Arnaldo J. Cruz-Ayoroa
Department of Informatics ISEE, Kyushu University, Japan

Compiler optimization space exploration using machine learning techniques

Computer programs are usually coded in a high-level language that is amenable to humans but unsuitable for execution. A compiler is a program used to translate the high-level code into a form that can be executed by the machine. The compilation process is composed of several translation phases and in each phase the code transformations applied will affect program performance. Therefore an essential task of the compiler is how to transform the program in a way that optimizes, or makes best use, of the hardware resources. We call an optimization scenario the sequence of transformations and their parameters that are applied to a program to improve its performance. Applying the correct scenario may improve performance several times-fold. On the other hand, applying the incorrect scenario may not change the performance in the best case, or will have a significant detrimental impact in the worst case. The capacity for the compiler to distinguish between both cases is now more important than ever given than single-threaded performance is stalling[1].

In order to improve performance, compilers have traditionally employed a small set of fixed optimization scenarios regardless of the input program. In addition, optimizations have a set of parameters that control the manner in which they are applied. These parameters are selected based on heuristics. In short, heuristics try to make best guesses based on rule-of-thumb and hand-crafted models of the target system. However, the fixed scenarios and heuristics approach has proven to be inefficient as programs, compilers and hardware become more complex. For example, we conducted an experiment to determine Intel's ICC, GCC and LLVM capabilities at generating profitable vector instructions. This *single* optimization, which is called automatic vectorization, resulted in slow-down for 26%, 26%, and 75% of our benchmarks, respectively.

The research objective of this thesis project is to study how to select optimization scenarios and optimization parameters that are appropriate for a given input program, such that its execution time is reduced. Because of the large number of optimization scenarios, it is not practical compile and run every program to select the best alternative. Thus we need a smart way to identify those scenarios with greatest potential for speedup. This we call optimization space exploration (OSE). The hypothesis is that using OSE would yield much better performance that the current techniques utilized by modern compilers. Concretely, we want to study how to construct accurate models that map input program characteristics to one or more optimization scenarios.

Our methodology for OSE uses machine learning models. A machine learning (ML) model is trained with previous experiences in order to predict future outcomes. In the compiler domain, the previous experience consists of a program characterization, an optimization scenario and the resulting speedup. The model is trained with these experiences to predict which scenario to apply for a new input program. Ideally, the more experiences are used during training, the more accurate the model will be in outputting a good optimization scenario.

There are three central challenges in using ML that are explored in this work; characterization of the model's input, generation of the training set, and validation of the models.

1 Peter Kogge et al. ExaScale Computing Study: Technology Challenges in Achieving Exascale Systems. 2008. http://users.ece.gatech.edu/~mrichard/ExascaleComputingStudyReports/exascal e_final_report_100208.pdf.

Characterization refers to the way in which the input is encoded so that accurate predictions can be obtained. This entails identifying relevant features of a program that can capture its behavior. In our case we consider different high-level static software characteristics (SSC). That is, we predict using sets of program characteristics that are extracted from high and intermediate level representations of the program. These characteristics do not require compilation or dynamic profiling. We also study how relevant are the SSC in characterizing the input program and compare the prediction accuracy when using the program characteristics of other works, such as the GCC Milepost project (ctuning.org).

The second challenge in ML is generating the training set. A good quality training set has enough samples covering the program characteristics and performance spaces. In other words, we need a set of optimization scenarios and benchmarks that respond with both speed-ups and speed-downs depending on which scenario is applied. For benchmark selection, the common method used in other works is to gather training examples from popular benchmarks. We took an alternative approach and created benchmark synthesizers. This allows generating many benchmarks with a better coverage of the program characteristics and performance spaces. Furthermore, because we have a better understanding of the program behavior, we can attain better insight as to which program characteristics are important and why programs respond to scenarios in a certain way. To the synthesized benchmarks we apply optimization scenarios that contain not only compiler flags, but also high-level source-to-source transformations to consider a wider variety of scenarios.

The third challenge in ML is validating the models. A model is evaluated based on its accuracy, or how close its predictions are to the true values. However, the accuracy of a trained model may be deceptive; high accuracy numbers do not imply that the model can be generalized to other kinds of input programs. Causes include self-validation, in which similar examples are not removed from the training set when cross-validating, and even program bugs in the experimentation scripts. To prevent this situation we are testing against non-synthetic benchmarks, comparing our results against models that output random scenarios and also models trained with random input characteristics.

The next phase of research is focused on synthesizing more complex benchmarks. Initially we considered realistic, yet simple nested loops. Now we want to be able to generate more complex programs to train models that can predict beneficial scenarios for popular benchmarks such as SPEC. Important questions that arise include what types of benchmarks should be generated, should data sets be considered in the program characterization, and which distributions to use when generating the program features. Answering these questions will make the application of machine learning to compilation more practical.