Reducing Training Time in a One-shot Machine Learning-based Compiler

John Thomson[1], Michael O'Boyle[1], Grigori Fursin[2], Björn Franke[1]

[1] University of Edinburgh, UK john.thomson@ed.ac.uk,[mob:bfranke]@inf.ed.ac.uk
[2] INRIA Sarclay, France grigori.fursin@inria.fr

Abstract. Iterative compilation of applications has proved a popular and successful approach to achieving high performance. This however, is at the cost of many runs of the application. Machine learning based approaches overcome this at the expense of a large off-line training cost. This paper presents a new approach to dramatically reduce the training time of a machine learning based compiler. This is achieved by focusing on the programs which best characterize the optimization space. By using unsupervised clustering in the program feature space we are able to dramatically reduce the amount of time required to train a compiler. Furthermore, we are able to learn a model which dispenses with iterative search completely allowing integration within the normal program development cycle. We evaluated our clustering approach on the EEMBCv2 benchmark suite and show that we can reduce the number of training runs by more than a factor of 7. This translates into an average 1.14 speedup across the benchmark suite compared to the default highest optimization level.

1 Introduction

For many applications, particularly in the embedded and high performance domains, performance is critical. Iterative compilation has been successfully applied to such applications [10] precisely because of this emphasis on performance. However, this comes at a cost, namely the large number of compilations and evaluations needed. While this may be acceptable for the final version of a program, it is ill-suited to the program development cycle. Machine learning techniques [1, 6] have been proposed which have the potential to overcome this by moving the online search cost for a particular program to an off-line training cost. However, this training cost is large and must be performed whenever the underlying platform is modified.

This paper proposes a new technique to significantly reduce the training cost by more than a factor of seven, allowing high-quality, machine learning-based optimizing compilers to be produced for a new architecture in a fraction of the time. This is particularly important for embedded applications due to inherent performance and time-to-market constraints.

Previous work [10, 1, 6] has shown that it is possible to obtain considerable improvement in execution speed by applying learning techniques to the problem of optimization selection. However, the training time for the system is a significant cost, which can run to weeks, or even months. Additionally, this training time grows linearly with the number of benchmarks used for training, necessitating the use of small benchmark suites with limited scope. This lack of scope limits previous learning compilers in their ability to properly characterize new programs, meaning they frequently rely on additional search of the optimization space to achieve good performance[5]. This paper

1. Cluster
   (a) Extract features vector **f** from program *P*.
   (b) Reduce **f** to the most significant principal components using PCA.
   (c) Cluster the new feature-space $\mathbf{f}'$ into *n* clusters, giving the *n* most typical centroid programs for the space.
2. Train
   (a) Train a model using using randomly selected optimizations on the *n* clustered programs.
3. Deploy

Fig. 1. Cluster Algorithm

presents a scalable technique to allow vastly greater coverage of the space of programs, at a fraction of the cost of previous approaches.

Although previous work [14] has successfully produced machine-learning solutions which do not require search of the target program, they typically focus on just one optimization heuristic and are limited in scope. Those approaches that attempt to find good solutions over wide optimisation spaces [10] inevitably require some type of feedback via a small search of the target program's optimization space. By significantly reducing the cost of learning, we can cover a larger program space than before, resulting in one-shot compilation, with no feedback. This is particularly useful during program development time, when the user does not want to spend additional runs of his/her program to see what the eventual performance will be.

In our approach, each program is characterized by a set of code features describing its overall structure. These sets of features form a feature space. If we examine the structure of this space using *unsupervised learning*, we can select a subset of programs that are most representative of the space as a whole. This in turn reduces the number of programs that need to be evaluated during training. When applying such *clustering* to the EEMBC benchmark suite we can reduce the number of programs to consider to just 6.

Additionally, we show that by gaining large scale coverage of the whole space of programs in our training data, we can dispense with search altogether, and produce a simple-to-use, one-shot compiler that gives excellent performance, giving an average speedup of 1.14 over O3 level across the benchmarks suite. Further, when trained in this way, we have knowledge of how well our compiler is trained in a particular domain of programs. When a new program is to be compiled, we can judge how confident we are that the new program is covered by our previous training, or whether retraining would be beneficial.

The remainder of the paper is structured as follows: Section 2 gives a description of our clustering approach. Section 3 describes the experimental methodology used, while section 4 presents our experimental results. After a short review of related work we finally draw some brief concluding remarks.

2 The Cluster-based Approach

Figure 1 gives an overview of our approach. First off, we have to extract the features from each of the programs which are potential candidates for training. This feature extraction is a way of representing the essence of a program for later classification. These features are represented as a vector of values, many of which are redundant. We then

apply principal components analysis which determines those features that are useful and those that are redundant. Once we have a have a reduced feature vector describing each program, we then attempt to group them using clustering to determine the most representative programs that span the space. This is the end of the initial clustering stage. The next stages are standard supervised learning and deployment. Here we learn a model based on training data associated with the *n* selected programs.

2.1 Features and Feature Reduction

Program features are vectors of numerical values which characterize the space of programs. Machine learning techniques can use features as a metric to gauge similarity. The features used in this paper are the ratios of each assembly level instruction to the total number of instructions executed by a program, i.e. the proportions of each type of instruction used. This is a very simple feature set, which is easy to capture, and yet, as we later show in section 4, provides excellent performance. We use ARM assembly instructions as our features to ensure that our technique is properly capturing information about the program and not some facet peculiar to a particular architecture. Hoste and Eeckhout [9] argue that the use of a generic RISC architecture is capable of representing and characterizing program performance and the ARM is used as an approximation a generic RISC core.

Features are extracted statically using a simple fast profiling tool, which counts the number and type of instructions used by the program, and predicts unbounded values. These raw counts are normalized and used our basic program features. The output of this stage is a 30-element feature vector describing the program.

These extracted features can be reduced in number, using a technique called *Principal Components Analysis* [4]. This technique reduces the dimensionality of the feature space by examining the variance within the data, and while preserving as much variance as possible, generating a new set of features which are a linear combination of the original set.

2.2 Clustering

The feature-space of programs is clustered using the a fuzzy clustering method called the *GustafsonKessel algorithm* [8], which is a variation on a standard C-means fuzzy clustering algorithm [4] which allows the detection of different geometrical shapes in one data set. The algorithm minimizes the objective function so that the fuzzy distances between the data points and cluster centers are minimised. The input to the algorithm was the reduced feature set of 9 principal components, as described above. Our GK clustering technique cannot determine the correct number of clusters which most accurately depict the space, which must be supplied *a priori*. We employed the technique suggested by Ray and Turi [13] which considers the proportion of the intra-cluster variance in respect to the inter-cluster variance, and selecting the first local minimum of this value K as the number of considered clusters increases.

2.3 Training and Deployment

Once we have selected the benchmarks we wish to train with, we use standard approaches to learn a model. This is achieved by applying random optimization to each of the programs to find the best optimization settings. We then need to build a model which maps the program features to the best optimizations found. There are many approaches

to building such a model. Since the goal of our approach is one-shot compilation, we simply record the best optimization flags found for our training programs and use a nearest neighbor model [1].

Having fixed the selected programs and completed the training for each, the compiler is ready for deployment. Firstly, feature extraction is performed on the new program input, and those features are then input to a nearest neighbor classifier which determines which of the *n* centroids it is most near using the squared Mahalanobis distance norm. Having been assigned a neighbor, the benchmark is compiled and executed, using the best performing compiler flags associated with that point, and the execution time recorded. Although this is simple model, it is later shown to be an effective one.

3 Experimental Methodology

We evaluated each approach on the EEMBCv2 [7] benchmark suite, which is targetted at the embedded domain. It contains many computationally intensive kernels and programs utilized in the embedded and general purpose worlds. A few EEMBC programs were excluded due to difficulties with GCC. When a choice of dataset was offered by EEMBC, the default dataset was chosen. These experiments were carried out on an Intel Core 2 Duo E6750 processor running at 2.66GHz. The machine was running a stripped down version of Ubuntu Linux 8.04 with linux kernel version 2.6.24. GCC compiler version 4.2.2 was used which allows additional optimizations over and above the default GCC to be accessed via compiler flags. We selected 88 of the different flags supported by GCC which formed our optimization space. By setting each flag to a particular value, we can evaluate many distinct optimizations

Cluster Approach

On applying our technique to the EEMBC benchmark suite, we found the correct number of clusters to be 6. The centroid programs were selected giving the 6 most typical programs to represent those clusters. These 6 programs were then used to train on. For each training program, we probe the optimization space by selecting 4000 random flag settings and executing the resulting code. The best performing flag setting is recorded and used by the nearest neighbor model. We use standard leave-one-out cross-validation [4] both for clustering and deployment. This means we exclude the benchmark being evaluated from the clustering process so that that the program has never been seen by the compiler before.

Standard Random Training Selection Given a limited amount of time and resources is available to train a machine learning compiler, the standard way to select programs to train on, is simply to use random selection. For a fair comparison 6 benchmarks were randomly chosen from the set of 44. Having selected the programs to be trained, the process proceeds in the same way as the clustering approach.

Given that there may be high variation in performance depending on the exact 6 programs selected, we repeated this random selection 1000 times to give a robust mean performance. This should minimise the effect of randomly choosing particularly good or bad training programs.

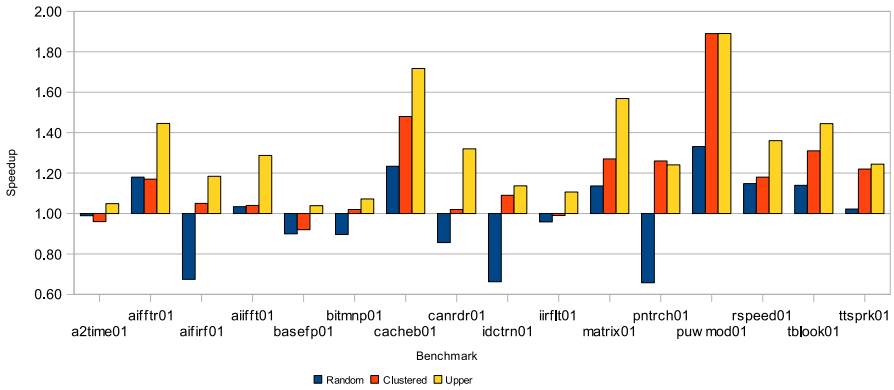

Fig. 2. Speedups obtained for EEMBCv2 for random selection, cluster selection, and the likely upper bound. Average is over all graphs: 1 of 3

Generating the upper bound Finally we need a measure of how much of the available performance we actually achieve using a machine learned model. We randomly applied 4000 different optimizations to a program and recorded the best execution time. It is unlikely that a model that predicts a good optimization setting without any feedback information will outperform the best of 4000 runs, so this provides a reasonable upper bound limit.

4 Results

This section evaluates our approach in terms of the performance gained on the EEMBC benchmark suite. To give a useful comparison, we show a reasonable upper bound for performance for each program and also compare our approach against standard uniform selection of training data.

Figures 2 to 4 show the performance of the 3 schemes described in section 3 on the benchmarks. The x-axis in each figure is simply the name of each of the 44 EEMBC benchmarks, while the y-axis shows the speedup relative to O3 of each approach. The first black bar shows the results from a machine-learning model when the training data is randomly selected. The middle dark grey bar shows the results of using the same model with our cluster based selection of training data. The final light gray bar shows the performance of an iterative search of that program's optimization space and represents the best that can be found after trying 4000 different optimization settings.

Uniform random training Consider the results of the first dark bar labeled `Random`. Here 6 random programs have been used to train the nearest neighbor model. The average speedup across the benchmarks it achieves is shown in the last set of bars in Fig 4. Using random training data leads to just a 1.03 speedup on average for the nearest neighbor model. Although it is able to determine significant performance improvements of up to 1.4 on `mp3player`, in 23 out of the 44 programs it actually causes a slowdown of upto 0.7 in 3 cases `airfirrf01, idctrn01, pntrch01`. This shows that although machine-learning techniques can improve performance, with a limited training budget, it is difficult to learn a model that performs well across the program space.

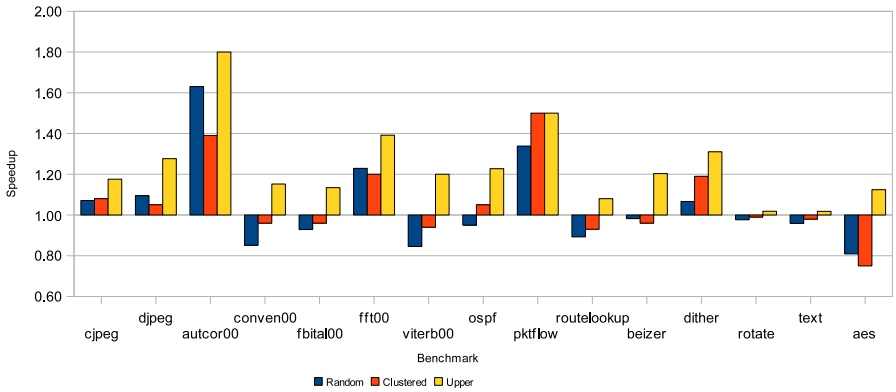

Fig. 3. Speedups obtained for EEMBCv2 for random selection, cluster selection, and the likely upper bound. Average is over all graphs: 2 of 3

Our clustering approach The light colored bars labelled `Clustered` in Figures 2 to 4 show the results of the same modeling approach with the same training budget of 6 programs. This time, however, the training programs have been selected using our clustering approach. As again the final set of bars in figure 4 show the average performance. On average, using just one evaluation, our clustering-based approach yields a speedup of 1.14 over the whole benchmark suite. This compares to a speedup of only 1.03 if a machine leaned compiler uses programs selected at random. This is a dramatic improvement over the standard approach given that both use exactly the same model and training budget. Our approach is able to avoid the large slowdowns on `airfirrf01`, `idctrn01`, `pntrch01` and achieves speedups of 1.5 on `cacheb01`, 1.9 on `puwmod01` and 2.71 on `mp3player`. These speedups are likely to be due to the kernelized nature of these codes, where changing a small section of code which is frequently used can have a large impact on the resulting speedup.

Iterative Search: Upper bound The final light gray bar labelled `Upper` in Figures 2 to 4 describes the best performance achieved when trying 4000 different oprimisations on each program. It is unlikely that a machine-learning based one-shot compiler could ever outperfrom this, so it acts as a useful upper bound. In every case it is at least as good as O3, by definition, hence there are no slowdowns. For certain codes`a2time01`,`basefp01`,`ospf` it shows that there is little room for improvement hence the poor behavior of the learned models. In other such as `puwmodel01`, `mp3player`, `autocor00`, it shows there is significant room for improvement, which our learned cluster based model frequently achieves. If we once again look at the average values, we see that it can can achieve on average a 1.28 times speedup. This shows the large potential performance available if there is sufficient time to tune each program. However, on average, our approach can achieve half of the performance improvement (1.14 vs 1.28) attained by iterative optimization using 4000 runs, in just a single evaluation. Additionally, we achieve a factor 7 increase in the additional optimization possible by using our clustering-based approach rather than standard random selection, which shows itself not to be a viable option when no search of the space is allowed.

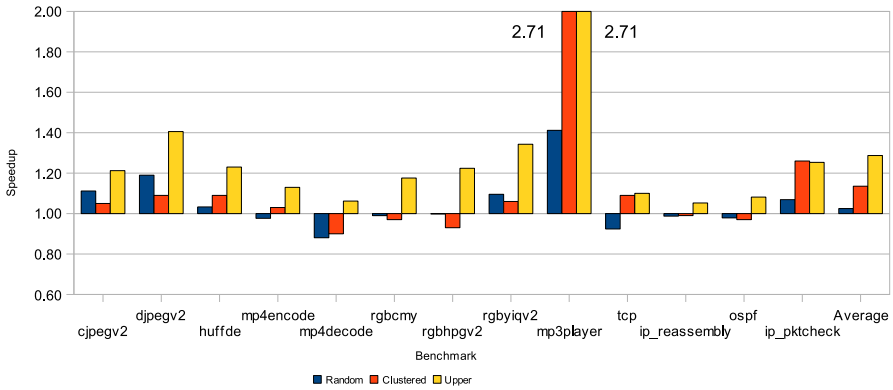

Fig. 4. Speedups obtained for EEMBCv2 for random selection, cluster selection, and the likely upper bound. Average is over all graphs: 2 of 3

5 Related Work

Most machine-learning based compiler techniques require some profiling or search. However, machine learning have been used previously to predict the most effective optimizations to apply, but in a heavily constrained environment without profiling. Stephenson et al. [14] examine the problem of parameter selection for loop unrolling. These papers consider a limited optimization space, consisting of either binary decisions or just one optimisation, which limits their scope for code improvement.

Intelligent search-based techniques can be thought of as a specialized example of online supervised learning, in which the search strategy is updated during the search. They traverse an optimisation space, evaluating points in that space and attempting to find the best result. In this case, compiler transformations are evaluated. The space can be searched, and if structure can be observed, then previous results can be used to determine where in the space is most profitable to search. Examples of this are *hill-climbers* and *greedy algorithms* [6]. Iterative techniques have progressed from simple random searching to evolutionary selection techniques, based on fitness [10] and stocastic search techniques [6]. Complimentary techniques employing probabilistic search have also been proposed and models used to speed up the search process [1].

Berube et al. [3] use clustering as a means to reduce workload as inputs vary in profile-directed compilation. Clustering profile data allows the authors to characterize the input data and specialize code transformation for different datasets. This work does not provide a baseline comparison and therefore it is hard to say if the clustering technique is better than a naïve selection process. Additionally, the work is intended to assist in estimating the performance of the benchmarks evaluated in the paper, and it is not possible to gain a similar benchmark subset on a different benchmark suite, without running the whole suite through a slow simulator and profiler.

6 Conclusion

We have demonstrated that, by clustering in the feature-space, we can dramatically reduce the amount of training required to achieve good performance by more than a factor of 7 when using a machine-learning compiler. By carefully selecting the training data

to be used, we can better characterize the program-space with a small number of points, rather than randomly selecting them. In addition, we have shown that a compiler trained in this way gives an average of 1.14 speedup on the EEMBCv2 benchmark suite over the O3 baseline in just one evaluation. This was achieved by training on just 6 programs. Finally, we have shown that instruction ratios are effective features for clustering.

References

[1] F. Agakov, E. Bonilla, J. Cavazos, B. Franke, G. Fursin, Michael O'Boyle, John Thomson, Marc Toussaint, and Chris Williams. Proceedings of the 4th Annual International Symposium on Code Generation and Optimization (CGO), New York, March 2006.

[2] R. Babuska, P.J. van der Veen, and U. Kaymak. Improved covariances estimation for Gustafson-Kessel clustering. IEEE International Conference on Fuzzy Systems, 2002.

[3] Paul Berube, Jose Nelson Amaral, Rayson Ho and Raul Silvera *Workload Reduction for Multi-input Profile Directed Optimization*, Proceedings of the 7th Annual International Symposium on Code Generation and Optimization (CGO) 2009.

[4] C. Bishop, *Neural Networks for Pattern Recognition*, OUP, 2005

[5] J. Cavazos, G. Fursin, F. Agakov, E. Bonilla, M. O'Boyle and O. Temam. Rapidly Selecting Good Compiler Optimizations using Performance Counters. CGO March 2007

[6] K. D. Cooper, A. Grosul, T.J. Harvey, S. Reeves, D. Subramanian, L. Torzon, and T. Waterman *Exploring the Structure of the Space of Compilation Sequences Using Randomized Search Algorithms* In Proceedings of the 2004 LACSI Symposium, 2004.

[7] EEMBCv2 benchmark suite. http://www.eembc.org/

[8] D.E. Gustafson and W.C. Kessel. *Fuzzy clustering with fuzzy covariance matrix*. In Proceedings of the IEEE CDC, San Diego, pages 761766. 1979.

[9] Kenneth Hoste and Lieven Eeckhout, *Comparing Benchmarks Using Key Microarchitecture-Independent Characteristics*, IISWC, pp. 83-92, 2006.

[10] P. Kulkarni, W. Zhao, H. Moon, K. Cho, D. Whalley, J. Davidson, M. Bailey, Y. Park, and K. Gallivan. *Finding effective optimization phase sequences.* In ACM LCTES, 2003.

[11] P. Kulkarni, D. Whalley, G. Tyson, and J. Davidson, *Evaluating Heuristic Optimization Phase Order Search Algorithms* published in the IEEE/ACM International Symposium on Code Generation and Optimization (CGO '07), pp. 157-169, March 2007

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

[13] S. Ray and R. Turi, *Determination of number of clusters in k-means clustering and application in colour image segmentation*, In Proceedings of the 4th International Conference on Advances in Pattern Recognition and Digital Techniques, pp. 137–143, 1999.

[14] M. Stephenson, S. Amarasinghe, M. Martin and U-M. O'Reilly *Meta Optimization: Improving Compiler Heuristics with Machine Learning* In PLDI 2003.