proposal1*, we propose to feed the predictor with important properties of the memory access pattern of TC4: the sequentiality of memory accesses, the re-use of data, and whether or not the memory accesses are sequential along the innermost loop nest (index l). It consists of three integer numbers per memory access, as detailed in Figure 4.11 , that is, 9 in total. We consider the load and store from/to A as a whole to avoid repetition; therefore we need 9 numbers to describe one TC kernel. This is detailed in Figure 4.11 *a* for an access x to a 2-dimension array. On the bottom is also given the exact value of the this software characteristics for the TC4 kernel of Figure 4.9 , with the wid *ij-ikl-jkl*.

Proposal2. Our second set of software characteristics, *proposal2*, starts from the assumption that the compiler processes the kernels that exhibit similar properties modulo nest interchange in a similar fashion. There is no need to explicit these properties, we only need to teach the machine learning device which parts of the array accesses can be transformed from one to the other by means of loop nest interchange. To do we so define *proposal2* so that it describes the repetition pattern in the wid. For instance, the first number is 1 if and only if the rightmost index of the access to A is the same as the leftmost index of the access to B, and it is 0 otherwise. An example is given in Figure 4.11 *b*.

We propose a method to decide whether to force automatic vectorization on behalf of Intel Compiler and the compiler user, using the setup in Figure 4.12 . Here thee classifier takes as input the set of software characteristics detailed previously

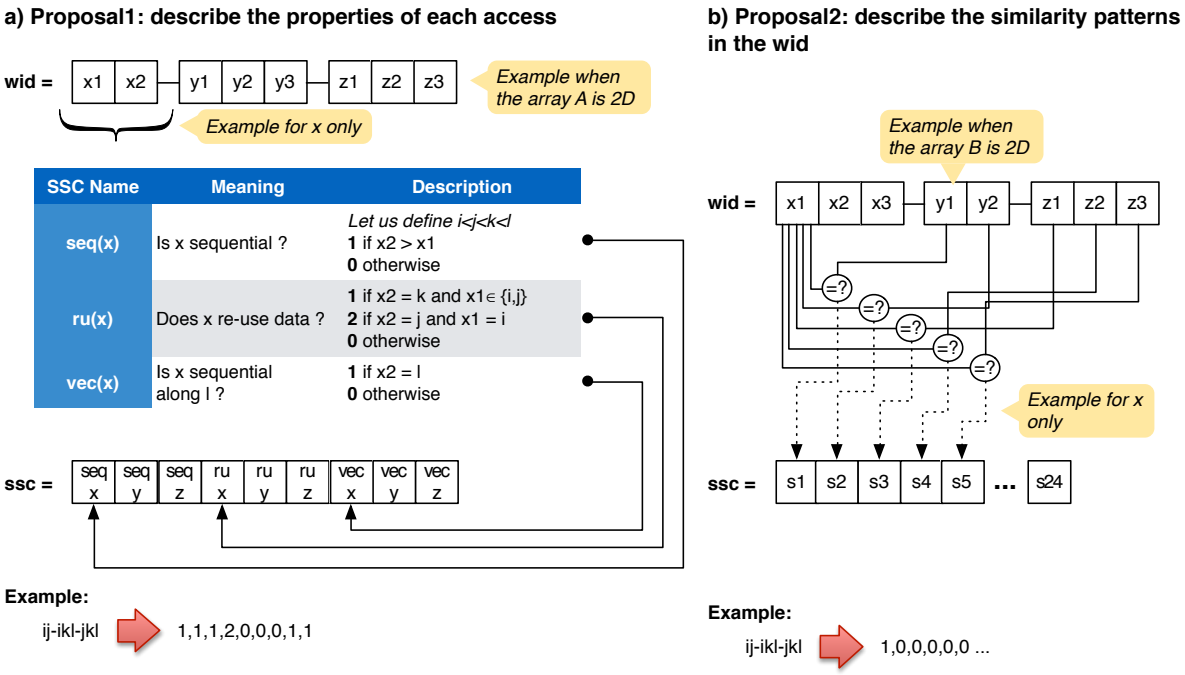


Figure 4.11 : The description of the two sets of SSCs we propose in this article.

as well as the size of the array dimension, and decides the option of the compiler depending on the predicted class.

To this end, we set up an arbitrary threshold of 5% in order to define the notion of *significant* speedup difference. With this definition, we can divide the data set into 3 classes: *class 0* when the vectorization speedup is more than 5% lower for vectorization forced than non forced; *class 2* when it is more than 5% larger; and *class 1* in between. Ideally, we want to predict each class so that we set the option *-vec-threshold0* for elements of class 2, and not for the ones of class 0. The prediction does not matter for class 1 (the option does not have any effect on performance in this case).

We used a code generator to vary TC4 parameters at random, and generated 1500 TC4 kernels to be used as the benchmark for this experiment. We vary N, the dimension of array dimensions, from 10 to 100. This represents a total of 136500

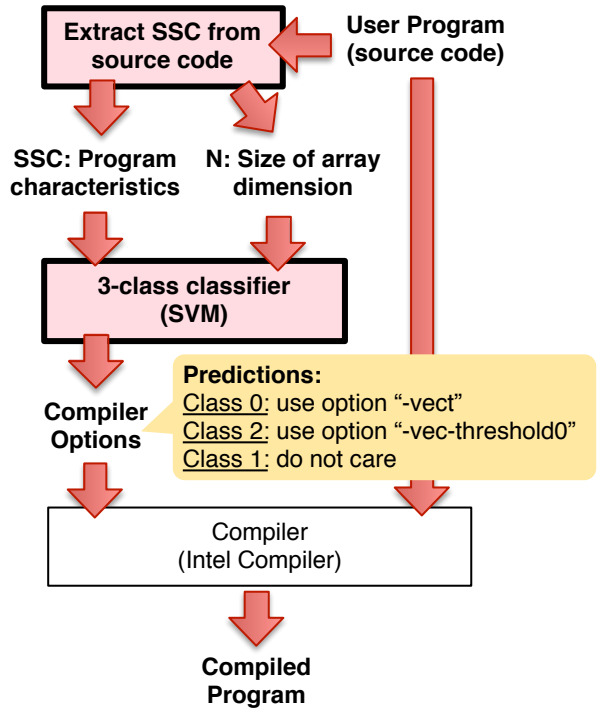


Figure 4.12 : Classifier for the second experiment on predicting vectorization profitability.

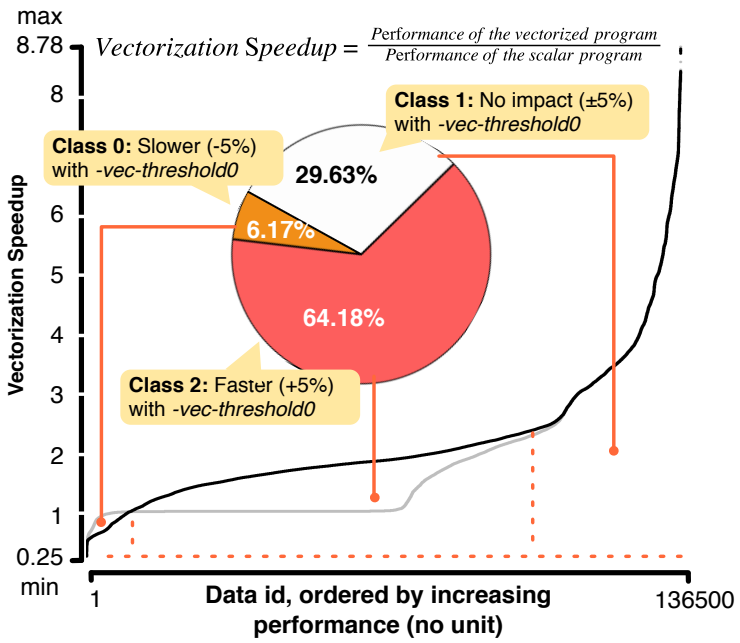


Figure 4.13 : Dataset overview, annotated with each class.

points (1500 kernels times 91 sizes). We focus on single-precision-floating-point calculation (`float` type in C). Figure 4.13 shows for this dataset the speedup compared to the program compiled without SIMD instructions with the command line:

```
icc -novec
```

The grey line corresponds to the default behavior of the compiler:

```
icc -vect
```

And the black line is when we force vectorization forced:

```
icc -vec-threshold0
```

The respective position of the lines tells us which option is most profitable performance-wise. When the grey line is above the black one, this means that the default behavior of the compiler is conservative and optimal (class 0). On the other hand, when the black line is above, this means that the compiler is missing a vectorization opportunity (class 2). The TC kernel for which both lines are approximately at the same position are in class 1. We can see that upon applying automatic vectorization, Intel Compiler takes the correct decision in 29.63% of the cases. It is too conservative for 64.18% of our data. By failing to vectorize, it misses speedups up to 2.5 times. However, it is not a good idea to always force vectorization for 6.17% of the data: this might slow down the compiled program by up to 4 times.

The grey line is under 1 for less than 1% of our dataset. It corresponds to the situation where the compiler is actually too aggressive, and vectorizes although it is not profitable. We do not focus on this situation in this article.

Our test machine is an Intel Core2 Extreme based on the Merom architecture. It features 4 processing cores, we however only use one. The processor supports the SSE3 SIMD instruction set, that is, 128-bit vectors; this corresponds to 4 float scalars. This is the same vector instruction set as Intel Silvermont, the state-of-the art micro-architecture from Intel for embedded systems. We compile using Intel Compiler version 12.1.5, using the default optimization level. The command line is:

`icc -xSSE3 -O2`

In particular, Intel Compiler carries out automatic vectorization at this level. The documentation of Intel Compiler suggests to use `-O3` for computation intensive programs; our measurements (not detailed here) however show that `-O2` is by far the best performing options in the specific case of TC4 kernels, never being the worse. When specified, we force vectorization by means of an extra option, *-vec-threshold0*; the command line becomes:

`icc -xSSE3 -vec-threshold0`

With this option, Intel Compiler applies vectorization even if it predicts the probability for it to be profitable to be 0%. Each data point is the resulting geometric mean of 5 measurements; data are discarded and re-measured if the standard error of these five measurements is larger than one fifth of the arithmetic mean, that is, if the coefficient of variation is larger than 0.2. We do not use other compilers such as GCC or LLVM because they do not provide such options.

As for the machine learning technique, we use Support Vector Machine (SVM) with a Gaussian Kernel as provided by the R package e1070 [1]. We have observed that the accuracy does not significantly change with the predictor, therefore we only show our results with SVM. This situation is coherent with the current common wisdom in the field of machine learning [30]. All the accuracy numbers are calculated by means of a 3-fold cross validation procedure on the wid. In particular, we make sure that all the data taken on kernels with the same wid do not span across both the training and test sets. This is very important because this reproduces the situation where the compiler is faced with a never-encountered program. Therefore, our predictions are the ones a compiler user would obtain in real situations.

4.3.2 Results

For the data set shown in Figure 4.13 we predict the class by means of SVM for the sets of software characteristics and experimental setup explained previously.

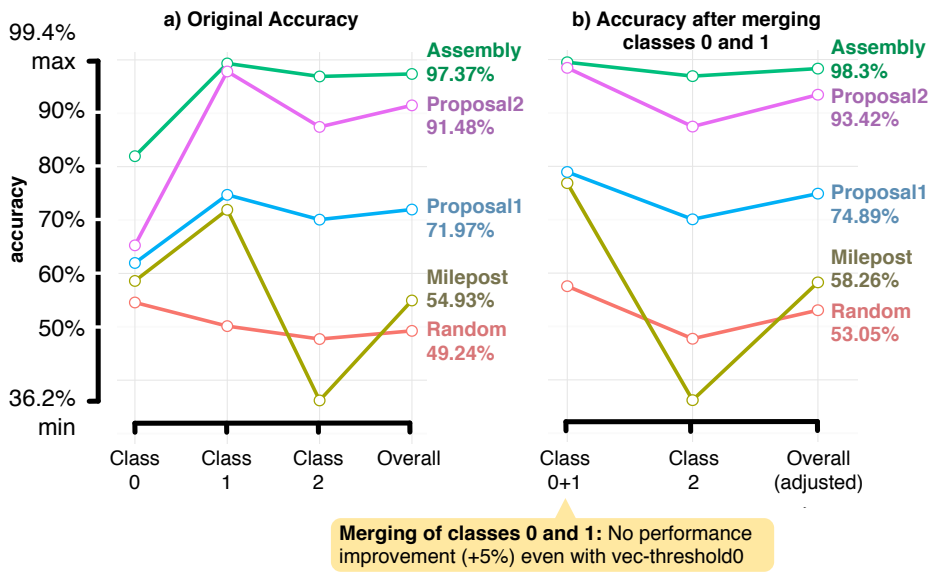


Figure 4.14 : The prediction accuracy, overall and for each class and set of software characteristics.

Figure 4.14 a shows the prediction accuracies for each class and each set of software characteristics. First, *random* yields low accuracy overall and for each class as expected, around 50%. Then, our two proposed sets of software characteristics, *proposal1* and *proposal2*, exhibit significantly higher accuracy; this means that they indeed provide useful information for our predictions. On the other hand, *milepost* yields notably low accuracy, close to *random* overall. Finally, *assembly* yields the highest accuracies for each individual class as well as in general. This is because the performances are largely decided by the vector instructions generated by compiler for memory load/store. This result is coherent with the results obtained by Stock et al. [5]. Still, *proposal2* is almost as accurate as *assembly*: 91.48% and 97.37% respectively. However, *proposal1* does not achieve as high accuracy as we had expected. Observing the generated assembly files, the reason is that Intel Compiler carries out loop interchange internally, changing the value of the properties measured by this software characteristics set.

We notice that the accuracy for class 0 is invariably significantly lower that for the other classes. After investigation, we have determined that most of the miss-predictions for class 0 are actually predicted as class 1. This is not a concern with respect to our final target which is to decide if whether or not the option *vec-threshold0* should be set. Indeed, it is enough to decide not to set the option for kernels predicted classes 0 and 1, and to set it for class 2. These accuracies corresponds to Figure 4.14 *b* in which classes 0 and 1 are merged into *class 0+1*. The new class now yields very high accuracy, especially for *proposal2* and *assembly*: 98.45% and 99.5% respectively. The overall accuracy is also slightly up for all the software characteristics.

Next, we look at the effect of applying our predictor in terms of program performance. We use the results as follows: given the predictor of Figure 4.12 , we compile with the command line

```
icc -xSSE3 file.c
```

if it predicts class 0 or 1, or

```
icc -xSSE3 -vec-threshold0 file.c
```

if it predicts class 2.

With our method, we expect to get speedup for class 2 and to avoid speed-down for class 0, while having no effect on class 1. The distribution of the speedups due to our method for the whole dataset is shown in Figure 4.15 . We only show the results for kernels originally in classes 0 and 2, that is, 50.63% of the data (see Figure 4.13). As a reference, we also display the results for the perfect predictor, as it sets the upper limit for the other sets of software characteristics.

The ranking of median speedups approximately matches the ones obtained for overall accuracy. The best sets of software characteristics are *assembly* and *proposal2*: both yield 1.69 time speedups. This is very close to the theoretical maximal of 1.7, shown by the target. On the other hand, *milepost* performs as bad as *random*; this can be explained by its dramatically low accuracy for class 2, as shown in Figure

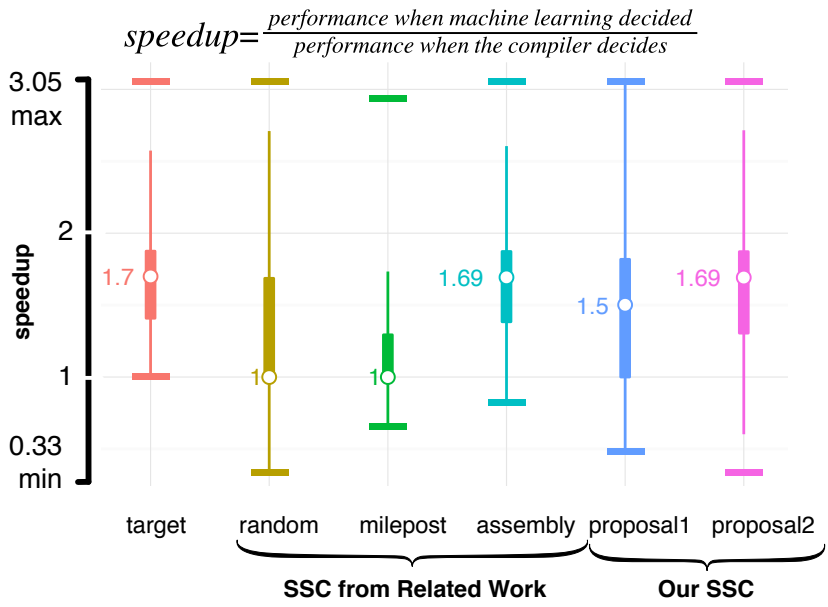


Figure 4.15 : The distribution of the speedups yield by our predictions, for kernels in classes 0 and 2. Target is given as reference, and designates the imaginary predictor with a perfect accuracy.

4.14 (36.2%). Finally, *proposal1* stands in the middle. In conclusion, we can say that *proposal2* provides the best set of software characteristics. Not only it is the best suitable for compilation, but it also yields very high speedup figures that are close to the maximum.

Yet, the worse case for *proposal2* is a dramatic 3-time slowdown. It is reached for the wid *li-lkj-kij-d4-n12*, and corresponds to a data predicted as class 2 although it is in class 0. It is a singularity: other values of N for the same wid achieve profitable vectorization. There are exactly 9 such miss-predictions in the whole data set for *proposal2* (0.006% of the dataset). A close study reveals that for those kernels, Intel Compiler fully unrolls the three innermost loops without interchange, with scalar loads and stores. This results in more than 10000 lines of assembly, mostly `movss` and `unpack` instructions. The SVM is not able to predict such behavior of the compiler from *proposal1* and *proposal2*. On the other hand, *assembly* being extracted from the vectorized assembly, succeeds in reporting the singularity to the SVM predictor.

Chapter 5
Predicting Vectorization for Energy Reduction

When it comes to compiler optimizations, speedup is usually the first optimization objective that comes to mind. However, as the integration level of electronic circuits continued its exponential growth, power reduction has also become an important objective. In recent years, data centers and the proliferation of mobile devices have also been drivers for reducing energy consumption in computing systems. Although hardware engineers are already well acquainted with design techniques for low power consumption, software power reduction is still a vastly unexplored topic particularly in the area of compilation. This is despite the fact that, ultimately, software is the main responsible of making efficient use of hardware resources.

This chapter presents an application of our vectorization profitability predictor of Chapter 4 for tuning the compiler with respect to energy reduction, and analyzes the potential of vectorization as an energy reduction technique. Experiments were conducted on a system-on-chip device featuring the popular ARM Cortex-A8 processor architecture.

5.1 Power consumption in the ARM architecture

ARM has become the most widely used embedded processor architecture [13]. There are seven versions of the ARM architecture; the first three are now obsolete and the latest one is called ARMv7. The ARM architecture is described in the Architecture Reference Manual, sometimes referred to as the ARM ARM. The latest version ARM is divided into three profiles; application (ARMv7-A), real-time (ARMv7-R),

and microcontroller (ARMv7-M). Thus the ARM manual is now divided into the ARMv7-AR Architecture Reference Manual and the ARMv7-M Architecture Reference Manual. Each profile is implemented as a family of cores called Cortex and are described in their respective Technical Reference Manual (TRM) [a] .

The purpose of dividing the architecture into profiles is for better accommodating it to different markets. ARMv7-A is intended for high-performance applications such as tablets, smartphones, and mobile gaming consoles that use complex operating systems and multimedia. ARMv7-R and ARMv7-M are intended for deeply embedded industrial applications and low-cost electronics where a simple operating system or no operating system is used. ARMv7-M can be considered a subset of ARMv7-R and the main difference is that the real-time profile operates at a higher frequency while the microcontroller profile is designed for fast interrupt processing.

Figure 5.1 shows a simplified block diagram of the ARM Cortex-A8, the microprocessor architecture of many System-on-Chip (SoC) devices. The main feature that characterizes an ARM as a RISC architecture is a load/store architecture, fixed-width instructions that usually execute in one cycle, and simple addressing modes. The register file is 32-bit wide and 16 words in depth. It implements a Harvard memory model where the data and instructions are stored in different memory caches. There are two Arithmetic Logic Units (ALU) and one Multiply And Accumulate (MAC) pipelines for parallel integer operations. The memory model consists of a single, flat address space of 2^{32} bytes and may have up to seven levels of cache. It can be addressed as bytes, 16-bit half-words, or 32-bit words with limited support for doublewords. Instruction access must be aligned and data access can be unaligned. In addition, data can be managed as both little and big endian.

[a] http://infocenter.arm.com/help/index.jsp

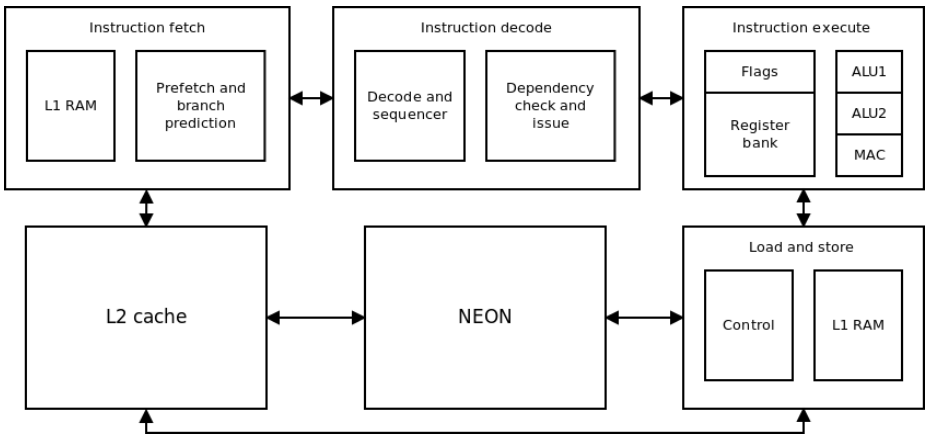


Figure 5.1 : The ARM Cortex-A8 architecture.

The ARMv7 architecture specifies five instruction sets (IS). Some cores support all of them while others only one. The regular ARM IS consists of 32-bit instructions. The Thumb IS provides 16-bit instructions with a subset of the functionality of the ARM IS. It allows for a trade-off between code density and performance since Thumb instructions can be used to emulate ARM instructions but sometimes at the expense of more execution cycles. The Thumb-2 IS extends Thumb with 32-bit instructions. It provides better code density than Thumb but with performance and functionality similar to ARM. Using the Unified Assembler Language (UAL) both ARM and Thumb-2 can be assembled from a single assembly source. The architecture also supports the execution of Java bytecode through the Jazelle IS. Its successor is called ThumbEE (Execution Environment) and provides hardware acceleration to dynamically generated code.

ARMv7 provides three extensions which are not required in the implementation. The first extension is called Vector Floating-Point (VFP) and adds IEEE 754 single- and double-precision floating-point support. The next two extensions add SIMD support (Section 4.2). The Advanced SIMD (NEON) extension can handle integer and single-precision floating-point vector operations and is included in the ARM

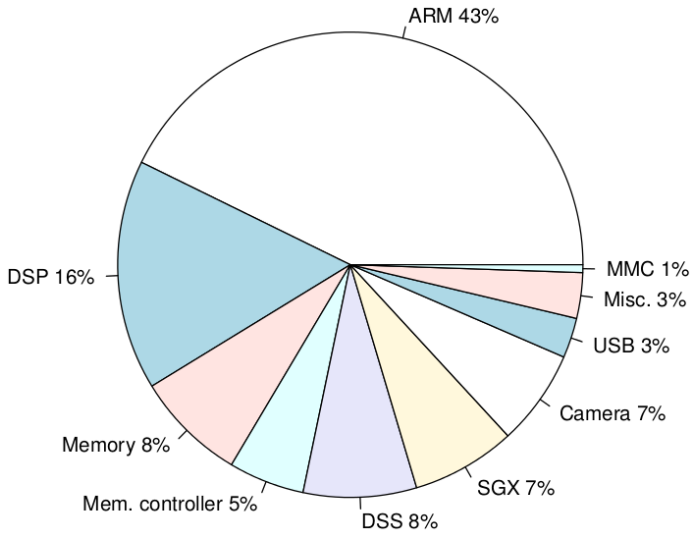


Figure 5.2 : Distribution of the estimated power consumption on the OMAP3530 [14] [15].

Cortex-A8, as shown in Figure 5.1 . The DSP extension adds a subset of this functionality to the ARMv7-M profile.

Texas Instruments' OMAP3530 SoC makes for an appropriate case study for this work because it includes an ARM Cortex-A8 core with the NEON SIMD instruction processing module, along with other subsystems typical of the SoCs that are used in mobile devices. Figure 5.2 and Table 5.1 show the power consumption distribution per subsystem. These values were obtained from the maximum estimated values of the OMAP3530 power estimation spreadsheet [14] [15].

The three components with highest power consumption are the ARM core (43%), the DSP core (16%) and the memory and memory controllers (13%). Thus, code optimizations should focus on making efficient use of these three subsystems. The consumption of the display and storage devices are also be significant, but optimization strategies for the utilization of these devices are beyond the scope of this work.

Module	Power (mW)	Description
ARM	548.64	ARM Cortex-A8 with NEON @500MHz
DSP	205.29	Digital Signal Processor
Memory	99.20	DDR2 SDRAM @162MHz [15]
Memory controller	67.66	SDMA (System Direct Memory Access) and SDRC (SDRAM) controllers @166 MHz
DSS	100.95	Display Sub-System
SGX	93.0	2D/3D Graphics Accelerator Engine
Camera	85.46	Camera and Image Signal Processor
USB	35.71	Universal Serial Bus
Miscellaneous	40.72	General purpose timers, general purpose I/O, I2C, SPI, and 1-wire
MMC	6.89	Multi Media Card controller

Table 5.1 : Estimated power consumption on the OMAP3530 [14] [15].

5.2 Methodology

Optimizing software to reduce the energy consumption began by identifying the ARM components that consume the most power (Figure 5.2 and Table 5.1). Next, we selected vectorization as a potential optimization because it can dramatically reduce the execution time and allows more efficient memory access patterns through the use of specialized SIMD hardware, as explained in Section 4.3.

The ARM architecture was the obvious choice for this project, as it is the most widely used architecture in mobile devices. Specifically, we selected the ARM Cortex-A8 which is the most common implementation in SoC. Another reason is that it includes a NEON co-processor for executing SIMD instructions which has a vectorization factor of 4. The Beagleboard is one of the few development boards that includes a Cortex-A8 core within its SoC. Thus we selected the Beagleboard-xM single-board computer which has an DM3730 SoC [b] . The ARM core within this SoC is clocked at 1GHz and the board has 512 MB of DDR RAM. An embedded version of Ubuntu Linux 12.10 was installed to facilitate experimentation.

[b] http://beagleboard.org/beagleboard-xm

The development board was instrumented as shown in Figure 5.3 . The system was designed to automatically compile, execute, measure the execution time and the average power values for a vectorized and nonvectorized version of each TSVC benchmark program (Section 4.2.1). Operating system installation was done from the workstation through the UART connection. Subsequent interactions with the board were done through SSH over the Ethernet connection. Interaction with the DS1052E oscilloscope was performed solely from the Beagleboard, and the workstation served only to setup the board and to retrieve all the collected data at the end of the experiment.

Instrumentation code was automatically inserted into the programs to count the execution time and to trigger the oscilloscope to record the average power consumption. The programs were compiled within the board using the GCC compiler. Two probes from the oscilloscope were connected to the J2 jumper on the board to measure the voltage drop over a 0.1Ω resistor. For each execution, 600 voltage samples were taken at the maximum oscilloscope frequency of 50MHz and transfered from the oscilloscope to the Beagleboard for voltage-to-power conversion. Data transfer and post-processing was performed in the Beagleboard rather than the workstation for convenience, but this did not have any impact in the experiment because this was carried after experimentation. This process was repeated 3 times and the values were averaged to obtain an average power consumption and average execution time. With the execution time and average power, the energy consumption was computed for the vectorized and unvectorized versions of each program.

We selected the GCC compiler because it supports the ARM architecture and NEON instructions, is open source, stable and widely used within production environments. The feature vectors for our benchmark were extracted from the high-level C source codes (Section 4.2.1). This required the creation of a static profiler to extract

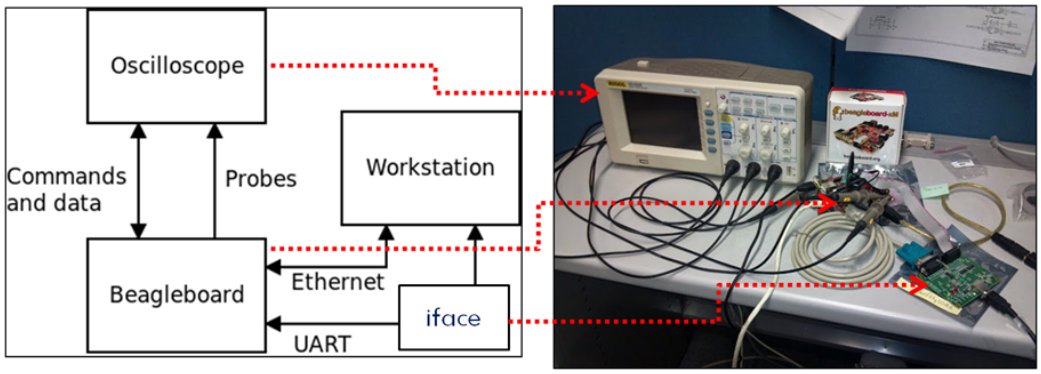


Figure 5.3 : Energy measurement setup.

the software features. The LLVM compiler was used because it includes libraries for this purpose.

Table 5.2 lists the compiler options used in GCC version 4.7.2 to activate and deactivate vectorization. In both modes, the optimization level 2 (O2) was used. This allowed the compiler to perform other kinds of transformations other than vectorization. This is analogous to a real-use case, where the ML component assists the compiler in optimizing the input program (Section 4.1). However, we deactivated loop unrolling with the *fno-unroll-loops* compiler option because we were already unrolling the benchmarks manually.

Mode	Compiler command line options
Vectorize	tree-vectorize tree-slp-vectorize lax-vector-conversions fivopts
No vect.	fno-tree-vectorize no-tree-slp-vectorize
Both	O2 std=c99 mfpu=neon funsafe-math-operations fno-unroll-loops no-modulo-sched

Table 5.2 : GCC command line options for vectorization.

We used the same benchmark, Test Suite for Vectorizing Compilers (TSVC) as explained in Section 4.2.1. Because the predictor performance tends to improve with the number of training examples, we artificially increased the number of examples by unrolling the original 151 loops by factors from 1 to 20 for a total of 3020 benchmarks.

Listing 5.1 shows an example of a TSVC loop called *s431*. In practice, each loop is surrounded by an outermost loop to control the number of times it will execute. This is needed to be able to measure a long enough execution time and enough power samples such that the execution environment noise is reduced. The same software characteristics were used, as for the first experiments in predicting vectorization to reduce execution time (Section 4.2.1).

```
int k1 = 1; int k2 = 2; int k = 2*k1-k2; int nl;
for(int i = 0; i <= 31999; i += 1)
    a[i] = a[i+k]+b[i];
```

Listing 5.1: Sample TSVC loop.

Vectorization profitability was again modeled as a binary classification problem. In other words, we wanted for the predictor to answer whether vectorizing the input program would be beneficial or detrimental. Figure 5.4 shows the contents of the *machine learning* component of Figure 4.2 as implemented in this work. The inputs to the predictor were the Software Characteristics (SC) of the program to be optimized and the output was one if vectorization was predicted to reduce energy consumption, and zero otherwise.

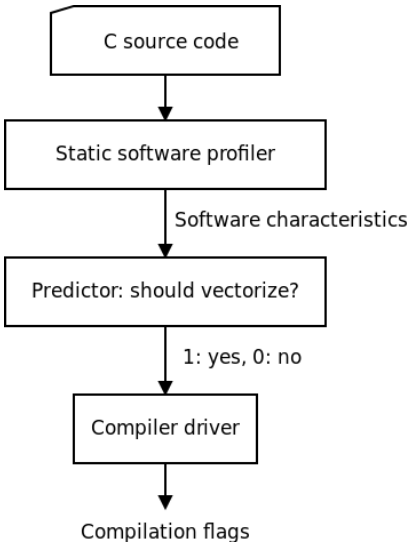


Figure 5.4 : Energy profitability prediction scheme.

Support Vector Machine (SVM) was selected for the predictor because it can account for the possibility of nonlinearilty in the relation between the SC and vectorization profitability. We used the *svm* function of the R language package e1071 implementation of SVM with the default training parameters and the Radial Basis Function (RBF) kernel[c] .

To train the predictor, the first step was to generate a training example for each of the benchmark programs. A training example consisted of an SC as feature vector and a target of one if vectorization reduced the energy after execution, or zero if it increased. That is, during the training phase the predictor was presented with many cases of a characterization of the input program and whether vectorization turned out to be profitable.

After training, we had a dataset in which each record specified the program identifier, execution time, power and energy ratios, the SC and whether vectorization increased the energy or not. The time, power, and energy values were expressed as ratios

[c] http://cran.r-project.org/web/packages/e1071/index.html

of the measurement when the program was vectorized divided by the measurement when vectorization was disabled. Thus ratios less than one indicated profitability due to vectorization and ratios greater than one indicated detrimental performance. These data were enough to test the predictor, to assess the impact of vectorization on energy, and to assess how well the predictor could exploit vectorization for reducing energy consumption.

During the testing phase, 5-fold cross validation was performed over the training examples to obtain a prediction for each example in the set. This fold value was taken as a rule-of-thumb, since typical values are 5 or 10. Now each example had both the target or the real outcome and a predicted outcome. With this information, the quality of the predictor was assessed by computing the precision and recall as defined in Section 4.1.

5.3 Results

In this chapter we answer the following two main questions of this experiment in predicting for reducing energy reduction. First, is vectorization a suitable optimization technique for reducing energy consumption in embedded systems? And second, is machine learning a suitable technique for predicting vectorization for energy reduction?

We answer the first question by examining the collected time, power, and energy measurements after compiling the TSVC benchmark with GCC. We then examine representative programs in order to assess the effect of vectorization on energy consumption. We answer the second question by training a predictor and analyzing its effectiveness at reducing energy consumption by making judicious use of vectorization.

It is important to note that both questions are answered using different toolsets. The vectorization impact is analyzed from the point of view of statistics, and accordingly, no data manipulation is performed. The vectorization profitability predictor on

the other hand, is trained and analyzed using standard machine learning techniques. As such, the objective is to train a practical predictor that is likely to suggest efficient vectorization while preventing detrimental application of it.

Figure 5.5 shows histograms for the performance distributions of the loop-unrolled TSVC benchmark for the GCC compiler in terms of time, power, and energy ratios. In this figure, the horizontal axis marks the measured ratios and the vertical axis shows, in logarithmic scale, the number of programs that fell within a 0.05 ratio interval. Henceforth, performance ratios are expressed as the measurement for a particular metric when vectorization is activated divided by the same metric when vectorization is deactivated. Therefore, ratios below unity were considered to be a beneficial application of vectorization (region marked ①), above unity were detrimental (region ③), and when equal or very close to unity, the compiler was unable to transform the code to vectorize it and the performance was neutral (region ②). Table 5.3 lists the ranges selected for each region. The ranges do not cover the whole performance interval, but rather capture the peak and nearby points that characterize the performance region.

Region	Id	Range
Beneficial	1	0.1 to 0.3
Neutral	2	0.9 to 1.1
Detrimental	3	1.4 to 1.8
Power peak	4	0.6 to 0.65

Table 5.3 : Performance ratio intervals for the regions annotated in Figure 5.5 .

The first noticeable feature in the histograms is that most measurements are concentrated around unity, or the neutral region ②. In the energy histogram there are 1834 programs in the neutral region from a total of 2753 programs, or 67%. In fact, a logarithmic scale was applied to compress this peak and to show other interesting patterns in the performance profile. The reason is that the GCC compiler is unable to

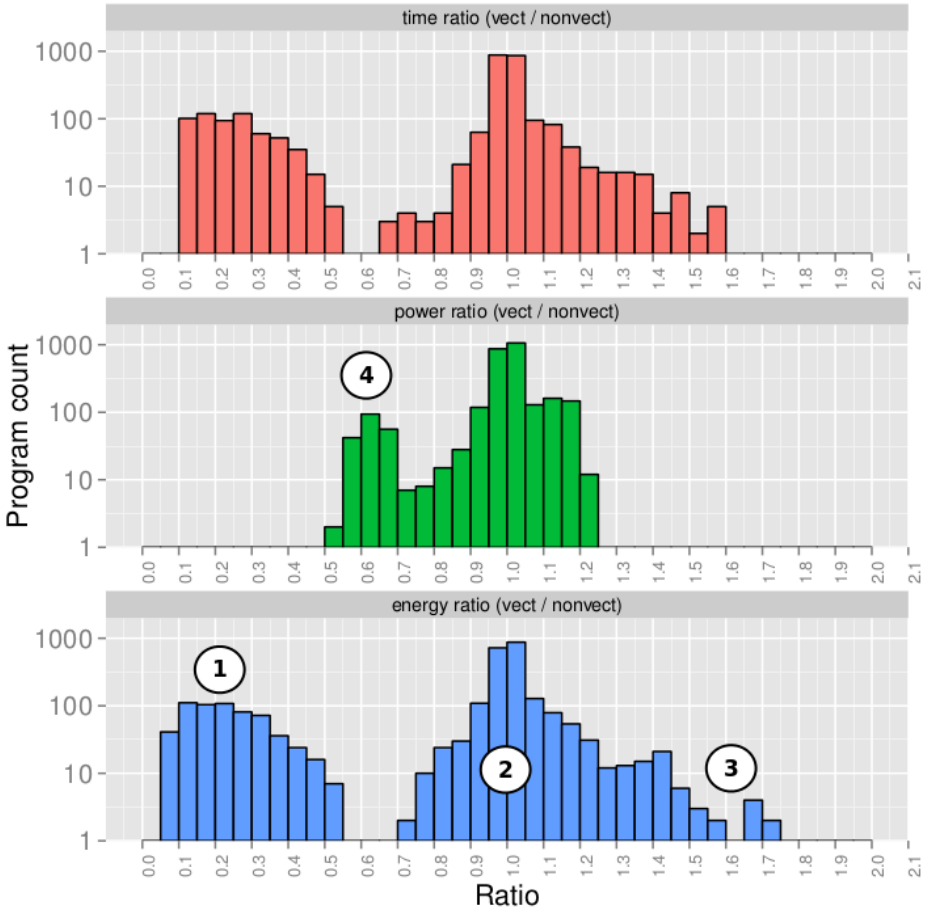


Figure 5.5 : Performance histograms for TSVC compiled with GCC.

transform the code to use vector instructions for most of our programs, even though every loop in TSVC is vectorizable if the right code transformation were applied. These results confirm the research by Maleki et al. [10] which found that 45% to 75% of TSVC failed to be properly vectorized by the tested GNU GCC, Intel ICC, and IBM XLC compilers. The tendency is shown in both the time and power histograms, which support this conclusion using two independent measurements. Programs that were not vectorized by the compiler were of no interest to this study, which aimed at assessing the potential of vectorization and not evaluating the capacity of the compiler at vectorizing programs.

Nevertheless, it is clear that the dominating factors in energy consumption were the changes in time rather than in power. This is confirmed by the similar behavior shown in both the energy and time histograms. Also, most measurements are located at and below unity, indicating that vectorization tends not no be detrimental to the performance. The programs also seem to cluster around 0.25 for both time and energy ratios. If we invert this time ratio, what we have is 4 times speedup. This value corresponds to the NEON's vectorization factor, that is, vector instruction can load, compute, and store 4 elements at once. Therefore, it makes sense that programs that benefit from vectorization group around this speedup value. Finally, another noticeable feature is a peak in the power ratio distribution at region ④ of Figure 5.5 which will be explained later on in this section.

Now, let us examine representative programs for each of the regions in the histograms to confirm the observations made above and to quantify the impact of vectorization in performance. As a reference, Tables 5.4 and 5.5 show a description of the ARM assembly instructions used in the sample programs to be discussed.

To understand the effect of activating automatic vectorization in the compiler, it is essential to study the assembly of a program before and after vectorization for each of the performance regions. Since it is not possible to analyze the thousands of

Inst.	Description
ldr	Load register
str	Store register
add	Add two scalars
fld	Load floating-point register
fst	Store floating-point register
fmac	Floating-point multiply-and-accumulate
fcpys	Copy floating-point register

Table 5.4 : Selected SISD (Single Instruction Single Data) instructions for the ARM core and VFP co-processor.

Inst.	Description
vldr	Load vector
vstr	Store vector
vadd	Addition of two vectors
vmla	Vectorial multiply-and-accumulate
vuzp	De-interleave two vectors

Table 5.5 : Selected SIMD instructions for the NEON vector co-processor.

programs in the dataset, samples were taken at random and examined to see if their assembly could explain the performance exhibited in the region they were extracted from. Table 5.6 summarizes the time, power, and energy ratios for the sample programs to be presented next.

Table 5.6 : Vectorization performance impact per representative program. The metrics are shown as ratios. UF stands for loop unroll factor.

Program	UF	Region	Time	Power	Energy
s235	17	1	0.26	0.65	0.17
s235	20	2	1.13	0.96	1.09
s235	4	3	1.82	0.94	1.72
s1115	20	3	1.44	1.10	1.58
s235	5	4	0.32	0.61	0.19
s000	2	4	0.11	0.62	0.07
s000	13	4	0.28	0.60	0.17
s000	16	4	0.28	0.62	0.17
vpvts	10	4	0.15	0.61	0.09

As a first approach at analyzing the assembly, it was useful to find programs that fell within all performance regions according to the unroll factor. This made it

easier to identify which modifications in the code contributed to their performance, in preparation to analyzing other sample programs. In our benchmarks, only the programs named *s231* and *s235* have loop unrolled instances that fall within regions ①, ②, and ③. That is, depending on how many times the innermost loop of these programs is unrolled, they will respond differently to vectorization in a beneficial, neutral, or detrimental way. The program to be studied is named *s235*, and the baseline C source code without loop unrolling is shown in Listing 5.2.

```
for(int i = 0; i <= 255; i += 1) {
    a[i] += b[i]*c[i];
    for(int j = 1; j <= 255; j += 1)
        aa[j][i] = aa[j-1][i]+bb[j][i]*a[i];
}
```

Listing 5.2: C source code for the TSVC loop *s235*.

When the innermost loop for program *s235* was unrolled 17 times, vectorization was dramatically beneficial across all metrics and the performance was within region ①. The innermost loop contained an assembly instruction pattern similar to the Listing 5.3, which was repeated 17 times. The impact on performance was a reduction of 74% in execution time, 35% in average power, and 83% in energy consumption. Vectorization had a big impact because memory access could be done efficiently without the need of packing or unpacking instructions (Section 4.2). In addition, although there were Single Instruction Single Data (SISD or serial) additions, the expensive multiply-and-accumulate instructions were performed with the SIMD version. This sample also shows that vectorization not only can reduce the execution time, but also the average power consumption.

```
add         r1, lr, r3
ldr         sl, [sp, #132]
vmla.f32    q9, q8, q10
vstmia      r5, {d18-d19}
vldmia      r2, {d20-d21}
ldr         r5, [sp, #140]
add         lr, r6, r3
add         r7, sl, r3
adds        r2, r5, r3
ldr         sl, [sp, #124]
```

Listing 5.3: Assembly pattern for *s235* unrolled 17 times.

However, when *s235* was unrolled 20 times, just 3 times more than in the previous example, the performance was located within the neutral region ②. Listing 5.4 contains a similar pattern than when unrolling 17 times, except that SISD rather than SIMD instructions were generated. The different unroll factor caused GCC's automatic vectorization algorithm to fail half-way in applying the optimization, even though extra overhead instructions were generated outside and within the innermost loop. Performance remained close to the unvectorized baseline, with an increase in execution time of 13%, a reduction in power of only 4% and an increase in energy of 9%.

```
add     r2, r0, #16384
add     r3, r5, #15360
flds    s13, [r2, #0]
flds    s14, [r3, #0]
fmacs   s14, s15, s13
add     r3, r0, #17408
flds    s12, [r3, #0]
add     r3, r0, #18432
flds    s13, [r3, #0]
add     r3, r0, #19456
flds    s17, [r3, #0]
add     r3, r0, #21504
flds    s0, [r3, #0]
```

Listing 5.4: Assembly pattern for *s235* unrolled 20 times.

In the third case, program *s235* was unrolled 4 times and as a result the performance was located within the detrimental region ③. The intuition was that this was due to unprofitable vectorization but once more, GCC was unable to vectorize the code. An assembly pattern was generated similar to the last case when the program was unrolled 20 times. However, this time the side effect of a failed vectorization was that the performance dropped significantly to 82% higher execution time, 5% less average power, and 72% increase in energy consumption. Inspecting the assembly did not reveal any distinguishing feature between the unvectorized baseline assembly; the instruction mixes and control flow graphs were similar. Determining how loop unrolling and vectorization interact in the compilation flow is beyond the scope of this work, but still this information is useful because it shows that not all performance

loss in our experimental dataset was due to vectorization but to other modifications introduced as part of the compilation process.

Still, it was important to find cases in which vectorization was detrimental. Otherwise, if the instruction overheads produced by GCC on a failed vectorization were to be solved, then the appropriate vectorization strategy to reduce energy consumption would be to always vectorize because the performance was either beneficial or neutral. But plenty of cases were found in which vectorization was the cause of poor performance, and here we examine program *s1115*. When this program was unrolled 20 times and vectorized, its performance was located in the detrimental region ③ with an increase of 44% in execution time, 10% in average power, and 60% in energy consumption. Listing 5.5 shows the instruction pattern that was found in the innermost loop. The program had low performance because SIMD instructions were used to load from memory while addition was done sequentially. Specifically, the *vuzp* instruction was used to de-interleave, or swap the upper two elements of the first vector register with the lower two elements of the second vector register. Although the purpose of this instruction was to load structured objects from memory, this instruction was being used here as a vector unpacking operation. One of the swapped registers was then used for memory addressing with the *vldr* instruction, and the retrieved value was added with a SISD instruction. In other words, SIMD instructions were purposelessly used load data for a SISD operation.

```
vuzp.32   q8, q2
vuzp.32   q6, q4
vld1.32   {q2}, [r3]
add       r3, fp, #336
vld1.31   {q4}, [r6]
add       r6, fp, #352
```

Listing 5.5: Assembly pattern for *s1115* unrolled 20 times.

The next cases studied explain the second peak that can be observed at region ④ of the power ratio in Figure 5.5 . This region is interesting because in the execution time and energy ratios there is no other noticeable peak other than in the neutral region ②. In region ④ there are 94 training example programs, 92 of which also belong to the beneficial region ① or perform even better. There were programs with all unroll factors, therefore this did not seem to be a factor of much influence in their good performance. Five programs were selected from this region for further study. The *s235* program with unroll factor 5 was selected for analysis because this program had already been studied for other unroll factors. The other four programs were selected at random and all the performance results were also included in Table 5.6 .

Program *s235* with unroll factor 5 was successfully vectorized and had a similar instruction mix and performance as when unrolled 17 times. Listing 5.6 shows the C source for the baseline *s000* program without unrolling.

```
for(int i = 0; i <= 31999; i += 1)
   X[i] = Y[i]+1;
```

Listing 5.6: C source code for the TSVC loop *s000*.

When *s000* was unrolled 2 and 13 times, the generated assembly code had about half SIMD and half SISD instructions. There were however no packing instructions present and this should account for the good performance. For an unroll factor 16, however, the generated assembly resembled that of s235 with unrolling factor of 17 where there was almost complete vectorization. The difference was that the *vadd* instruction was used in place of *vmla*, because the innermost statement is adding rather than multiplying.

The last case studied was for the program *vpvts* unrolled 10 times. Listing 5.7 shows the C source for the baseline *vpvts* program without loop unrolling. The instruction mix of the vectorized assembly was similar to program *s235* unrolled 17 times, except there was a much greater number of *vldr* and *vstr* instructions. Still, the performance improvement was even better than for *s235* because most of the *ldr* and *str* in the unvectorized version of the program were replaced by their SIMD counterpart. This highlights the importance of efficient memory access through SIMD instructions. Performing vectorial memory accesses can significantly improve or hurt performance, beyond the impact of the computational parallelization aspect of SIMD instructions.

```
for(i = 0; i <= 31999; i += 1)
    a[i] += b[i]*s;
```

Listing 5.7: C source code for the TSVC loop *vpvts*.

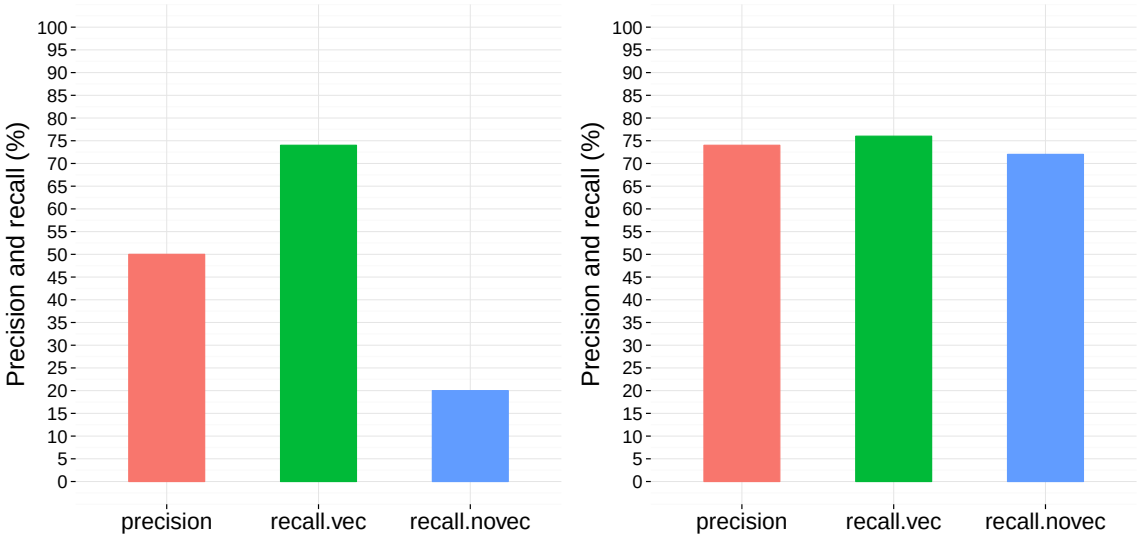
In summary, vectorization is a double-edged optimization technique that can have a significant impact on the execution time, power and energy consumption. If performed correctly, it can be a powerful technique for energy reduction. Several cases were presented in which energy was reduced by as much as 90%. At the same time, another example was presented in which vectorization caused around 50% greater energy consumption. Improvements in performance were achieved through not only

computational parallelization, but also more efficient memory accesses. This was the case even if a significant number of SISD instructions were not able to be vectorized. On the other hand, detrimental performance was found to be caused by packing instructions and artifacts introduced by the compilation process. This experiment also revealed the limitations of modern compilers in transforming the code for successful vectorization, which confirmed the results of Maleki et al. on TSVC [10].

Having analyzed the vectorization performance, the next step was to train an SVM using the extracted software characteristics and the measured energy ratios, which was then tested using cross-validation. Cross-validation required less than 10 seconds to perform for all programs. The predictor performance was evaluated according to its precision and recall. Precision is the ratio of correct predictions over all predictions. Recall is similar to precision but computed for each class to be predicted. It is defined as the ratio of correct predictions for that class over all predictions for that class. The objective of these metrics was to estimate how good our predictor was at deciding whether vectorization would be beneficial or not when applied to a given input program.

Figure 5.6 a shows the precision and recall ratios for our first attempt at predicting vectorization profitability. Unfortunately the precision was at 50%, in other words, predictions were being performed at random. Furthermore, the recall of the *should vectorize* class on the second bar was several times larger than the *should not vectorize* recall on the third bar. Our predictor was biased, meaning that it would be much better at predicting one class than the other. A balanced predictor was desired since the impact of mispredicting vectorization can also be high.

In order to improve prediction performance, first we re-examined Figure 5.5 which shows that most points lie around the neutral region ②. These points mostly correspond to programs for which the compiler was unable to apply vectorization. We decided to tune the predictor by methodically removing points within the neutral



(a) Untuned predictor.

(b) Tuned predictor.

Figure 5.6 : Predictor's precision and recall.

region, starting at the center of the region and moving outwards until acceptable precision was achieved (Table 5.3). When implementing the proposed methodology in a compilation flow, an automated alternative would be to parse the compiler optimization reports to detect those loops in which vectorization failed in order discard them before the training phase.

There are several reasons that justify the predictor tuning. First by intuition, it would be unreasonable to ask a predictor to accurately classify those points that are on the fringe of being labeled as beneficial or detrimental. Furthermore, mis-predicting these points would have the same weight as mis-predicting points that do have a significant impact on energy consumption, thus unnecessarily reducing the predictor's precision. The second reason is that those points in the neutral region are mainly composed of programs which the compiler was unable to vectorize. As previously mentioned, the objective of this work is not to study or predict the compiler's ability at transforming the code to achieve vectorization, but rather on whether vectorization should be applied or not according to the impact on energy.

The last two reasons that justify tuning are related to machine learning techniques. During the training phase, it is standard practice to manipulate the training data in order to improve prediction accuracy. This is true as long as training does not include points to be used in the testing phase, otherwise known as self-validation. But this situation is avoided in this work by using cross-validation. As an example of the data manipulations used, Andrew Ng suggests in his machine learning course to artificially generate training examples by adding noise to already existing training examples to improve hand-writing recognition [d] . Thus, removing the neutral points from the training phase is an acceptable practice in machine learning. The last reason is whether removing these points makes sense during the predictor's testing phase and during precision and recall evaluation. In this research we are looking for a predictor that is accurate at predicting the cases with significant impact. However, the points within the neutral region have an average energy ratio of 1.002. On average, the benefit of having a predictor that is accurate at predicting the neutral cases is negligible. Nevertheless, we found that there is a trade-off between the number of points within the neutral region and the overall predictor precision. That is, minimizing the number of points in this region resulted in higher precision and recall scores for the points that do have significant impact on energy. In this way, tuning reveals the true vectorization profitability prediction capacity. In a real application, the predictor would be inaccurate at predicting for the neutral programs, but the performance of these programs would not vary much in the case of mis-predictions.

The tuning was performed until either the precision and recall stabilized or we reached the end of the neutral region. Figure 5.7 shows the tuning progression as points were removed and how it affected the precision and recall metrics. The end of the neutral region was reached and the precision and recall curves did not converge.

[d] http://class.coursera.org/ml-005/lecture

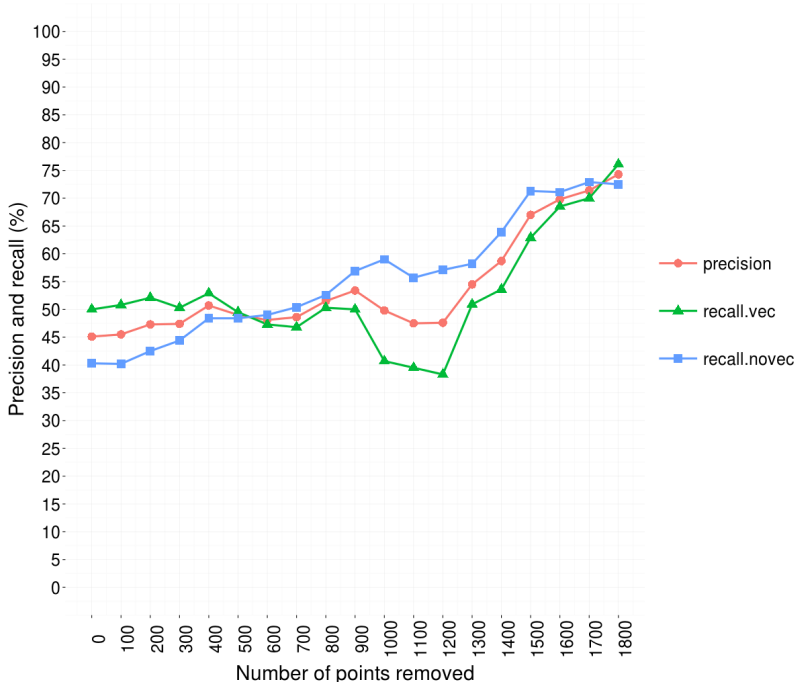
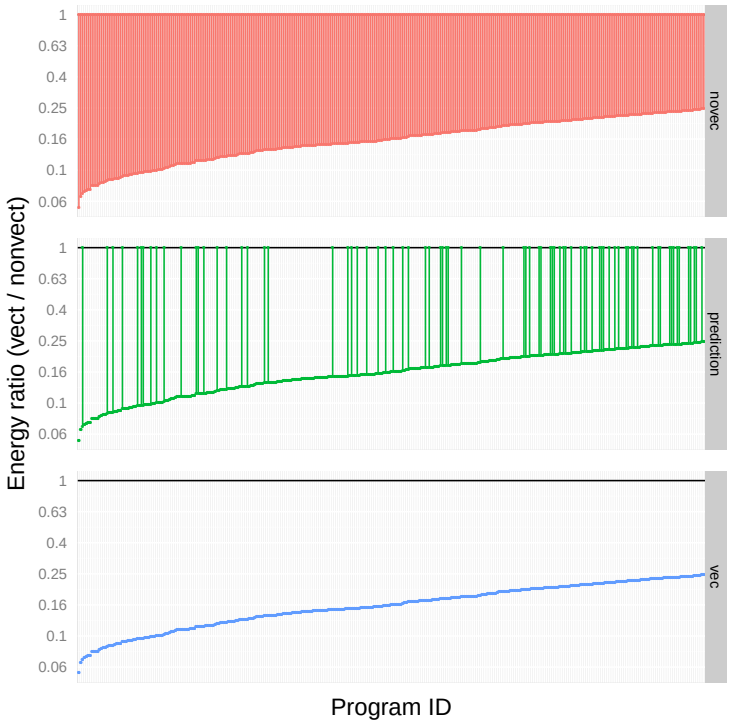


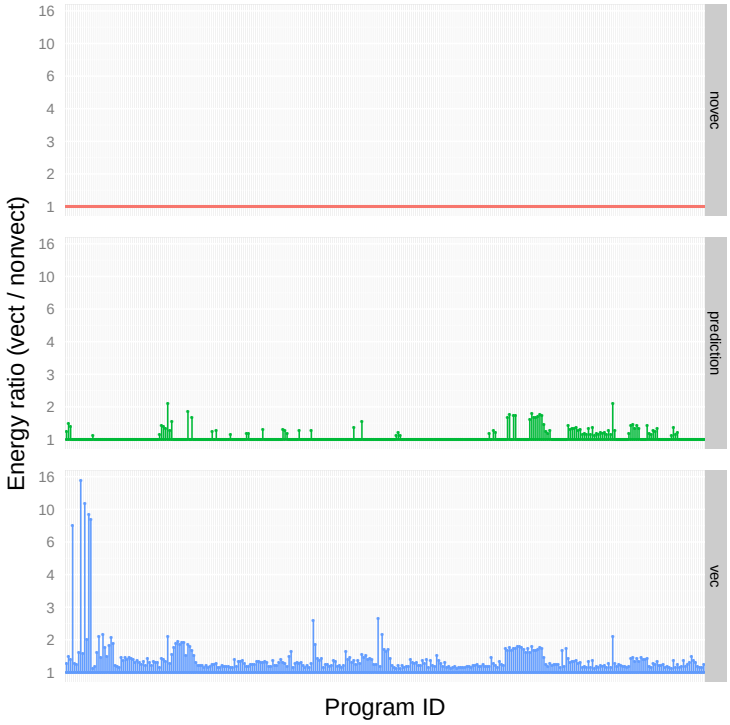
Figure 5.7 : Predictor tuning.

However, by carefully removing these neutral points, the recall for the *should not vectorize* class had a dramatic increase from 20% to 70% while the recall for *should vectorize* remained stable as shown in Figure 5.6 b. As a consequence, the precision also increased from 50% to 74%. We achieved training a vectorization profitability predictor with fairly high precision and balanced recall.

The precision and recall bar plots presented above give an aggregate measure of the overall predictor performance. Next, we analyze the predictor performance on a per program basis to get a better sense of its utility. Figure 5.8 shows the distance in energy ratio to a perfect predictor for three optimization strategies. The vertical axis marks the energy ratios for each remaining program after predictor tuning. There is one point per program arranged in increasing energy ratio. The range of programs is divided in two; Figure 5.8 a corresponds to programs that can benefit from vectorization while Figure 5.8 b corresponds to the programs where vectorization is detrimental. These figures are explained in more detail next.



(a) Beneficial vectorization region.



(b) Detrimental vectorization region.

Figure 5.8 : Energy ratio distance to a perfect predictor.

A perfect predictor would always choose the correct answer of whether to vectorize or not. In other words, the resulting energy ratios of the perfect predictor are either below unity when vectorization is beneficial or at worst unity when it is detrimental. The three optimization strategies are *novec* (vectorization is always deactivated), *prediction* (our predictor), and *vec* (vectorization is always activated). The vertical lines show the distance from applying that strategy to the ratio of the perfect predictor. The better the strategy, the less vertical lines in the plot.

For the *novec* strategy, all beneficial cases were missed but there were no detrimental cases as vectorization was never applied. On the contrary, for the *vec* strategy all beneficial cases are hit but so are the detrimental cases. Therefore no strategy is good by itself in the general case. But when using *prediction*, which is our proposed strategy, the worst detrimental cases are avoided while at the same time hitting 76% of the beneficial vectorization cases.

Statistics for the energy ratios of Figure 5.8 a and Figure 5.8 b are summarized in Table 5.7 and Table 5.8 , respectively. Comparing the results in Table 5.8 a where vectorization is beneficial, the predictor managed to predict correctly for the program with the highest energy reduction of 94% (0.06 ratio). On average the predictor achieved 64% (0.36) reduction in energy, while the perfect predictor and the *vec* strategy reached 84% (0.16). On the other hand, the *novec* strategy missed any benefit from vectorization.

Strategy	Avg	Sdev	Min	Max
Perfect	0.16	0.05	0.06	0.025
Vec	0.16	0.05	0.06	0.025
Prediction	0.36	0.36	0.06	1.00

Table 5.7 : Energy ratio statistics for Figure 5.8 a

Strategy	Avg	Sdev	Min	Max
Perfect	1.00	1.00	1.00	1.00
Vec	1.33	1.20	1.06	15.13
Prediction	1.05	0.11	1.00	1.67

Table 5.8 : Energy ratio statistics for Figure 5.8 b

Now comparing the results of Table 5.8 b where vectorization is detrimental, on average the predictor only increased energy consumption by 5% (1.05) versus 33% (1.33) increase when the *vec* strategy was applied. And the worst prediction increased energy consumption by 67% (1.67) while the *vec* strategy increased energy consumption by 15 times (15.13). Clearly, using our predictor is the better strategy with 76% probability of choosing vectorization when appropriate while increasing the energy consumption by only 5% on average.

Chapter 6
Predicting Multiple Optimizations

In the previews chapters we presented Machine Learning Driven Compiler Tuning (MLDCT) techniques for predicting vectorization profitability, modeling the predictor as a binary classifier. In this chapter we turn to the more complex problem of predicting optimization strategies composed of multiple optimizations.

Figure 6.1 illustrates the challenge of choosing a good optimizations strategy. Here a *good strategy* is defined on a program basis, and it is an optimization strategy whose speedup difference to the best strategy for that program is at most 0.05. The histogram accounts for 432 strategies applied to 750 tensor contraction kernels, and it shows the number of optimized programs that benefit from a good strategy. Tensor contraction kernels are generalized multiplication loops that are often used in computational chemistry suites and are are described in Section 4.3. The optimizations considered include loop permutations and unroll-and-jam, whose parameters are discussed later in this chapter. There are no beneficial strategies when vectorization is deactivated, and have been removed from the histogram for clarity. As the histogram in Figure 6.1 shows, there is no single strategy that is good for all or most kernels; the good strategies are distributed across the optimization space in the horizontal axis. This highlights the need for a smart optimization space exploration technique as an alternative to the methods employed by modern compilers.

To this end, we have devised an optimization space reduction methodology called *Common Good Strategies* (CGS) that prunes optimization strategies that are unlikely

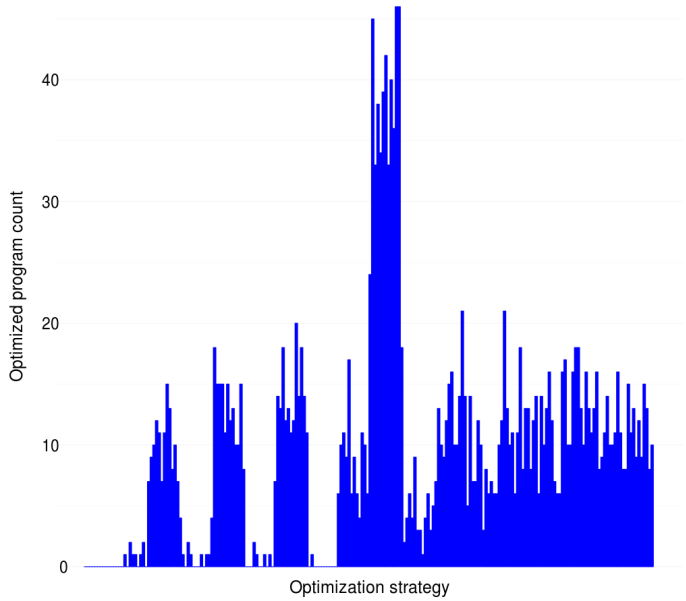
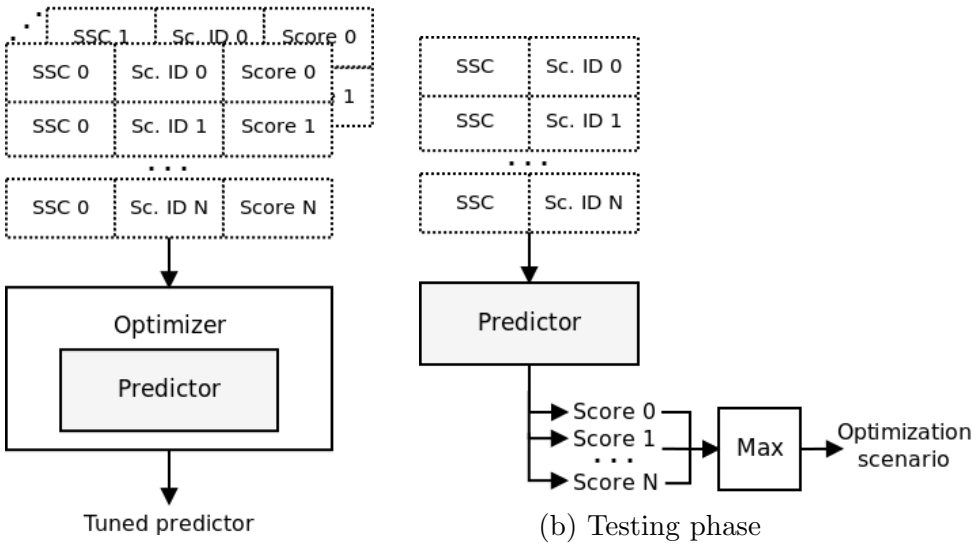


Figure 6.1 : Distribution of optimized programs that benefit for each optimization strategy, that is, that are within 0.05 distance from their best speedup.

to be profitable for a new program. This technique can be combined with those proposed in related works for predicting optimization strategies. We show how CGS makes score-based prediction, such as those used by Park et al. [4] [2], practical by reducing the prediction time from exponential growth on the number of optimization techniques, to logarithmic growth on the number of training programs. More specifically, we propose three optimization space reduction techniques that when combined, make score-based prediction feasible by bounding the prediction time to logarithmic complexity without degrading prediction accuracy. The effectiveness of this methodology is illustrated with a gain of 40% in speedup over the original score-based prediction scheme, on a set of 864 optimization strategies applied to tensor contraction kernels.

6.1 Score-based prediction

To explain what we term *score-based prediction*, let us review how MLDCT works from the point of view of this technique. A predictor's usage is divided in two phases;



(a) Training phase

(b) Testing phase

Figure 6.2 : Training and usage of the predictor.

the training phase (Figure 6.2 a) and the testing phase (Figure 6.2 b). The purpose of the training phase is to tune the predictor with a set of inputs and expected outputs. In the testing phase, we ask the predictor to output a score for the given program and optimization strategy encoding.

During the training phase, an optimizer is used to tune the predictor with as many example cases as possible in order for it to recognize future inputs. For each example case, the predictor is presented with a training input and its prediction is compared with the expected output. The predictor has coefficients that are tuned by the optimizer to increase the accuracy, or the rate of correct predictions to overall predictions. More specifically, a *training example* is composed of characterization of the input program and the target output, which is a score that is computed after running the test program. The collection of training examples is called the *training set*.

The first step for building a training set is to select a benchmark representative of the programs we want to predict for. Since the source code cannot be input directly into the predictor, each of these test programs is encoded into vectors of software

characteristics using a profiler. This is followed by using the target compiler to apply each optimization strategy independently to each of the benchmark programs. The optimized programs are then executed and a score is computed for each, the program speedup in our case. Because we are using baseline software characteristics, an encoding of the strategy (Sc. ID) is also included as input to the predictor such that it can associate the baseline software characteristics to the score of applying some optimization strategy. This encoding we are using is an enumeration of the strategy, from 1 to the maximum number of strategies. In summary, the predictor is trained with one matrix per benchmark program, where each row of the matrix has the baseline software characteristics for that program, a strategy identifier that matches the row number, and the performance score of the strategy applied to that program.

During the testing phase, the objective is to predict for a new input program the one or more optimization strategies that are expected to produce the highest speedup. Figure 6.2 b shows a block diagram of the testing phase, where the predictor outputs the strategy with the maximum score. The inputs to the predictor are generated in a similar fashion to the training phase, except that the score is unknown. That is, each row of the matrix has the baseline software characteristics for the new program and a strategy identifier that matches the row number. These matrices, one per new input program, are called the *testing set*. As the matrix is passed to the predictor, it will predict one score per row. The output optimization strategy can be the one with the highest score, or the first N strategies which could then be applied and executed to choose the best one.

Compared to score-based prediction, a classifier that outputs the best optimization strategy directly would be ideal. However, it is not clear how the optimization strategy can be encoded as an input and output for the predictor's training and testing phases. One approach may be to fix the optimization strategy and use it as a

sort of template, whereby multiple classifiers are trained for predicting the parameters of each optimization. For example, one classifier to predict whether to apply vectorization or not, and another one to choose which loop permutation to apply. This is a cumbersome approach and that does not consider the impact that optimization ordering. Furthermore, as will be explained in the next section, there are multiple optimization strategies that may be beneficial to a program rather than a best one. Using a classifier is then further complicated if we also consider how to predict multiple good optimizations.

On the other hand, a more natural approach is to use regression to score the strategies. The limitation of this approach is that that one score has to be predicted per optimization strategy, which binds the prediction time to the size of the optimization space. And because this is a combinatorial problem, both the optimization space and the prediction time will grow exponentially. Figure 6.3 illustrates how the prediction time grows as a function of adding new optimization techniques, assuming one prediction takes 1 second. The optimization strategies considered in this example subsequently add one new technique or modify the optimization parameters. When considering vectorization, loop nest interchange, tiling and unroll-and-jam, the prediction time would reach more than 2 hours and 30 minutes for a single program. In practice, optimization strategies may be arbitrarily large, making this an unfeasible approach. In the following sections we solve this limitation with a methodology that binds the prediction time to the training set's size while maintaining the predictor's accuracy.

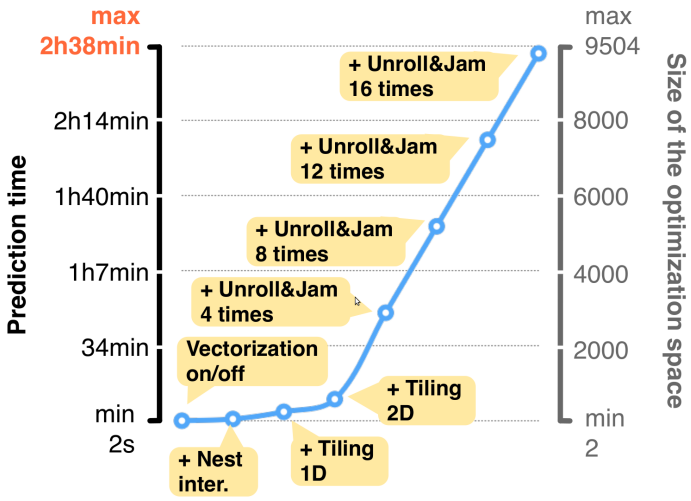


Figure 6.3 : Prediction time as a function of the number of optimization techniques, assuming 1 second per prediction.

6.2 Experimental setup

Our experimental benchmark was a collection of auto-generated Tensor Contraction (TC) kernels, as described in Section 4.3. We selected four standard compiler optimizations and parameters to apply to the TC benchmarks, which are listed in Table 6.1 . The optimization space totaled 864 strategies. Intel's ICC compiler was used to apply the optimization level (O2 or O3) and vectorization. The loop permutations and unroll-and-jam transformations are applied by the TC generator. Execution was done on a single core of a Xeon E5 E5-2670 v2 processor, executing 5 times per program, and the average speedup was computed. Training and testing was then performed using 5-fold cross-validation over the examples set to train and test a linear regression model *lm* provided in the standard R language environment.

In the following sections we will present the Common Good Strategies technique which makes score-based optimization predictors practical, and in the case of our benchmarks, also increase the prediction accuracy. The results are summarized in the box plots of Figure 6.4 , where the vertical axis is the speedup of the technique with respect to ICC with no optimization level flags (similar to level O2). The first two

Optimization	Parameters
Optimization level	O2 or O3
Vectorization	Forced off (-no-vec)
	or forced on (-vec -xSSe3)
Loop permutation	All combinations for the 4
	nesting levels
Unroll-and-jam	1 dimension sized 4 or 8

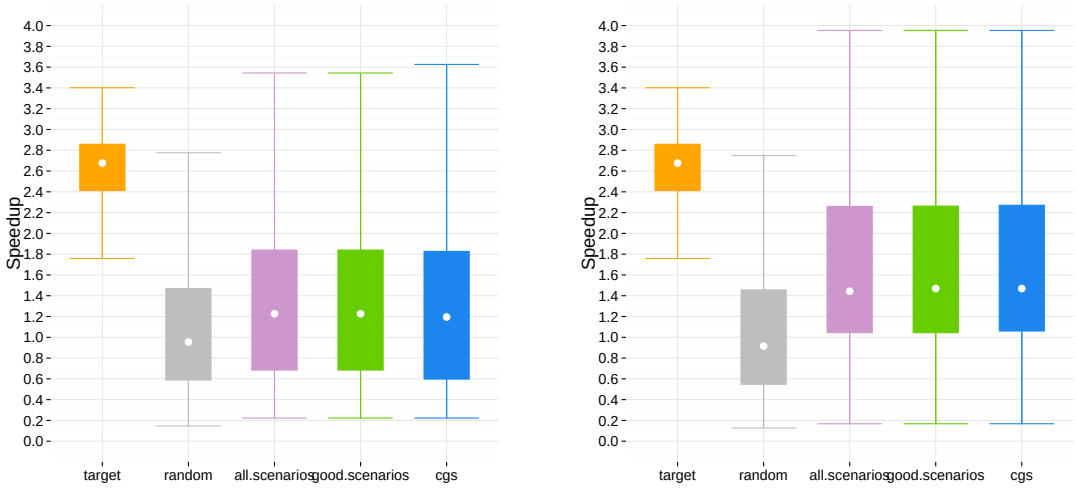
Table 6.1 : The optimization space. An optimization strategy is composed of one parameter per optimization.

boxes are labeled *target* and *random*, and serve as baselines to assess the effectiveness of each predictor. The *target* box shows the speedups for a perfect predictor. On the other hand, the *random* box shows the speedups for random optimization strategy selection.

6.3 The complete optimization space versus the good strategies

First we start with a linear regression predictor that is trained and cross-validated over the complete optimization space, and which we consider to be the baseline predictor. The speedups distribution is shown in the box plot labeled *all.strategies* in Figure 6.4 a. Note that while the distribution is better than random prediction, the first and part of the second quartile cause speed-downs. More importantly, when a new program is to be optimized, the predictor has to score every optimization strategy. Yet the number of optimization strategies grows exponentially and so does the prediction time, which makes this method impractical.

Our first approach at improving this methodology was to reduce the optimization space to only the good strategies. A *good strategy* is defined on a program basis, and it is an optimization strategy whose speedup difference to the best strategy measured for that program is at most 0.05. The selection of a good strategy instead of the best optimization strategy per program that is used in related works, is based on the



(a) Single predictor.

(b) One predictor per strategy.

Figure 6.4 : Speedups ranges for each prediction technique. The circle within each box marks the median value and the whiskers the first and last inter-quartile range.

observation that the definition of *best* is often vague. As an illustration, Figure 6.5 shows 10 TC kernels, each with the range of speedups for their top 10 optimization strategies after 5 executions. The best strategy changes between executions and no one strategy can be identified as the best. Therefore it is more sensible to choose those strategies that are within a performance margin on a program basis.

A Good Strategy (GS) set can be constructed with Algorithm 1. The speedup range for a predictor trained using GS is shown in the box labeled *good.strategies* of Figure 6.4 a. By using this technique, the optimization space was reduced to 328 strategies out of the original 864 (62% smaller) while keeping the prediction performance similar to training with all strategies. GS can greatly reduce the number of strategies, yet the prediction time is still bound to the growth of the optimization space.

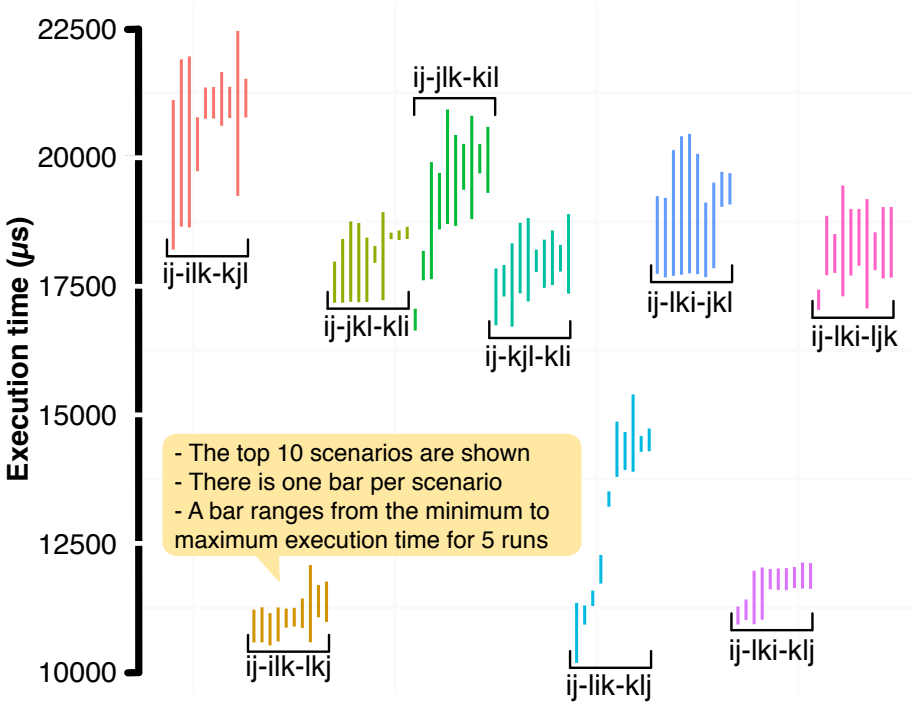


Figure 6.5 : Speedup range among 5 execution time measurements of the top 10 optimization strategies for 10 TC kernels.

Algorithm 1 Generate the set of good strategies per program

```
procedure GS
    gs ⇐ ∅
    for all training programs do
        best speedup ⇐ max(program's speedups)
        for all program strategies do
            difference ⇐ best speedup - strategy's speedup
            if difference < margin then
                gs ⇐ gs ∪ {(program, strategy)}
```

Algorithm 2 Generate the set of common good strategies

```
procedure CGS
    gs ⇐ GS(training programs)
    cgs ⇐ {MFCGS(gs)}
    for all training programs do
        strategies ⇐ {s | (p, s) ∈ cgs where p = program}
        if strategies ∩ cgs = ∅ then
            cgs ⇐ cgs ∪ {random(strategies)}
```

6.4 The common good strategies

The Common Good Strategies (CGS) is a refinement of training with the good strategies. Rather than considering all the good strategies, we can iteratively construct a set that adds a strategy only if there is no good strategy for the current program present in the set.

Algorithm 2 lists the steps to construct the CGS. First, it computes the GS and iterates over every training program. If a good strategy is not present in the CGS for the current training program under analysis, one good strategy is picked at random from the good strategies for this program. The CGS is not unique because new strategies are added at random. To try to construct the shortest CGS, the set is first initialized with the Most Frequent Common Good Strategy (MFCGS). The MFCGS is the good strategy that is most frequently present among the training programs.

By employing CGS, the total number of strategies is bounded to the total number of training programs. When training and testing with all the strategies, and to a lesser degree with the GS, the number of strategies and the prediction time grows exponentially with the number of optimization techniques. With CGS however, there can be at most the same number of strategies as training programs in the unlikely event that each program has a unique good strategy. More precisely, the CGS can be as large as the training set, or as large as the optimization space if the space is smaller than the training set. This technique, as we are about to show, makes optimization score-based prediction practical.

A third predictor is trained using CGS, and the speedup range is shown on the box labeled *cgs* in Figure 6.4 a. In the resulting CGS there is an average of 212 strategies out of a total of 864, a four-fold decrease in the optimization space. The performance however is slightly less than when training with all strategies or GS. In exchange, the number of strategies is bounded from exponential to logarithmic growth on the number of training programs as is shown in Figure 6.6 . For each

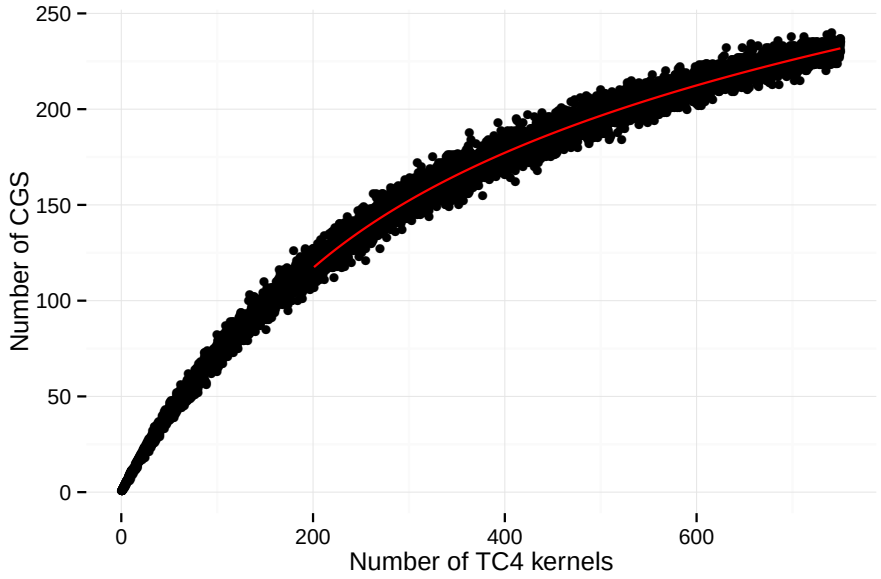


Figure 6.6 : Evolution of the CGS size with an incrementing number of training programs.

step in the horizontal axis in this plot, a CGS is constructed for each of 10 randomly chosen training programs samples of size 1 to 750. Multiple samples are used to account for the non-determinism when constructing the CGS. The red curve is an approximation of the growth, and is the logarithmic expression $-342.5726 + 86.7561 * log(num.programs)$.

6.5 Predicting without the strategy as input

The only reason the strategy identifier is included as a predictor input is to indicate which optimization strategy was applied to the baseline software characteristics. This identifier is an artifact of the prediction scheme; the number itself carries no meaning as to the behavior of the program and we hypothesized that it could actually reduce prediction performance.

To be able to remove the strategy identifier, rather than training one predictor for all strategies, one predictor is trained for each strategy. This method simplifies the problem to that of predicting the speedup for a single strategy. Figure 6.4 b shows

that using multiple predictors increases the prediction performance for all techniques. Now the second quartile, and thus the majority of predictions, result in speedups. The speedup geometric mean for *all.strategies* and *good.strategies* increases by 28% and 29%, respectively, over using a single predictor. But the proposed CGS technique benefits the most, with a mean increase of 40% in speedup.

6.6 Conclusions

In this chapter we studied how to make score-based optimization predictors more practical in a machine learning driven compilation flow. To this end, we proposed an optimization space reduction technique called Common Good Strategies. This methodology makes score-based prediction feasible by bounding the prediction time from exponential growth on the number of optimization techniques, to logarithmic growth on the size of the training set. This was achieved without degrading the predictor's performance, increasing the mean program speedup by up to 40% in our testing set.

A next step for this research is to try the proposed techniques on more varied benchmarks and to increase the optimization space. Still, there are several challenges besides prediction time that also need to be addressed in order to make MLDCT practical. These challenges are presented in Section 8.1.

Chapter 7
Related Works

There are already several works on Machine Learning Driven Compiler Tuning that apply support vector machine (SVM) [4, 28], nearest neighbor (NN) [27, 28], artificial neural networks (ANN) [3,5], and logistic regression [21]. They differ mainly on their prediction objective: Stephenson et al [28], Park et al. [4] and Agakov et al. [27] predict parameters for code optimizations, Kulkarni et al. [3] order optimization passes in the middle end, and Pekhimenko et al. [21] use machine learning to focus search algorithms. In this chapter we summarize the related works in MLDCT and their contributions.

The work from Stephenson et al. [28] used classification to determine for a given program the unroll factor that yields the best performance. Their test programs consisted of 2500 loops extracted from several well-known benchmarks targeted at high-performance computing as well as embedded computing. They leveraged both SVM and NN, and managed to correctly predict the best unroll factor with an accuracy of 65% and 62%, respectively. This proved to be far better than their baseline, the Open Research Compiler[a] .

Stock et al. [5] also targeted vectorization profitability prediction, but using regression rather than classification. They extracted their own set of static software characteristics from assembly in order to predict performance. Their technique

[a] Now called Open64: http://www.open64.net/

showed high accuracy on a tensor contraction kernels benchmark, consisting of perfectly nested, independent loops with a single innermost statement. Their technique, however, performed only slightly better than random prediction on stencil kernels.

In Chapter 4 we used SVM to predict vectorization profitability for the Intel Compiler on the TSVC compiler benchmark, and achieved around 70% prediction accuracy considering as predictor input our own set of high and intermediate level static software characteristics and the unroll factor. In Section 4.3 we focused our vectorization predictor on tensor contraction kernels and two sets of software characteristics tailored for this type of program. We achieved up to 98% accuracy, close to the accuracy obtained when using instruction mix at the assembly level without requiring compilation, thus greatly accelerating the training phase. In Chapter 5 we further considered the impact of vectorization on energy consumption in embedded systems, and also trained an SVM predictor that can target energy reduction.

Kulkarni et al. [3] used machine learning to tackle the complex problem of the ordering of optimization techniques. They modeled an optimization strategy using a Markov process to construct optimization strategies iteratively, one optimization technique at a time, using ANN at each step. They applied their method to the just-in-time compiler of a Java virtual machine, and managed to reduce the execution time of compiled programs by up to 20%.

Similar to our work, the motivation behind the GCC Milepost project by Fursin et al. [7] was to add machine learning capabilities to mainstream compilers so that they could automatically choose optimization sequences for heterogeneous reconfigurable processors. Milepost GCC supports multi-objective optimizations to reduce the execution and compilation time, and also the code size. They selected over 50 static software features to cover a large variety of optimizations. Their system also stores training data in an online database to allow collaborative optimization between users and continuous retraining to improve prediction accuracy. However, it is not

clear how this heterogeneous data would be used for training in a production environment. They achieved an average of 11% reduction in execution time for the MiBench suite running on an ARC reconfigurable processor. A limitation of their approach is that feature extraction and optimizations are performed at the function level, while we considered loop-level granularity. Furthermore, in our experiments in Section 4.3 we discovered that the Milepost software characteristics are not appropriate for certain types of programs such as tensor contractions, which are syntactically identical but exhibit very different memory access patterns and consequently, different performance.

Most works in MLDCT endeavor to improve the compiler's output by reducing the execution time of the compiled programs. The works from Agakov et al. [27] and Pekhimenko et al. [21], however, 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. was to reduce the time required to find the best optimization sequence to apply to a given program. They adopted a technique based on NN to bias an existing search algorithm (random or genetic) and managed to reduce the search time by one order of magnitude. On the other hand, Pekhimenko et al. [21] used logistic regression to determine the parameters of optimization techniques inside a fixed optimization strategy, 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 program's execution time.

The work of Park et al. that [4] proposes an alternate way of characterizing the input programs to facilitate machine learning modeling while improving prediction performance. 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, it requires to actually execute the

program which may be impractical in real scenarios due to time constraints and lack of run-time data [25]. Software features may further be hardware dependent and hardware independent. The former suffers from lack of portability, for example using hardware counters on a given machine that are not available on another one, because of differences in micro-architecture. To address these problems, Park et al. leveraged graph mining techniques to directly feed the program's dataflow graph to an SVM for predicting the best optimization strategy. This is contrast to the common way of selecting software features which is a tedious process that may not yield high accuracy. They compared their approach to Milepost GCC and obtained better prediction accuracy.

The works presented in this chapter reported promising results on different aspects of MLDCT. Yet, there are several challenges we identified during our research that are preventing MLDCT from becoming a mainstream compiler optimization driving technique. In the next chapter we discuss the challenges that future research should tackle in order to make MLDCT practical in production environments.

Chapter 8
Conclusion

The main objective of this research was to find ways in which to assist a compiler in selecting beneficial optimization strategies in order to improve its optimization capacity at speeding up programs. This was motivated by the observation that conventional compilers use heuristics that rely on hand-built machine models that are not only difficult to develop, but have been insufficient in handling complexity of modern hardware, compilers, and input programs. We explored the use of machine learning techniques for predicting from a high-level source program when to apply a given optimization strategy to boost compiler performance, an approach we termed Machine Learning Driven Compiler Tuning (MLDCT). MLDCT is an attractive solution because inherently encapsulates through a training process the target system and compiler behaviors. Our research yielded promising results for this method, although we also identified several challenges that need to be addressed in order to bring MLDCT to the hands of programmers who want the compiler to make best use of the available resources in an automated fashion. These challenges are discussed in the next section.

In Chapter 4 we began our investigation in MLDCT by predicting profitability for a single optimization called vectorization. Vectorization is a powerful technique that, when applied correctly, can bring significant speedup due to data parallelization and optimized memory access, but at the same time can result in detrimental performance if applied indiscriminately. We demonstrated how Support Vector Machine can be used to predict vectorization profitability for the Intel Compiler on TSVC,

a benchmark suite for vectorizing compilers. Prediction was performed from high- and intermediate-level static software characteristics, and we achieved a prediction accuracy of around 70%, compared to 56% accuracy for the Intel Compiler.

Our next objective was to improve the vectorization prediction accuracy by selecting new software characteristics and constructing a larger training set. For the training set, we utilized auto-generated tensor contraction kernels for which we selected software characteristics tailored to this benchmark. In doing so, we achieved 98% of accuracy, slightly below the accuracy obtained from assembly level characterization without the need of compiling the input program. Thus we were able to find a methodology to train a highly accurate vectorization predictor, but with the limitation that it was specific to our benchmark.

In Chapter 5 we presented our research results on the impact of vectorization on the power and energy consumption of system-on-chip devices, and how the vectorization profitability predictor could also be useful in reducing energy consumption. The experiments were carried out on an implementation of the ARM Cortex-A8, and the results showed an average 64% decrease in energy consumption and only 5% increase in the case of mispredicitons.

In Chapter 6 we considered the more complex problem of predicting multiple optimizations. We proposed an optimization space pruning technique that would make score-based predictors feasible by bounding the prediction time to the size of the training set. This was achieved without decreasing prediction accuracy, but rather increasing by up to 40% in speedup over the original score-based prediction scheme for our tensor contraction benchmark.

8.1 Challenges in MLDCT

This section summarizes what what we have found to be the main challenges in making MLDCT a practical technique for production environments. Future research efforts should concentrate in these areas.

Challenge 1: Software characterization. MLDCT relies on an accurate software characterization in order to associate a program to be optimized to a beneficial strategy. There is no clear methodology on how to best encode an input program in order to predict a beneficial optimization strategy with high accuracy, and a characterization that is good for one type of program may not be adequate for another.

Programs can be characterized whether dynamically during execution, semi-dynamically during assembly generation, or statically from high-level or intermediate representations. Higher levels are more practical to profile at the expense of being more distant from the final dynamic behavior. Another aspect to consider that goes in hand with designing software characteristics is determining whether to predict optimization strategies by predicting performance as in regression, or by comparing programs as in classification.

We have found that using visualization techniques can serve as an aid in qualitatively comparing different sets of software characteristics. To do so, we project all the data points from the multi-dimensional space of software characteristics to a two-dimension plane. We use a method proposed by van der Maaten et al. [24], called t-Distributed Stochastic Neighbor Embedding (t-SNE). It is a dimension-reduction technique meant for visualization that strives to reproduce in 2D the similarities between multi-dimensional data. The output is a visualization where data with similar software characteristics are represented close to each other.

We explain the usefulness of t-SNE within MLDCT by comparing the *proposal1*, *proposal2*, and related characteristics and data set introduced in Section 4.3. Figure 8.1 shows the data projection for each set of software characteristics, with the color indicating the class; whether vectorization is profitable or not. For clarity, only a representative close-up of the whole data is shown.

The best sets of software characteristics are also the ones that best discriminate the two classes in term of similarity. Indeed, both classes overlap for *milepost* and

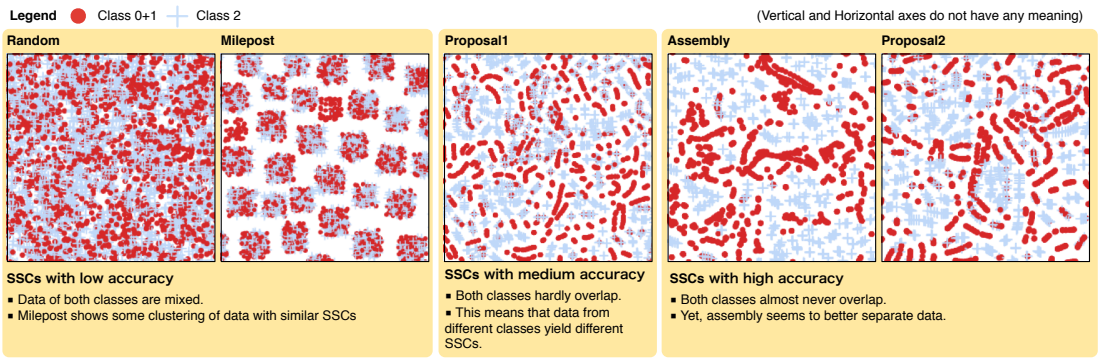


Figure 8.1 : Projection of the data from the software characteristic space to a 2D plane by means of the t-SNE visualization method.

random, but are clearly separated for other sets of software characteristics, especially *proposal2* and *assembly*. It is interesting to notice that even though *milepost* exhibits some clusters, those are unrelated to the classes. This means that it stores some information about the programs, but this information is unrelated to the problem of deciding if vectorization should be applied.

Another insight to be gained from this visualization is that we are likely to obtain similar prediction trends regardless of the machine learning algorithm that we use. This is because the set of software characteristics should at least be able to discriminate between classes. If not, no machine learning algorithm would be able to re-create this information. This is an intrinsic property of the set of software characteristics, unrelated to the machine learning method.

Challenge 2: Predictor modeling. How to define the predictor's input and output in order to maximize prediction performance? For example, an optimization strategy can be fixed and each optimization parameter predicted independently. Another alternative would be to predict a strategy one optimization at time. If using regression, the predictor's input may be the software characteristics and the strategy for which to predict the performance. In this case a strategy encoding must also be defined. The strategy input can be avoided by training one predictor for each strategy. However, the number of optimization strategies may be too large for this

technique to be practical, for which a possible solution is presented in Chapter 6. Classification is also an alternate approach but is harder to model and is prone to a strongly unbalanced training set, which ML techniques do not handle well.

Challenge 3: Predictor generality. How varied are the types of programs for which the predictor can accurately find beneficial optimizations? This entails first determining which application domain would benefit the most from MLDCT. The next step is identifying a source from which to mine programs to train and test the predictor. There are different sources that can be mined to generate the software characteristics including benchmarks, auto-generated code, and crowd-sourcing [16]. In Chapter 3 however, we discovered that performance benchmarks are not appropriate for MLDCT research since they have already been hand-optimized. Once we have a program source to mine, we need some method to ensure enough program varieties are covered [25].

In Chapter 6 we relied on tensor contraction code generators as a way of sourcing enough programs for training an accurate predictor. This concept could be expanded to create generators that can produce a wide variety of programs using a small set of program prototypes with varying parameters, in order to train predictors that can be more generally applied in production environments. One possibility is to generate a training set based on the Berkeley Dwarfs Mine,[a] a collection of algorithms that represent a broad range of important applications.

Challenge 4: Research reproducibility. In the MLDCT field there is a general lack of trust in published results because the experimental data is usually not readily accessible for independent investigation. Moreover, much research effort is lost because the tools employed are often developed ad-hoc and also not made publicly available. This is further complicated because there is no standard methodology

[a] http://view.eecs.berkeley.edu/wiki/Dwarfs

or metrics for evaluating and comparing different prediction approaches. Therefore there is a need for experimental environment sharing services and predictor evaluation methodologies to enable collaboration that can lead to resolving the challenges presented in this work.

There are a few initiatives to increase the trust-worthiness of research results and facilitate collaboration between researchers, particularly in the MLDCT field. One of these projects is called cTuning [b] . We have also published our experimental data through our TeaBowl Project [c] in hopes to motivate other researchers to do the same to help tackle each of the challenges presented in this section.

[b] http://ctuning.org/lab/people/gfursin

[c] As of October 2015, can be accessed at http://54.64.73.237/organization/teabowl-project

Bibliography

[1] D. Meyeret al. Misc Functions of the Department of Statistics (e1071) (online: http://CRAN.R-project.org/package=e1071)

[2] E. Park et al. Predictive Modeling in a Polyhedral Optimization Space, In *International journal of parallel programming*, vol. 41, pp. 704-750, 2013.

[3] S. Kulkarni and J. Cavazos. Mitigating the compiler optimization phase-ordering problem using machine learning, In *Conference on object-oriented programming, systems, languages, and applications (OOPSLA)*, 2012.

[4] E. Park et al. Using graph-based program characterization for predictive modeling. In *International symposium on code generation and optimization (CGO)*, pp. 196-206, 2012.

[5] K. Stock et al. Using machine learning to improve automatic vectorization, In *ACM transaction on architecture and code optimization (TACO)*, vol. 8-4, 2012.

[6] S. Kamil and A. Fox. Bringing parallel performance to Python with domain-specific selective embedded just-in-time specialization, In *The 10th python for scientific computing conference*, 2011.

[7] G. Fursin et al. Milepost GCC: machine learning enabled self-tuning compiler, In *International journal of parallel programming*, vol. 39, pp. 296-327, 2011.

[8] J. L. Hennessy and D. A. Patterson. In *Computer Architecture (Fifth Edition)*, Morgan Kaufmann, 2011.

[9] M. Markus et al. Compiler Based Exploration of DSP Energy Savings by SIMD Operations, In *Proceedings of the 2004 Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2004.

[10] S. Maleki et al. An Evaluation of Vectorizing Compilers, In *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2011.

[11] K. Trifunovic et al. Polyhedral-Model Guided Loop-Nest Auto-Vectorization, In *Proceedings of the 2009 18th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2009.

[12] A. Trouv et al., Using Machine Learning in Order to Improve Automatic SIMD Instruction Generation, In *International Conference on Computational Science (ICCS)*, 2013.

[13] J. Fitzpatrick, An Interview with Steve Furber, In *Communications of the ACM*, 2011.

[14] Texas Instruments, OMAP3530 Power Estimation Spreadsheet, http://processors.wiki.ti.com/index.php/OMAP3530_Power_Estimation_Spreadsheet

[15] Texas Instruments, TMS320DM6446/3 Power Consumption Summary, http://www.ti.com/lit/an/spraad6b/spraad6b.pdf

[16] G. Fursin et al. Collective optimization: a practical collaborative approach, In *ACM transaction on architecture and code optimization (TACO)*, col. 7, no. 4, 2010.

[17] K.kHoste and L. Eeckhout Automated just-in-time compiler tuning, In *International symposium on code generation and Optimization (CGO)*, 2010.

[18] K. Hoste and L. Eeckhout Characterizing the Unique and Diverse Behaviors in Existing and Emerging General-Purpose and Domain-Specific Benchmark Suites, In *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2008.

[19] M. W. Benabderrahmane et al. The polyhedral model is more widely applicable than you think, In *ETAPS international conference on compiler construction (ETAPS-CC)*, pp. 283-303, 2010.

[20] H. Leather et al. Automatic feature generation for machine learning based optimizing compilation, In *International symposium on code generation and optimization (CGO)*, 2009.

[21] G. Pekhimenko and A. D. Brown. Efficient program compilation through machine learning techniques, In *International workshop on automatic performance tuning (iWAPT)*, 2009.

[22] K. Hoste and L. Eeckhout. COLE: compiler optimization level exploration, In *International symposium on code generation and optimization (CGO)*, 2008.

[23] U. Bondhugula et al. A practical automatic polyhedral parallelizer and locality optimizer, In *International conference on programming language design and implementation (PLDI)*, 2008.

[24] L. J. P. van der Maaten and G.E. Hinton. Visualizing high-dimensional data using t-SNE, In *Journal of Machine Learning Research*, pp 2579-2605, vol. 9, 2008.

[25] K. Hoste and L. Eeckhout. Microarchitecture-independent workload characterization, In *International Symposium on Microarchitecture (MICRO)*, vol. 27-3, pp. 63-72, 2007.

[26] C. M. Bishop. In *Pattern recognition and machine learning*, Springer, 2006.

[27] F. Agakov et al. Using machine learning to focus iterative optimization, In *International symposium on code generation and optimization (CGO)*, pp. 295-305, 2006.

[28] M. Stephenson and S. Amarasinghe. Predicting unroll factors using supervised classification, In *International symposium on code generation and optimization*, pp. 123-134, 2005.

[29] R. Allen and K. Kennedy. In *Optimizing compilers for modern architectures*, Morgan Kaufmann, 2002.

[30] L. Breiman. Statistical modeling: the two cultures, In *Statistical Science*, vol. 16, pp. 199-231, 2001.