An Evaluation of Different Modeling Techniques for Iterative Compilation

Eunjung Park, Sameer Kulkarni, and John Cavazos

Department of Computer and Information Sciences
University of Delaware
{epark,skulkarn,cavazos}@cis.udel.edu

Abstract. Choosing the right set of optimizations can make a significant difference in the running time of a program. However, compilers typically have a large number of optimizations to choose from, making it impossible to iterate over a significant fraction of the entire optimization search space. Recent research has proposed "intelligently" iterating over the optimization search space using predictive methods. An important step in developing predictive methods for compilation is deciding how to model the problem of choosing the right optimizations. In particular, state-the-art methods in iterative compilation techniques use characteristics of the code being optimized to predict good optimization sequences to evaluate.
In this paper, we evaluate three different ways of modeling the problem of choosing the right optimization sequences using machine learning techniques. We evaluate a novel prediction modeling technique, namely a tournament predictor, that is able to effectively predict good optimization sequences. We show that our tournament predictor out-performs current state-of-the-art predictors and out-performs the most aggressive setting of the Open64 compiler `-OFast` on an average by 76% in just 10 iterations.

Keywords: compiler optimization, iterative compilation, machine learning, regression

1 Introduction

Most applications can greatly benefit from a fine-tuned set of compiler optimization sequences. However, finding the right set of optimizations for each application is a non-trivial task since the search space is extremely large. Recent research has focused on intelligent search space exploration, in order to efficiently search for the right optimization sequence [3,4,6,8,9,13,15,18,26]. In this previous work, models are developed automatically using machine learning or statistical techniques to predict good optimizations to apply based on characteristics of the code being optimized. The advantage of these approaches is that they use characteristics (or features) of the code and therefore a model has the potential to predict optimizations that are specifically tailored to the code.

An important step in using machine learning or statistical techniques to choose optimizations to apply is how to model the problem. That is, when the optimizer needs to make a decision, certain factors (the features) are inputs to a decision function, whose output selects a choice from two or more optimization possibilities. One way to phrase the optimization problem is to develop a decision function that given a set of characteristics of a program, chooses whether an optimization should be applied or not. We can develop several of these decision functions each of which controls a specific optimization. Using these decision functions, we can construct a sequence of optimizations to apply to the code being optimized. A key property is framing the optimization decision problem so that the function to be learned is as simple as possible.

This paper evaluates different ways of modeling the problem of finding good sets of optimizations for code being optimized. We use a large set of kernels which provides good coverage of a wide range of scientific applications. To evaluate our models on a reasonable fraction of the optimization search space, we used random search to generate a set of optimization sequences. This set of randomly generated optimization sequences is used to train and evaluate our models.

We propose a new model to predict good optimization sequences to try. We call this new model the *tournament predictor* and compared this predictor to two current state-of-the-art prediction models. In total, we evaluate three different ways of modeling the problem of choosing good optimization sequences to apply to a program. For each of these prediction models, we use performance counters to characterize the dynamic behavior of a program. In recent work, performance counters have been shown to be better at characterizing applications than static code features [6]. We collected performance counters for kernels and evaluated those kernels with randomly selected optimization sequences. We generated our proposed modeling techniques using regression technique with the performance counter program features, then evaluated the generated models using leave-one-out cross validation on our kernels.

The rest of the paper is organized as follows. Section 2 describes the problem of developing predictors for intelligent modeling and describes the different predictors we evaluate. Section 3 gives an overview of our solution, especially Section 3.1 describes each prediction models used in this paper and how we construct them. Section 4 explains the characteristics of the programs used in this paper. Section 5 lists the characteristics of each of the testbeds (both hardware and software) that were used. Section 6 shows the results for each of the testbeds used in our program. Section 7 lists and explains related work and their main contributions and differences with our models. Section 8 presents our conclusions and future work.

2 Learning to Optimize

Recent work has shown that iterative compilation applied to certain programs can achieve significant benefits over the highest setting available in a compiler. However, many of the proposed techniques for exploring optimizations (e.g., genetic algorithms [3], random search [19], statistical techniques [15], or exhaustive search [18]) are expensive which limits their practical use. This has led compiler researchers to propose using “intelligent” prediction models that focus exploration to beneficial areas of the optimization search space [2, 6, 10, 12, 20, 27]. Prediction models can reduce the cost of finding good optimizations, but increase complexity in the design of the search function because models require characterizing the program being optimized (e.g., with source code features or performance counters), generation of training data, and a training phase. One important step in designing the prediction model is how to phrase the problem of choosing good optimizations for a program. Several ways of modeling the problem of finding good optimizations have been proposed in the literature, but there has been little effort on evaluating these different methods. In this paper, we show this step can have a significant impact on code being optimized. Section 2.1 describes three different prediction models that we evaluate in this paper. In Section 2.2, we presents preliminary experiment results showing the potential of our new predictor, tournament predictors.

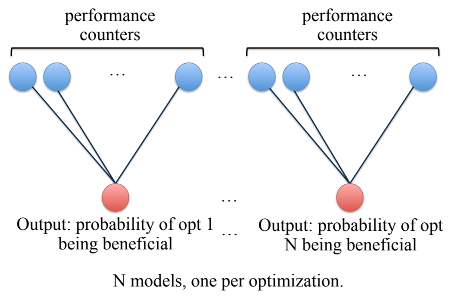
2.1 Modeling the Optimization Problem

A general formulation of the problem is to construct a search function that takes a characterization of a program being optimized as input and generates a set of one or more optimization sequences to evaluate as output (either implicitly or explicitly). However, there are several ways to specifically model this problem. In this paper, we evaluate three different methods of modeling the problem of finding good optimization sequences, which we name the *sequence predictor*, the *speedup predictor*, and the *tournament predictor*. Figure 1 depicts the models that we evaluate in this paper which we describe in more detail below. We briefly describe each of these predictors here and in more detail in Section 3.1.

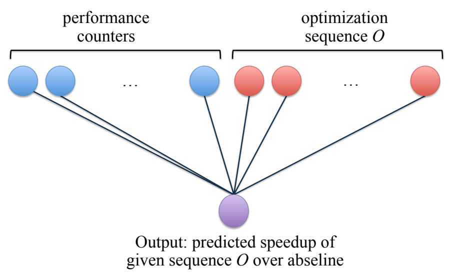
Sequence Predictor Previous work [6, 12] has proposed to model the problem by characterizing a program using performance counters and generating an optimization sequence that will benefit the program. The performance counter characterization serves as input to a model, and the model predicts a probability distribution of optimizations to apply to that program. We term this model a *sequence predictor* because it can be used to construct a sequence of optimizations.

Speedup Predictor Another recent model that has been proposed [5,11] takes as input both the characterization of the program being compiled and an optimization sequence, and it predicts as output the speedup of that optimization sequence relative to a default optimization setting. We refer to this model as the *speedup predictor.*

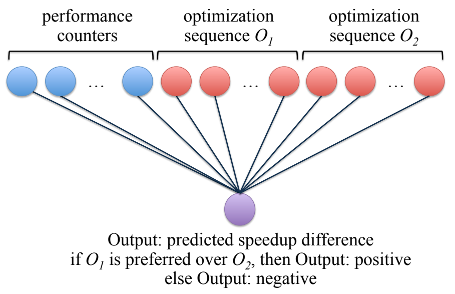
Tournament Predictor Finally, we propose a new method of choosing optimization sequences, the *tournament predictor* which takes as input a triple


performance counters
performance counters
Output: probability of opt 1 being beneficial
Output: probability of opt N being beneficial
N models, one per optimization.

(a) Sequence Predictor.



(b) Speedup Predictor.



(c) Tournament Predictor.

Fig. 1. Three prediction models that are evaluated in this paper.

corresponding to the characterization of the program and two optimization sequences. This model predicts whether the speedup of the first optimization sequence will be more or less than the second optimization sequence. We can use this predictor to provide an ordering of a set of optimization sequences to be used for iterative compilation.

2.2 Preliminary Experiment of the Speedup and Tournament Predictors

This section describes an experiment that shows our speedup and tournament predictors are able to capture optimization speedup trends and therefore have the potential to be used for predicting good optimizations to use for iterative compilation. First, we generated a set of 200 randomly generated optimization sequences constructed from 63 optimizations available in the Open64 compiler suite. We evaluated these 200 optimization sequences on a set of 74 kernel benchmarks and obtained speedups relative to the most aggressive optimization level available in Open64 (`-Ofast`). We then performed leave-one-out cross validation to construct our speedup and tournament predictors, leaving out the benchmark `ATAX` (one of Polybench [24]) for testing, and training our models with the remaining kernels. A regression algorithm was used to build our models. Our training data consisted of performance counter characterization of our kernels and the optimization sequences from all but one (left out) kernel.

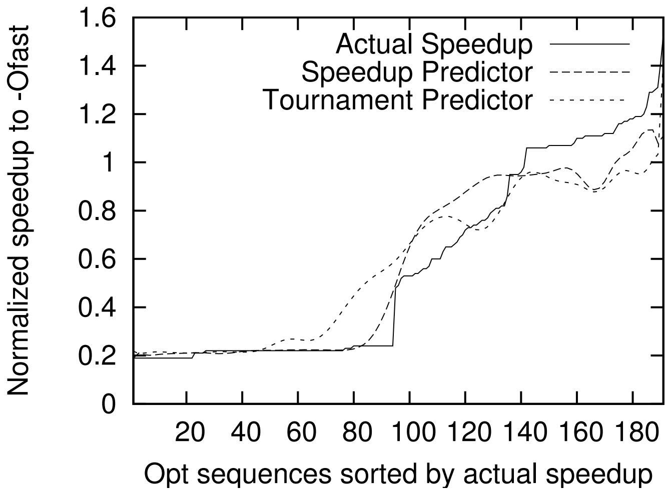


Fig. 2. Speedup Predictor and Tournament Predictor for ATAX

Figure 2 shows how predicted speedups with the proposed models compares to actual speedup for the one unseen `ATAX` kernel we used for our test. The y-axis is the speedup obtained after applying a sequence, and the x-axis is the optimization sequences sorted by increasing (actual) speedup. The solid line represents the actual speedup and two dotted lines show the speedup estimated by two prediction models. As can be seen, the speedup and tournament predictors can predict speedups for each sequences quite accurately. Therefore, using these models to choose a good optimization sequences for an unseen program has the potential to obtain significant improvements when used in iterative compilation.

3 Automatically constructing a model

This section describes details of how we trained our different models. We used the HPCToolkit [1] to extract the performance counters to characterize each program being optimized. Using HPCToolkit, we collected 29 performance counters to characterize each program. To collect the performance counter values, we compiled each program using the `-O0` optimization level of the Open64 compiler. The optimization level `-O0` was selected to minimize the effects of compiler optimizations on our performance counter characterization of the program. We then sampled our optimization search space by randomly generating 500 optimization sequences from 45 selected optimizations. The complete list of those optimizations that were used for our experiments are discussed in Section 4.2. We transformed each sequence to a vector t by using a technique similar to thermometer encoding to cover all optimizations whether they are taking binary value or numeric values. For example, for the ‘prefetch’ optimization in Open64, we have 4 different possible values which are 0 through 3. We used '0 0 0' for 0, '1 0 0' for 1, '1 1 0' for 2, and '1 1 1' for 3. For optimizations taking binary values, we simply use 0 and 1 to signify the optimizaion is on and off, respectively. We used each of our 500 randomly generated optimization sequences to compile and execute the programs. We then ran each compiled version five times and used the average running time. Once average running time is obtained, we can calculate speedup over `-Ofast` as follows:

$$speedup = \frac{t_{Ofast}}{t_{seq}}$$

where t_{Ofast} is the average time taken by the program when compiled with `-Ofast`, and t_{seq} is the average time taken by the program when compiled with a given optimization sequence. This speedup is used to train our models either directly, e.g., with our speedup predictor, or indirectly, e.g., with our tournament or sequence predictors.

3.1 Prediction Models

There are specific differences as to how the training and test sets are used and represented depending on the type of the training model being developed. In this section, we describe the three different modeling techniques that we evaluated for iterative compilation.

Sequence Predictor We obtained a bit vector with size 67 bits, and we construct separate predictor for each bit. Thus this model is a collection of 67 different models where each model is trained to predict whether a specific bit should be 1 or 0. The input to each model is the performance counter characterization of a program, and the output of each model predicts a probability that a particular optimization (associated with that output) will benefit the program. Optimization sequences can be generated from this model by sampling at the mode of the distribution of each model's output. The training data for this model consists of one sequence per benchmark with the best speedup. Once the predictor is trained, we can use it to predict good sequences to apply to an "unseen" program by feeding as input the performance counter characterization of the program to the model. The model then outputs a probability p_i for each bit predicting whether the bit should be 1 or 0. For optimizations taking binary values, we simply decide whether we turn on or off a given optimization. For ones taking more than 2 possibilities, we select a bit with the highest probability among bits representing a specific optimization and decide which numerical value we use for that optimization. Thus, all the outputs from a probability distribution which we can then sample from to generate an optimization sequence to apply. This model can be used to generate multiple optimization sequences by sampling as many times as we wish.

Speedup Predictor For this predictor, the model is trained to accurately predict the speedup of the program relative to the program compiled by using `-Ofast`. The input to the model is the performance counter characterization of a program and a bit vector describing the optimization sequence, and the output from the model is the predicted speedup of that sequence relative to `-Ofast`. The training data for this model consists of all 500 optimization sequences applied to each kernel and the speedups obtained for each sequence relative to `-Ofast`. We use this predictor to predict good optimization sequences for "unseen" program, by evaluating sequences in order based on their predicted speedup. Another way of evaluating the quality of this predictor is to use it to search for good optimization sequences that were "unseen." In other words, we can evaluate this predictor in terms of how well it predicts the speedup of sequences that were not used to train our model. We show that this predictor is effective at finding good sequences for both "unseen" programs.

Tournament Predictor This model was trained to predict the better of two different optimization sequences presented to the model. The inputs supplied to this model were the performance counters for a program and two optimization sequences that could be applied to the program. The output of the model is either positive or negative depending on whether the model predicts the first optimization sequence is better or worse than the second sequence, respectively. Since it predicts speedup difference of two optimization sequences, not only does the model predict which optimization sequence is better, but it also predicts how big is the difference. One can view this as learning a relation over triples (PC, O_i, O_j), where PC is the characterization of the program being optimized and O is the set of optimization sequences from which the selection is to be made. Those triples that belong to the relation define pairwise preferences in which the first optimization sequence is considered preferable to the second. Each triple that does not belong to the relation represents a pair in which the first optimization is not better than the second. The process of selecting the best of the optimization alternatives is like finding the maximum of a list of numbers. We keep track of the current best optimization sequence, and proceed with pairwise comparisons, always keeping the better of two sequences being compared. This model can be used to generate sequences to try for a new program, by sorting different optimization sequences based on the results of the pairwise comparisons.

4 Program Characteristics and Optimizations

This section describes how we characterize programs. Section 4.1 describes the performance counters we collected to characterize programs, and Section 4.2 describes the optimization space we selected to construct our sequences from our testbed compiler.

4.1 Performance Counter Characterization

Each of our models predicts optimizations to apply to "unseen" programs that were not used in training the model. To do this, we need to feed as input to our models a characterization of the "unseen" program. We use performance counters to collect dynamic features that describe the runtime behavior of a program. Models using performance counter characteristics of programs have been shown to out-perform models that use only static code features of program [6]. We used 29 different performance counters shown in Table 1.

Category of PCs	List of PCs selected
Branch Related	BR-CN, BR-INS, BR-MSP, BR-NTK, BR-TKN
Cache Line Access	CA-SHR
Level 1 Cache	DCA, DCM, ICA, LDM, STM, TCM
Level 2 Cache	DCR, DCW, ICA, STM, TCA, TCM, TCW
Floating Point	FDV-INS, FML-INS, FP-INS, FP-OPS
Interrupt/Stall	RES-STL
TLB	TLB-DM
Total Cycle or Insts.	TOT-CYC, TOT-IIS, TOT-INS
Vector/SIMD	VEC-INS

Table 1. Performance Counters: We collected 29 different performance counters to characterize a program

4.2 Optimization Space

We selected five optimization phases from the Open64 compiler and from these phases we selected 45 individual optimizations (shown in Table 2). These 45 optimizations make up the optimization space we will explore with our models. Most optimizations came from global and loop nest optimization phases because these optimizations have the most potential to obtain significant running time improvements.

For our 45 optimizations, we generated a set of 500 random optimization sequences. We then evaluated each sequence on the programs in our training set, and measured their speedup relative to `-Ofast`, the most aggressive optimization level available in Open64 compiler. The random optimization sequence and its corresponding speedup are used as training data.

5 Experimental Setup

This section briefly describes the experimental setup. First, we describe the hardware platform, OS, and optimizing compiler. Second, we describe the benchmarks and the optimizations we used for our study.

Optimization Phase	List of Optimizations
LNO	blocking-size, cs1, cs2, fission, full-unroll, fusion, interchange, ou-prod-max, pf2, prefetch, prefetch-ahead, simd, trip-count
WOPT	aggcm-threshold, aggstr, canon-expr, combine, dce-aggressive, iv-elimination, spre, value-numbering
OPT	alias, align-padding, div-split, goto, ptr-opt, swp, unroll-size, unroll-times-max
CG	cflow, local-sched-alg, ptr-load-use, use-prefetchnta
GRA	optimize-boundary, prioritize-by-density
TENV	frame-pointer
IPA	callee-limit, ctype, dve, field-reorder, min-hotness, plimit, pu-reorder, small-pu, space

Table 2. List of 45 Optimizations: We selected 45 optimizations from 7 optimization phases in Open64 compiler.

5.1 Platform

We perform our experiments on a pair of Intel Quad CPU Q9650 machines, each with 2.0GHz processors with 8GB of memory, running Ubuntu Linux release 8.04. We use the HPCToolkit [1] along with latest PAPI 3.6 hardware counter library [21] to collect hardware performance counters for the benchmarks. Table 1 gives a brief description of the counters we used. We collect performance counters using level `-O0`, so that the characteristics of a benchmark are not masked by higher optimization levels (e.g., `-Ofast`). We used the open-source Open64 compiler version 4.2.1 [22], and all speedups reported are relative to `-Ofast`, the most aggressive optimization level available in this compiler. We used linear regression in Weka [16] 3.6.2 to build each prediction model.

5.2 Benchmarks

We decided to experiment with small pieces of code that would allow us to quickly evaluate different modeling techniques. We collected a large set of functions and kernels from various well-known benchmarks suites (e.g., UTDSP, NAS, Linpack, and Polybench) for our study. However, we noticed that several of these kernels had large variability for different runs of the same optimized code. In order to reduce this variability, we flushed the cache before every run to reduce possible cache interference. For the remaining code, we used the standard leave-one-out cross validation procedure to evaluate our models. That is, the models were trained using $N-1$ benchmarks and tested on the Nth benchmark that was left out.

6 Experimental Results

This section describes our experiment results that we conducted to evaluate our different modeling techniques. In Section 6.1, we discuss experiment results of three different modeling techniques by evaluating predicted optimization sequence for unseen programs with 10 evaluations. In Section 6.2, we discuss how two modeling techniques performs for up to 100 evaluations.

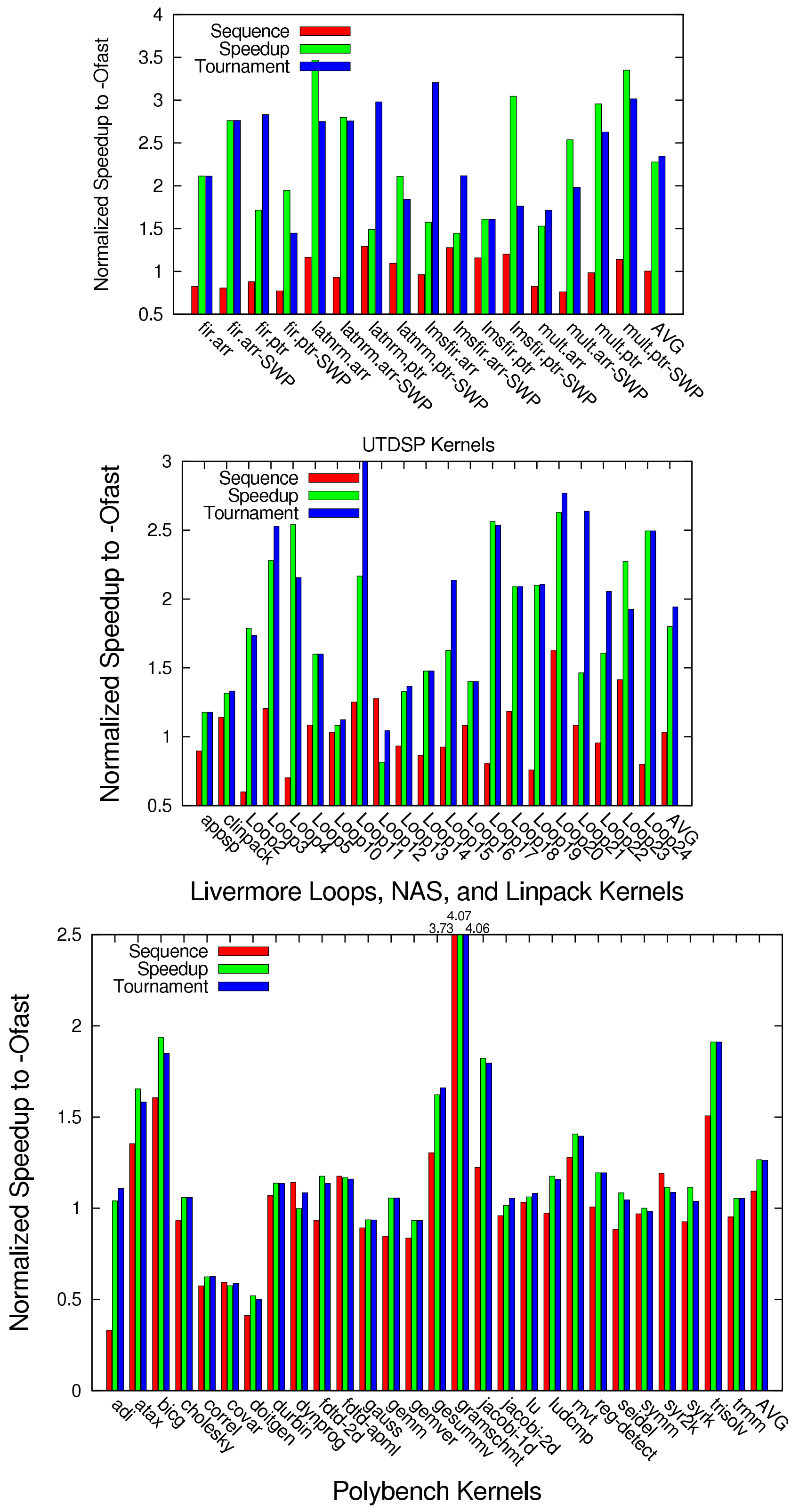


Fig. 3. This figure shows the maximum speedup obtained with 10 predictions for each predictor when trained using regression models. On average the sequence, speedup, and tournament predictors achieve 5%, 69%, 75% speedup respectively.

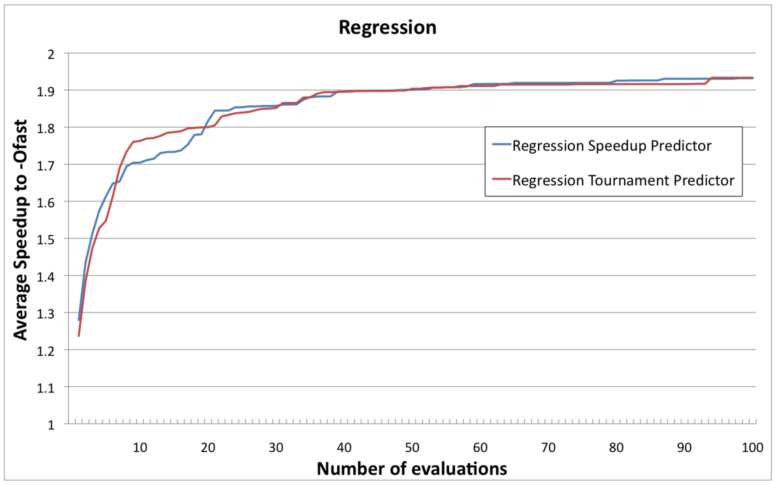


Fig. 4. This speedup of our tournament and speedup predictors (trained (from left to right) using Regression) averaged across all our kernels versus the number of optimization sequences evaluated.

6.1 Leave-One-Out Cross Validation on Kernels

To begin evaluating our modeling techniques, we used leave-one-out-cross validation over our set of kernels. This allowed us to evaluate the relative advantages between different predictor methods. Here, we used the same 500 optimization sequences for both training and testing. That is, we train a model on 500 optimization sequences applied to $N-1$ kernels, then we use the model to predict which of those same 500 sequences should be applied to the Nth program that was left out. We have recent results that show a model trained on one set of optimization sequences is effective at predicting the performance of "unseen" optimization sequences on an "unseen", but this is beyond the scope of this paper. Figure 3 shows the same results using regression to train each predictor. For this experiment, we have set the number of evaluations selected by each model to 10 evaluations. We found that regression performs well depending on the modeling techniques. The sequence predictor gave the least improvement of the three predictors evaluated, achieving a 6% on average.. The tournament predictor performs best achieving 76% average speedups across the kernels for regression. The speedup predictor also performs well, but slightly worse than the tournament predictor, by achieving 70%.

6.2 Performance Versus Number of Evaluations

While Figure 3 shows results for 10 evaluations, in this section we give a different view of how our two top predictors, namely the speedup and tournament predictors, perform as a function of the number of evaluations. Figure 4 shows the performance achieved averaged across all the kernels versus the number of sequences evaluated up to a maximum of 100 for the speedup and tournament predictors.

Both predictor reached to maximum possible speedup with similar curve. The speedup predictor achieves better speedup than the tournament predictor until 7 evaluations. The tournament predictor starts to perform better then the speedup predictor from 8 evaluations, and achieves 76speedup predictor to reach this speedup in 18 evaluations.

7 Related Work

A very innovative approach to iterative compilation was proposed by Parello *et al.* [23] where they tried to use performance counter at each stage of the tuning process to propose new optimization sequences. These sequences would be evaluated and based on the new performance counters they would choose new optimizations to try. They manually developed a decision tree to assist them in choosing which optimizations to apply based on performance counters. Even though this was a very systematic approach, the time to develop the decision tree was several weeks for each benchmark. Our technique does not need to generate performance counters during each iteration, but uses various past runs along with the initial performance counters to learn the best possible optimization sequence.

In the recent years a lot of research has shown the benefit of iterative compilation [8,9,13,15,17]. Iterative compilation has been shown to regularly outperform the most aggressive compilation settings of most commercial compilers and has been shown to be comparable to hand-optimized library functions [14,25,28,29].

Almagor *et al.* [3] take a more radical approach to reduce the total number of evaluations. They examine the structure of the search space, in particular the distribution of local minima relative to the global minima and devise new search based algorithms that outperform generic search techniques. Kulkarni *et al.* [18] introduced a system where they tried to use databases to store previously tested code and thus save on running time. They also disabled some optimizations that did not seem to improve the running time of the kernel. These techniques are only effective when programs are extremely small, such as those used in embedded domains. Cooper *et al.* [8] use genetic algorithms to solve the compilation phase-ordering problem. They were concerned with finding “good” compiler optimization sequences that reduced code size. Their technique was successful at reducing code size by as much as 40%. Unfortunately, their technique is application-specific. That is, a genetic algorithm has to *retrain* for each program to decide the best optimization sequence for that program.

Several researchers have also looked at using machine learning to construct heuristics that control a single optimization. Stephenson *et al.* [26] used genetic programming (GP) to tune heuristic priority functions for three compiler optimizations: hyper block selection, register allocation, and data prefetching within the Trimaran's IMPACT compiler. For two optimizations, hyperblock selection and data prefetching, they achieved significant improvements. However, a closer look at the results indicate that all the improvement was obtained from the initial population indicating that these two pre-existing heuristics were not well tuned. For the third optimization, register allocation, they were able to achieve on average only a 2% increase over the manually tuned heuristic.

In the work by Cavazos *et al.* [7] they used supervised learning to create a predictor model specialized to decide to enable or disable instruction scheduling. This reduced up to 75% of the scheduling effort without loosing any performance. Recently, Cavazos *et al.* [4] describe using static code features and supervised learning to control several optimizations to apply during method compilation in a JIT compiler. Since Java methods are typically small, static code features were successfully used to characterizing them.

For example, Lau *et al.* [19] present an online framework, called performance auditing, that allows the evaluation of the effectiveness of optimization decisions. The framework permits on line empirical optimization, which improves the ability of a dynamic compiler to increase the performance of optimizations while preventing performance degradations. Instead of using models to predict an optimization's performance, these approaches compile versions of the same method with different optimization settings chosen randomly. Then, they run each of these versions evaluating their performance empirically on the real machine. These approaches worked well for single optimizations or for a very small set of optimization sequences to be tested; however, they will be impractical for finding good sequences of optimizations from a large set of optimizations. Also, predictive modeling is not used to decide the optimized versions of the code to try.

8 Conclusion

In this paper we address the problem of developing a good modeling technique for predicting compiler optimizations by using performance counters to automatically construct optimization sequences. We do this by using one of machine learning techniques, regression, that predict good code optimization sequences to apply given a program's performance counter features. We evaluated three different predictors, sequences, speedup, and tournament predictors and show that speedup and tournament predictors perform well, especially tournament predictor. For future works, we expect to apply more various machine learning algorithms to build our prediction models. We can also extend our testbed to large applications, different compilers.

References

1. Adhianto, L., Banerjee, S., Fagan, M., Krentel, M., Marin, G., Mellor-Crummey, J., Tallent., N.R.: Hpctoolkit: Tools for performance analysis of optimized parallel programs. Concurrency: Practice and Experience. To appear. (2010)

2. Agakov, F., Bonilla, E., Cavazos, J., Franke, B., Fursin, G., OBoyle, M., Thomson, J., Toussaint, M., Williams, C.: Using machine learning to focus iterative optimization. In: Proceedings of the International Symposium on Code Generation and Optimization. pp. 295–305 (2006)
3. Almagor, L., Cooper, K.D., Grosul, A., Harvey, T.J., Reeves, S.W., Subramanian, D., Torczon, L., Waterman, T.: Finding effective compilation sequences. In: Proceedings of the 2004 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems. pp. 231–239. ACM Press, New York, NY, USA (2004)
4. Cavazos, J., O'Boyle, M.: Method-specific dynamic compilation using logistic regression. In: Proceedings of the ACM SIGPLAN '06 Conference on Object Oriented Programming, Systems, Languages, and Applications. ACM Press, Portland, Or. (October 2006)
5. Cavazos, J., Dubach, C., Agakov, F., Bonilla, E., O'Boyle, M.F., Fursin, G., Temam, O.: Automatic performance model construction for the fast software exploration of new hardware designs. In: Proceedings of the International Conference on Compilers, Architecture, And Synthesis For Embedded Systems (CASES 2006) (October 2006)
6. Cavazos, J., Fursin, G., Agakov, F.V., Bonilla, E.V., O'Boyle, M.F.P., Temam, O.: Rapidly selecting good compiler optimizations using performance counters. In: CGO. pp. 185–197 (2007)
7. Cavazos, J., Moss, J.E.B.: Inducing heuristics to decide whether to schedule. In: Proceedings of the ACM SIGPLAN '04 Conference on Programming Language Design and Implementation. pp. 183–194. ACM Press, Washington, D.C. (June 2004)
8. Cooper, K.D., Schielke, P.J., Subramanian, D.: Optimizing for reduced code space using genetic algorithms. In: Workshop on Languages, Compilers, and Tools for Embedded Systems. pp. 1–9. ACM Press, Atlanta, Georgia (July 1999), `citeseer.nj.nec.com/cooper99optimizing.html`
9. Cooper, K.D., Subramanian, D., Torczon, L.: Adaptive optimizing compilers for the 21st century. Journal of Supercomputing 23(1), 7–22 (August 2002)
10. de Mesmay, F., Voronenko, Y., Püschel, M.: Offline library adaptation using automatically generated heuristics. In: International Parallel and Distributed Processing Symposium (IPDPS) (2010)
11. Dubach, C., Cavazos, J., Franke, B., O'Boyle, M., Fursin, G., Temam, O.: Fast compiler optimisation evaluation using code-feature based performance prediction. In: Proceedings of the ACM International Conference on Computing Frontiers (May 2007)
12. Dubach, C., Jones, T.M., Bonilla, E.V., Fursin, G., O'Boyle, M.F.: Portable compiler optimization across embedded programs and microarchitectures using machine learning. In: Proceedings of the IEEE/ACM International Symposium on Microarchitecture (MICRO) (December 2009)
13. Franke, B., O'Boyle, M., Thomson, J., Fursin, G.: Probabilistic source-level optimisation of embedded programs. In: Proceedings of the 2005 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems. pp. 78–86. ACM Press, New York, NY, USA (2005)
14. Frigo, M., Johnson, S.G.: The design and implementation of FFTW3. Proceedings of the IEEE 93(2), 216–231 (2005), special issue on "Program Generation, Optimization, and Platform Adaptation"

15. Haneda, M., Knijnenburg, P.M.W., Wijshoff, H.A.G.: Automatic selection of compiler options using non-parametric inferential statistics. In: Proceedings of the International Conference on Parallel Architectures and Compilation Techniques. pp. 123–132. IEEE Computer Society, Washington, DC, USA (2005)
16. in Java, W..D.M.S.: http://www.cs.waikato.ac.nz/ml/weka
17. Kisuki, T., Knijnenburg, P.M.W., O'Boyle, M.F.P.: Combined selection of tile sizes and unroll factors using iterative compilation. In: Proceedings of the International Conference on Parallel Architectures and Compilation Techniques. p. 237. IEEE Computer Society, Washington, DC, USA (2000)
18. Kulkarni, P., Hines, S., Hiser, J., Whalley, D., Davidson, J., Jones, D.: Fast searches for effective optimization phase sequences. In: Proceedings of the ACM SIGPLAN '04 Conference on Programming Language Design and Implementation. pp. 171–182. ACM Press, New York, NY, USA (2004)
19. Lau, J., Arnold, M., Hind, M., Calder, B.: Online performance auditing: using hot optimizations without getting burned. SIGPLAN Not. 41(6), 239–251 (2006)
20. Monsifrot, A., Bodin, F., Quiniou, R.: A machine learning approach to automatic production of compiler heuristics. In: AIMSA '02: Proceedings of the 10th International Conference on Artificial Intelligence: Methodology, Systems, and Applications. pp. 41–50. Springer-Verlag (2002)
21. Mucci, P.: Papi – the performance application programming interface. http://icl.cs.utk.edu/papi/index.html (2000)
22. Open64, I.: http://www.open64.net (2006)
23. Parello, D., Temam, O., Cohen, A., Verdun, J.M.: Towards a systematic, pragmatic and architecture-aware program optimization process for complex processors. In: SC '04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing. p. 15. IEEE Computer Society, Washington, DC, USA (2004)
24. Polybench: http://www-roc.inria.fr/ pouchet/software/polybench
25. Puschel, M., Moura, J., Johnson, J., Padua, D., Veloso, M., Singer, B., Xiong, J., Franchetti, F., Gacic, A., Voronenko, Y., Chen, K., Johnson, R.W., Rizzolo, N.: Spiral: Code generation for dsp transforms. Proceedings of the IEEE 93(2), 232–275 (2005), special issue on "Program Generation, Optimization, and Platform Adaptation"
26. Stephenson, M., Amarasinghe, S., Martin, M., O'Reilly, U.M.: Meta optimization: Improving compiler heuristics with machine learning. In: Proceedings of the ACM SIGPLAN '03 Conference on Programming Language Design and Implementation. pp. 77–90. ACM Press, San Diego, Ca (June 2003)
27. Stephenson, M., Amarasinghe, S.P.: Predicting unroll factors using supervised classification. In: Proceedings of the International Symposium on Code Generation and Optimization. pp. 123–134 (2005)
28. Vuduc, R., Demmel, J.W., Bilmes, J.A.: Statistical models for empirical search-based performance tuning. Int. J. High Perform. Comput. Appl. 18(1), 65–94 (2004)
29. Whaley, R.C., Dongarra, J.J.: Automatically tuned linear algebra software. In: SC '98: Proceedings of the 1998 ACM/IEEE conference on Supercomputing. pp. 1–27. IEEE Computer Society, Washington, DC, USA (1998)