An Evaluation of Different Modeling Techniques for Iterative Compilation

Eunjung Park
University of Delaware
Newark, USA
epark@cis.udel.edu

Sameer Kulkarni
University of Delaware
Newark, USA
skulkarn@cis.udel.edu

John Cavazos
University of Delaware
Newark, USA
cavazos@cis.udel.edu

ABSTRACT

Iterative compilation techniques, which involve iterating over different sets of optimizations, have proven useful in helping compilers choose the right set of optimizations for a given 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 to "intelligently" iterate over the optimization search space using predictive methods. In particular, state-the-art methods in iterative compilation use characteristics of the code being optimized to predict good optimization sequences to evaluate. Thus, an important step in developing predictive methods for compilation is deciding how to model the problem of choosing the right optimizations.

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 can outperform current state-of-the-art predictors and the most aggressive setting of the Open64 compiler (`-Ofast`) on an average by 75% in just 10 iterations over a set of embedded and scientific kernels. Moreover, using our tournament predictor, we achieved on average 10% improvement for a set of MiBench applications.

Keywords

compiler optimization, iterative compilation, machine learning, regression

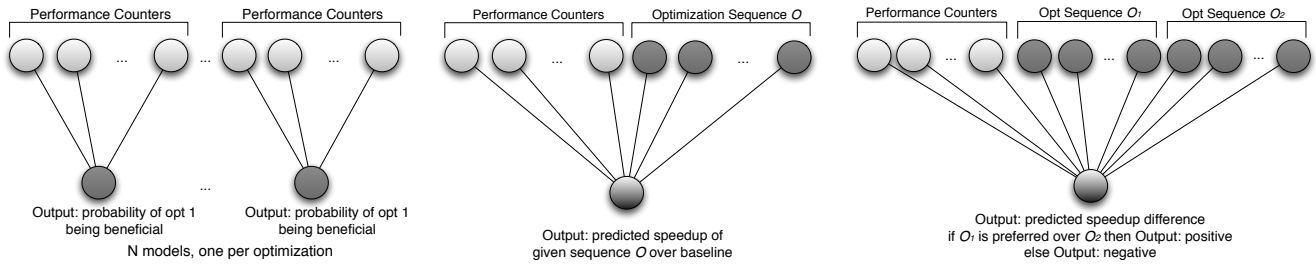
1. INTRODUCTION

Most applications can greatly benefit from a fine-tuned set of compiler optimization sequences. However, finding the right set of optimizations for an application is a non-trivial task since the search space is extremely large. In addition, it is difficult to build analytical models that can predict how combinations of optimizations will interact with each other. Recent research has focused on intelligent search space exploration, in order to search for the right optimization sequence efficiently [5, 7, 9, 10, 14, 18, 20, 26]. In this approach, models are developed automatically using machine learning or statistical techniques to predict good optimizations to apply based on characteristics of the code being optimized. A model that uses characteristics (or features) of the code has the potential to predict optimizations that are specifically tailored to the code.

To use this approach, a key step is "how to model" the problem of choosing the right set of optimizations to apply to a particular code being optimized. One common method of modeling this problem is to develop a decision function that takes as input a given set of features of the code to be optimized and produces as output a prediction of whether a specific 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. A key property is framing the optimization decision problem so that the function to be learned is as simple as possible.

This paper evaluates three different techniques for modeling the problem of predicting a good set of optimizations for a given program. Note, we do not solve the problem of predicting the ordering of optimizations, but instead predict the set of optimizations that should be turned on. We use a large set of kernels which provides good coverage of a wide range of embedded and 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.

In addition, we propose a new modeling technique to predict good optimization sequences to apply for a given program. We refer to this new modeling technique as the *tournament predictor*, and we compare this technique to two state-of-the-art prediction models, which we refer to as the *sequence predictor* and the *speedup predictor*. Each of these prediction models are constructed using performance counters to characterize the dynamic behavior of a program. In recent work, performance counters were shown to be better at characterizing applications than static code features [7]. We collected performance counters for kernels and optimized those kernels with a large set of randomly selected optimization sequences. We evaluated our three modeling techniques



(a) Sequence Predictor (b) Speedup Predictor (c) Tournament Predictor

Figure 1: Three prediction models that are evaluated in this paper.

using two different machine learning algorithms, namely linear regression and support vector machines (SVMs). Also, our models were compared in two different scenarios: a non-iterative scenario where only the best predicted sequence is used and in an iterative scenario where the top-N predicted sequences are used.

Using just the top predicted sequence (i.e., the non-iterative scenario), the tournament predictor trained using SVMs outperforms all models achieving on our kernel suite an average of 37% improvement over our baseline, the most aggressive optimization setting in our compiler, Open64 using `-Ofast`. When we use the top 10 predicted sequences (i.e., the iterative fashion), the sequence and speedup models achieve close to 70% improvement and the tournament predictor achieves 75% improvement over our baseline. Moreover, we tested our models with several benchmarks from the MiBench [17] benchmark suites to show how prediction models trained with kernels performed with embedded applications.

The rest of the paper is organized as follows. We first describe the problem of developing predictors for intelligent modeling, and we briefly describe the different modeling techniques in Section 2. We then give an overview of our solution in Section 3 giving more details of our models, including how we construct them. Section 3.2 describes the machine learning techniques used to build the prediction models. Section 4 explains the characteristics of the programs and the optimization space we used in our evaluation framework. Section 5 describes our evaluation setup, and Section 6 presents our results and analysis. Section 7 describes related work, and Section 8 presents our conclusions.

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 [5], random search [21], statistical techniques [18], or exhaustive search [20]) 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 [4, 7, 11, 13, 22]. 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 present preliminary experiment results showing the potential of our new predictor, a tournament predictor.

2.1 Modeling the Optimization Problem

A general formulation of the problem is to construct a prediction model 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. We briefly describe each of these predictors here and in more detail in Section 3.1.

Previous work [7] 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. Another recent model that has been proposed [6, 12] 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*. Finally, we propose a new method of choosing optimization sequences, the *tournament predictor*, which takes as input a triple 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 the 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

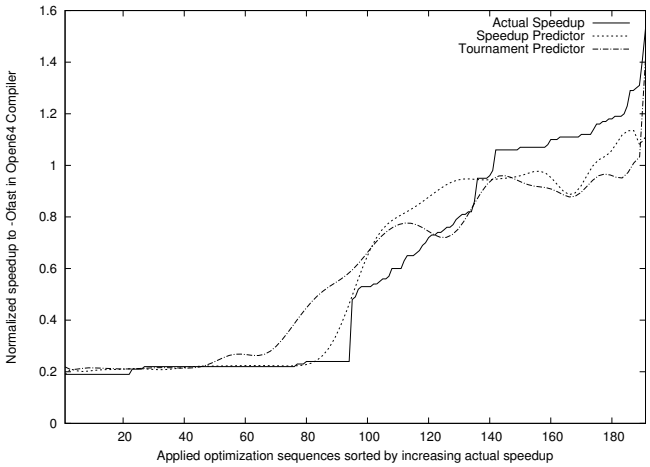


Figure 2: Speedup Predictor and Tournament Predictor for ATAX

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 speedup and tournament predictors, leaving out the benchmark `ATAX` (one of the PolyBench programs [24]) for testing, and training our models with the remaining kernels. A linear regression technique and SVMs were used to build our models. Our training data consists of performance counter characterization of our kernels and the optimization sequences from all but one (left out) kernel. We discuss more about these two algorithms and the training data in Section 3.

Figure 2 shows how predicted speedups with the proposed models compare to actual speedup for the one unseen `ATAX` kernel we used for our test. The y-axis shows speedup obtained after applying a sequence, and the x-axis shows 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. It is worth noting that the tournament predictor's best predicted sequence was the optimal point of our search space, whereas the speedup predictor's best predicted sequence was a good, but not the optimal point in the search space. Using these models, especially the tournament predictor, to choose 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 our training data, how we trained our different models, and details of the specific models themselves. Figure 3 depicts the four steps in constructing our speedup prediction model, (a) collecting performance counter data from programs, (b) creating training data by applying random optimization sequences from our search space and obtaining speedups from the kernels, (c) bringing the performance counter and optimization sequence data together to construct a model, and (d) using the model on an "unseen" program. The sequence and tournament predictor are built in a similar way, but with different inputs and outputs.

We used the HPCToolkit [3] 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 Open64 optimizations. The complete list of optimizations that were used for our experiments are shown in Section 4.2.

We represent each optimization sequence as a bit vector `T`. Optimizations that are turned off or on are represented by a 0 or 1, respectively. Optimizations that require a numeric value are represented by $N-1$ bits where N is the number of possible optimization values we control. For example, for the 'prefetch' optimization in Open64, we use 4 different possible values (0-3), i.e., '0 0 0' for 0, '1 0 0' for 1, '0 1 0' for 2, and '0 0 1' for 3.

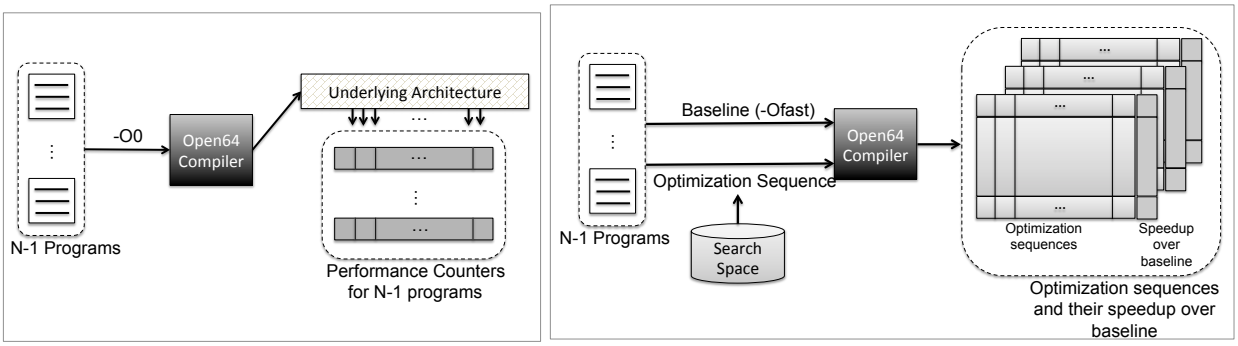
We used each of our 500 randomly generated optimization sequences to compile and execute the programs. We then ran each compiled version five times and used the average running time to calculate speedup over `-Ofast`. Calculated speedup is used to train our models either directly, e.g., with our speedup 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 prediction model being developed. In this section, we describe the three different modeling techniques that we evaluated.

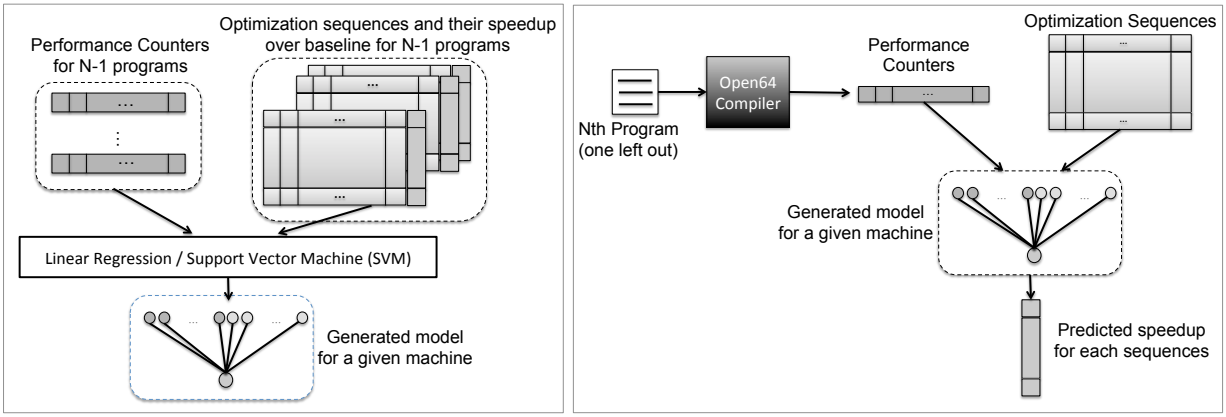
3.1.1 Sequence Predictor

With this predictor (see Figure 1a), we obtain a bit vector corresponding to the number of optimizations we are trying to control. We construct a separate predictor for each optimization. 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) would 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 optimization sequence per benchmark corresponding the optimization that achieves the best speedup for that benchmark. 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 that "unseen" 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 optimizations with more than two possibilities, we use the bit position with the highest probability among all associated bits for a specific optimization. Thus, all the outputs form a probability distribution which we can then



(a) Collecting Perf. Cntrs.

(b) Obtaining Speedup over the Baseline with Opt. Seqs.



(c) Building a Prediction Model

(d) Using a Prediction Model

Figure 3: This figure shows how we build and use a speedup prediction model in a leave-one-out cross-validation scenario. Figure 3(a) depicts the collecting of performance counters for $N-1$ programs. Figure 3(b) shows how we collect optimization training data from $N-1$ programs. Figure 3(c) depicts use of performance counter and optimization sequence data to construct a prediction model. Figure 3(d) shows the use of the trained prediction model.

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.

3.1.2 *Speedup Predictor*

For this predictor (see Figure 1b), the model is trained to accurately predict the speedup of an optimization sequence applied to a program relative to the program optimized using `-Ofast`. The input to the model is the performance counter characterization of a program and a bit vector describing the optimization sequence. The output from the model is the predicted speedup of that optimization 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 optimization 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 previously "unseen". In other words, we can use this predictor to predict the speedup of sequences that were not previously used to train our model. Our evaluation demonstrates that this predictor is effective at finding good sequences for both "unseen" programs and "unseen" sequences.

3.1.3 *Tournament Predictor*

This model (see Figure 1c) is trained to predict, which of two different optimization sequences presented to the model is better. The inputs supplied to this model is the performance counter characterization of a program and two optimization sequences that could be applied to the program. The output of the model is either true or false depending on whether the model predicts the first optimization sequence is better or worse than the second sequence. Since it predicts speedup difference of two optimization sequences, this model not only predicts which optimization sequence is better, but it also predicts by how much better the optimization sequence is. 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.

3.2 Learning Algorithms

A variety of learning algorithms can be brought to bear on the task of predicting the right set of optimizations to apply to a program. We consider two methods here. The first method is support vector machines (SVMs) which is a class of supervised learning algorithms that can be used for both classification and regression. SVMs use kernel functions to transform the training data into a different, linearly separable feature space, and then a linear classifier is constructed that separates the points into multiple classes. The second method we used was linear regression. This is the standard linear regression technique found in many statistical and machine learning text books.

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	L1-DCA, L1-DCM, L1-ICA, L1-LDM, L1-STM, L1-TCM
Level 2 Cache	L2-DCR, L2-DCW, L2-ICA, L2-STM, L2-TCA, L2-TCM, L2-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 available from PAPI to characterize a program.

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 [7]. We used 29 different performance counters shown in Table 1.

4.2 Optimization Space

We selected 7 optimization phases from the Open64 compiler and 45 individual optimizations from these selected phases. These 45 optimizations make up the optimization space we explored 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. Most selected optimizations are either turned on or off, so they could easily be represented as binary values. However, certain optimizations require values in a given range, e.g., loop unrolling factor or loop tile size. All optimizations used in our paper are shown in Table 2.

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, value-numbering, combine, dce-aggressive, iv-elimination, spre, canon-expr
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, space, plimit,pu-reorder, small-pu, min-hotness

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

From the listed optimizations in Table 2, we generated a set of 500 random optimization sequences. We then evaluated each sequence on the programs in our training set to achieve actual speedup of that sequence over `-Ofast`. The random optimization sequence and its corresponding speedup are used as training data.

5. EXPERIMENTAL SETUP

This section describes the experimental setup including the machine configuration and software platform used. We also describe the benchmarks used for our experimental study. We performed 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 collected performance counters using HPCToolkit [3] and the PAPI 3.6 hardware counter library [23]. Table 1 gives a brief description of the performance counters we used for this study. We used the open-source Open64 compiler version 4.2.1 [2], and all speedups reported are relative to `-Ofast`, which is the most aggressive optimization level available in this compiler. For our machine learning algorithms, we used linear regression and support vector machines (SVMs) contained in Weka [1] 3.6.2. For the SVMs, we used a normalized polynomial kernel, and for linear regression we used the default Weka configuration.

5.1 Benchmarks

We decided to first experiment with small pieces of code that would allow us to quickly evaluate different learning algorithms and 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. We then removed all kernels that had a variability of more than a few percent after 50 runs of the code. For the

Regression

Evaluations	1	5	10	20	30	40	50
Sequence	1.15×(59%)	1.53×(78%)	1.61×(82%)	1.70×(87%)	1.72×(88%)	1.74×(89%)	1.76×(90%)
Speedup	1.27×(65%)	1.60×(82%)	1.69×(86%)	1.80×(92%)	1.84×(94%)	1.88×(96%)	1.88×(96%)
Tournament	1.22×(62%)	1.55×(79%)	1.75×(89%)	1.80×(92%)	1.86×(95%)	1.88×(96%)	1.89×(96%)

SVM

Sequence	1.18×(60%)	1.53×(78%)	1.63×(82%)	1.81×(92%)	1.84×(94%)	1.85×(94%)	1.86×(95%)
Speedup	1.23×(63%)	1.52×(78%)	1.68×(86%)	1.87×(95%)	1.88×(96%)	1.89×(96%)	1.89×(96%)
Tournament	1.37×(70%)	1.67×(85%)	1.74×(89%)	1.86×(95%)	1.89×(96%)	1.89×(96%)	1.90×(97%)

Table 3: We show the speedup over baseline for our benchmark suite using the top 1, 5, 10, 20, 30, 40, and 50 predicted optimization sequences from our different models. Each speedup is followed by a percentage value (in parentheses). This value shows the percentage of search space optimal speedup we achieved for a given number of predicted sequences. We show results for our models trained using linear regression and using support vector machines (SVMs).

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.

For another set of experiments, we used a subset of the MiBench benchmarks that we could both compile successfully with Open64 and which did not have high variability in the runs. We evaluated our tournament predictor by training our model on kernel programs, and testing the model on the "unseen" subset of MiBench benchmarks. We also evaluated our model on both the "unseen" MiBench benchmarks and "unseen" optimization sequences. This experiment allowed us to test whether our tournament predictor could be trained using small kernels, and later used for optimizing larger applications. In addition, this experiment allowed us to test whether the model could generalize to optimization sequences that it had not been trained on.

6. EXPERIMENTAL RESULTS

This section describes our experiment results that we conducted to evaluate our different modeling techniques.

6.1 Leave-One-Out Cross Validation on Kernels

To evaluate our modeling techniques, we used leave-one-out cross-validation over a set of kernel benchmarks. This allowed us to evaluate the speedups obtained between our different modeling techniques when used to optimize "unseen" kernels.

Table 3 shows speedups obtained relative to `-Ofast` for each of our models for 1 to 50 "evaluations." In parenthesis, we show the speedup achieved relative to the maximum speedup found for the 500 random optimization sequences evaluated. We show results for training our models using linear regression and SVMs. The row labeled "Evaluations" refers to the number of predicted sequences we evaluated from our model. For example, 1 evaluation (first column) corresponds to taking only the top predicted optimization sequence from our models, i.e., using the models in a non-iterative fashion. The rest of the columns corresponds to using the models in an iterative fashion, where N evaluations refers to keeping the best actual speedup obtained from a model's top N predicted optimization sequences. As the table shows, all our models achieve significant speedups over `-Ofast`.

The speedup predictor trained using regression performed fairly well, out-performing the tournament predictor and sequence predictor for 1 evaluation. Interestingly, the tournament predictor starts to catch up after 5 evaluations, and outperforms all predictors at 10 evaluations. After 10 evaluations, the speedup and tournament predictors perform about even, with the sequence predictor lagging behind.

On the other hand, we achieve better performance from the tournament predictor when trained using SVMs versus regression. [1] SVMs can construct non-linear models, which obviously benefits the tournament predictors. Constructing a tournament predictor using SVMs allows it to perform better than any other models at 1 evaluation. As we increase the number of evaluations to 10, all three modeling techniques show good speedup, with the tournament predictor still outperforming the other predictors. At 20 evaluations and greater, both the speedup and tournament predictors perform equally well. Thus, regardless of learning algorithm, the tournament predictor performed best at 10 evaluations by achieving 75% over `-Ofast`, achieving almost 90% of the speedup available from the search space. Also, for 1 evaluation, the sequence predictor performs significantly worse than the other models.

Table 4 shows detailed results for each of our models trained using regression and SVMs for 10 evaluations. We notice that most kernels benefit from using predicted optimization sequences for all predictors. Some benchmarks, such as `Livermore Loop11 (Livermr-Lp11)` or `bicg`, obtain large benefits from using the sequence predictor over the other two predictors. However, the tournament and speedup predictor perform better than sequence predictor on most kernels. On average, the tournament predictor outperforms the other two predictors achieving 74% of performance improvement over `-Ofast`. At 10 evaluations, both machine learning techniques performed similarly on all predictors, and all predictors achieved more than 80% of the maximum available for the search space explored.

In summary, the tournament predictor trained using SVMs works consistently well in an iterative (> 1 evaluation) and non-iterative fashion (1 evaluation). Thus, for the remainder of the paper we further explore the tournament predictor trained using SVMs.

6.2 Evaluations on MiBench

This section describes results using our tournament predictor trained on kernels and tested on MiBench benchmarks. We performed two different kinds of experiments. First, we tested how well our predictor found good optimization sequences to apply to MiBench benchmarks from

[1]Note that the same training data was used with regression and SVMs.

	Regression			SVM			
Benchmark	Sequence	Speedup	Tournament	Sequence	Speedup	Tournament	Opt
fir.arrays	1.27×(60%)	2.11×(100%)	2.11×(100%)	1.58×(74%)	1.72×(81%)	2.11×(100%)	2.11×
fir.arrays-SWP	1.83×(59%)	2.76×(90%)	2.76×(90%)	2.01×(65%)	2.49×(81%)	2.76×(90%)	3.07×
fir.ptrs	1.84×(64%)	1.72×(60%)	2.83×(99%)	1.45×(51%)	2.83×(100%)	2.83×(99%)	2.83×
fir.ptrs-SWP	1.34×(57%)	1.95×(83%)	1.91×(82%)	2.49×(107%)	2.32×(100%)	1.45×(62%)	2.32×
latnrm.arrays	1.64×(47%)	3.47×(100%)	3.47×(99%)	1.91×(55%)	3.47×(100%)	2.75×(79%)	3.47×
latnrm.arrays-SWP	1.98×(70%)	2.80×(100%)	2.76×(98%)	2.24×(80%)	2.76×(98%)	2.76×(98%)	2.80×
latnrm.ptrs	2.59×(87%)	1.49×(49%)	2.98×(100%)	2.47×(82%)	2.98×(100%)	2.98×(100%)	2.98×
latnrm.ptrs-SWP	2.35×(111%)	2.11×(100%)	1.84×(87%)	2.41×(114%)	2.11×(100%)	1.84×(87%)	2.11×
lmsfir.arrays	2.80×(87%)	1.57×(49%)	1.37×(42%)	1.45×(45%)	1.76×(54%)	3.21×(100%)	3.21×
lmsfir.arrays-SWP	1.16×(50%)	1.45×(62%)	2.12×(91%)	2.23×(96%)	1.68×(72%)	2.12×(91%)	2.31×
lmsfir.ptrs	1.87×(91%)	1.61×(78%)	2.04×(100%)	1.91×(93%)	1.70×(83%)	1.61×(78%)	2.04×
lmsfir.ptrs-SWP	1.70×(55%)	3.05×(100%)	1.66×(54%)	1.00×(32%)	3.05×(100%)	1.76×(57%)	3.05×
mult.arrays	0.99×(53%)	1.53×(82%)	1.71×(92%)	1.76×(94%)	1.71×(92%)	1.71×(92%)	1.86×
mult.arrays-SWP	1.07×(41%)	2.54×(100%)	1.98×(78%)	0.91×(36%)	1.98×(78%)	1.98×(78%)	2.54×
mult.ptrs	1.04×(35%)	2.96×(100%)	2.63×(88%)	2.81×(95%)	1.87×(63%)	2.63×(88%)	2.96×
mult.ptrs-SWP	1.27×(37%)	3.35×(100%)	3.35×(100%)	1.18×(35%)	1.88×(56%)	3.01×(89%)	3.35×
appsp	1.11×(90%)	1.18×(95%)	1.18×(95%)	1.06×(85%)	1.04×(83%)	1.18×(95%)	1.24×
clinpack	1.00×(73%)	1.31×(96%)	1.37×(100%)	0.97×(70%)	1.22×(89%)	1.33×(97%)	1.37×
Livermr-Lp2	1.01×(83%)	1.08×(89%)	1.12×(92%)	1.06×(87%)	1.08×(89%)	1.12×(92%)	1.21×
Livermr-Lp3	2.36×(76%)	2.17×(69%)	2.17×(69%)	2.71×(87%)	2.17×(69%)	3.11×(100%)	3.11×
Livermr-Lp4	1.69×(153%)	0.81×(73%)	0.92×(83%)	0.93×(84%)	1.04×(94%)	1.04×(94%)	1.10×
Livermr-Lp5	1.24×(83%)	1.33×(89%)	1.42×(96%)	1.39×(94%)	1.33×(89%)	1.36×(92%)	1.48×
Livermr-Lp10	2.71×(144%)	1.48×(78%)	1.48×(78%)	3.61×(192%)	1.76×(93%)	1.48×(78%)	1.88×
Livermr-Lp11	2.53×(118%)	1.63×(76%)	2.14×(100%)	2.41×(112%)	1.60×(74%)	2.14×(100%)	2.14×
Livermr-Lp12	2.11×(89%)	1.40×(59%)	2.37×(99%)	2.49×(105%)	2.37×(100%)	1.40×(59%)	2.37×
Livermr-Lp13	1.86×(72%)	2.56×(100%)	2.42×(94%)	1.85×(72%)	2.54×(98%)	2.54×(99%)	2.56×
Livermr-Lp14	2.71×(123%)	2.09×(94%)	2.09×(94%)	1.43×(65%)	1.55×(70%)	2.09×(94%)	2.20×
Livermr-Lp15	2.37×(112%)	2.10×(99%)	2.10×(99%)	2.27×(107%)	2.10×(99%)	2.10×(100%)	2.10×
Livermr-Lp16	2.15×(120%)	1.79×(100%)	1.73×(96%)	3.62×(202%)	1.79×(100%)	1.73×(96%)	1.79×
Livermr-Lp17	1.45×(52%)	2.63×(94%)	2.63×(94%)	1.09×(39%)	2.65×(95%)	2.77×(99%)	2.77×
Livermr-Lp18	1.98×(74%)	1.46×(55%)	2.64×(100%)	2.13×(80%)	2.64×(100%)	2.64×(100%)	2.64×
Livermr-Lp19	2.33×(96%)	1.61×(66%)	1.61×(66%)	2.50×(103%)	2.36×(97%)	2.06×(84%)	2.42×
Livermr-Lp20	2.11×(89%)	2.27×(96%)	2.27×(96%)	1.80×(76%)	1.48×(62%)	1.93×(81%)	2.37×
Livermr-Lp21	1.81×(72%)	2.49×(100%)	2.49×(100%)	1.67×(66%)	1.76×(70%)	2.49×(100%)	2.49×
Livermr-Lp22	3.05×(106%)	2.28×(79%)	2.28×(79%)	2.87×(100%)	1.96×(68%)	2.53×(88%)	2.85×
Livermr-Lp23	2.54×(99%)	2.54×(100%)	2.54×(100%)	2.00×(78%)	1.50×(58%)	2.16×(84%)	2.54×
Livermr-Lp24	1.21×(47%)	1.60×(62%)	1.60×(62%)	2.14×(83%)	1.58×(61%)	1.60×(62%)	2.56×
adi	1.14×(102%)	1.04×(93%)	1.11×(99%)	0.60×(54%)	1.03×(92%)	1.11×(99%)	1.11×
atax	1.33×(73%)	1.65×(91%)	1.60×(87%)	1.36×(74%)	1.82×(100%)	1.58×(87%)	1.82×
bicg	2.55×(129%)	1.94×(98%)	1.85×(94%)	2.27×(115%)	1.94×(98%)	1.85×(94%)	1.97×
cholesky	1.10×(103%)	1.06×(99%)	1.03×(97%)	1.02×(95%)	1.05×(98%)	1.06×(99%)	1.06×
correlation	0.58×(88%)	0.62×(94%)	0.65×(98%)	0.59×(90%)	0.59×(89%)	0.63×(94%)	0.66×
covariance	0.62×(102%)	0.57×(95%)	0.60×(98%)	0.61×(100%)	0.60×(98%)	0.59×(97%)	0.60×
doitgen	0.47×(88%)	0.52×(98%)	0.50×(94%)	0.42×(80%)	0.51×(96%)	0.50×(94%)	0.53×
durbin	1.06×(89%)	1.14×(96%)	1.14×(96%)	1.07×(90%)	1.16×(98%)	1.14×(96%)	1.18×
dynprog	1.19×(102%)	1.00×(85%)	0.98×(83%)	1.06×(90%)	1.11×(95%)	1.08×(93%)	1.17×
fdtd-2d	0.93×(77%)	1.18×(98%)	1.14×(94%)	0.96×(80%)	1.13×(94%)	1.14×(94%)	1.20×
fdtd-apml	1.19×(92%)	1.17×(90%)	1.18×(91%)	1.17×(90%)	1.18×(91%)	1.16×(90%)	1.28×
gauss-filter	0.86×(78%)	0.94×(84%)	0.97×(88%)	0.95×(86%)	0.82×(74%)	0.94×(84%)	1.10×
gemm	0.97×(85%)	1.06×(92%)	1.06×(92%)	0.93×(81%)	1.00×(87%)	1.06×(92%)	1.15×
gemver	0.94×(96%)	0.93×(95%)	0.94×(96%)	0.88×(89%)	0.91×(93%)	0.93×(95%)	0.97×
gesummv	1.84×(93%)	1.62×(82%)	1.64×(83%)	1.95×(98%)	1.76×(89%)	1.66×(84%)	1.97×
gramschmidt	3.91×(93%)	4.07×(97%)	4.06×(97%)	3.95×(94%)	4.04×(97%)	4.15×(99%)	4.16×
jacobi-1d-imper	1.77×(90%)	1.82×(93%)	1.82×(93%)	1.61×(82%)	1.87×(95%)	1.80×(91%)	1.96×
jacobi-2d-imper	1.02×(93%)	1.02×(93%)	1.05×(96%)	1.01×(93%)	1.08×(99%)	1.05×(96%)	1.09×
lu	2.16×(92%)	1.06×(45%)	1.06×(45%)	1.04×(44%)	1.05×(44%)	1.08×(46%)	2.34×
ludcmp	1.02×(86%)	1.18×(100%)	1.18×(100%)	1.03×(87%)	1.14×(97%)	1.16×(98%)	1.18×
mvt	1.13×(72%)	1.41×(90%)	1.47×(94%)	1.42×(91%)	1.55×(99%)	1.39×(89%)	1.56×
reg-detect	1.17×(89%)	1.19×(91%)	1.24×(94%)	1.15×(87%)	1.27×(97%)	1.19×(91%)	1.31×
seidel	0.87×(79%)	1.08×(99%)	1.08×(98%)	0.99×(90%)	1.06×(97%)	1.05×(95%)	1.09×
symm	1.01×(96%)	1.00×(95%)	1.01×(96%)	0.98×(93%)	1.00×(95%)	0.98×(93%)	1.05×
syr2k	0.99×(78%)	1.11×(88%)	1.12×(88%)	0.91×(72%)	1.14×(90%)	1.09×(86%)	1.26×
syrk	1.13×(99%)	1.11×(97%)	1.04×(91%)	1.11×(97%)	1.07×(94%)	1.04×(91%)	1.14×
trisolv	2.40×(104%)	1.91×(83%)	1.96×(85%)	2.26×(98%)	1.80×(78%)	1.91×(83%)	2.30×
trmm	1.09×(103%)	1.05×(100%)	1.05×(100%)	0.68×(64%)	1.01×(95%)	1.05×(100%)	1.05×
AVG	1.61×(82%)	1.69×(86%)	1.75×(89%)	1.63×(83%)	1.68×(85%)	1.75×(89%)	1.96×

Table 4: We show speedups over baseline (ICC -fast) using the top 10 predictions for the sequence, speedup, and tournament predictor. We include (in parenthesis) the percent of search space optimal speedup that was obtained. The last column 'Opt' represents search space optimal speedup.

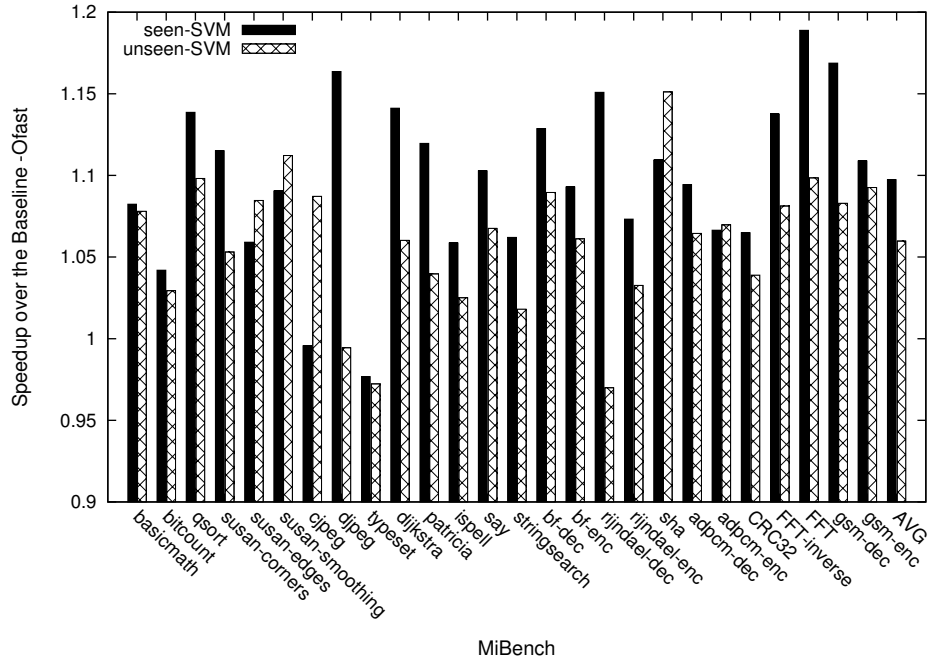


Figure 4: This figure shows the maximum speedup obtained with 10 predictions for our tournament predictor trained using SVM for a subset of the MiBench benchmarks, which we could compile. Two different predictors were evaluated, one using a set of "seen" optimization sequences were the benchmarks were not seen and another using on a set of "unseen" sequences and "unseen" benchmarks. The SVM predictors achieved an average speedup for "seen" and "unseen" sequences of 10% and 6%, respectively.

the same set of optimization sequences we had used in the kernel training data. That is, a tournament predictor was trained on kernels and tested on MiBench ("unseen") benchmarks, but the optimizations sequences used for training and testing were the same. Second, we again constructed a tournament predictor on kernels, but this time we tested the predictor on both "unseen" benchmarks (MiBench) and "unseen" sequences.

As is shown in Figure 4, the tournament predictor significantly out-performed `-Ofast` on most benchmarks (except for `cjpeg` and `typeset`) for "seen" sequences. For "seen" sequences, we achieve on average 10% performance improvement over `-Ofast`, where the maximum speedup available found was 15.4%. We also achieved good performance improvements with "unseen" sequences by obtaining 6% on average, where the maximum speedup found was 12%. We note that performance improvement is better for "seen" sequences, however, we can still achieve good results using our models to predict the performance of "unseen" sequences.

6.3 Optimization Sequences Across Different Machines

This section looks at the portability of our models across different machines. We looked at three different Intel machines that varied in processor speed, cache, and memory size. Machine 1 is an Intel Quad Core2 Q9300 2.5GHz with 4GB RAM and 6MB L2 cache, Machine 2 is an Intel Quad Core2 Q9650 3.0GHz with 8GB RAM and 12MB L2 cache, and Machine 3 is an Intel Xeon E5335 2.0GHz with 2GB RAM and 8MB L2 cache. We evaluated 500 different optimization sequences on a representative benchmark `FFT-inverse` from the MiBench suite. Figure 5 shows the optimization sequences sorted by their performance on Machine 1. Note, the optimization sequences for this benchmark have the same speedup trends on the different machines. We conclude that sequences performing well on our models trained on one machine would perform well on other similar architectures. We saw similar trends for our other programs.

7. RELATED WORK

In recent years, several previous works have shown the benefits of iterative compilation [5,14,18,19]. 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 [15,25,27].

However, iterative compilation techniques can be expensive, and current research has looked into improving the speed of these techniques. Almagor *et al.* [5] take a 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.* [20] 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 may miss opportunities because the constrain the search space a-priori before optimizing a program. Cooper *et al.* [10] 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, all these techniques are application-specific. That is, these search algorithms (e.g., genetic algorithms) have to be "retrained" for each program to decide the best optimization sequence for that program.

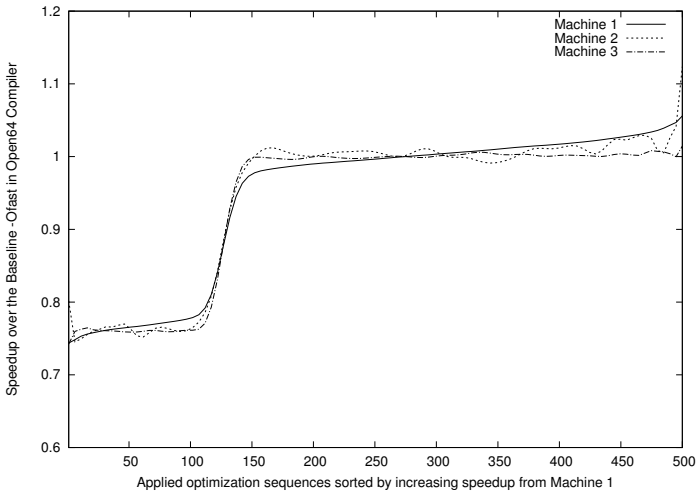


Figure 5: This graph represents how each optimization sequence performs over different architectures for the benchmark `FFT-inverse`. Other benchmarks produce similar graphs. The x-axis shows the sequences sorted by speedup on Machine 1. The y-axis shows the speedup over baseline, which is -Ofast. Machine 1 is Intel Quad Core2 Q9300 2.5GHz 4GB RAM and 6MB L2 cache, Machine 2 is Intel Quad Core2 Q9650 3.0GHz 8GB RAM and 12MB L2 cache, and Machine 3 is Intel Xeon E5335 2.0GHz 2GB RAM and 8MB L2 cache. We observe the sequence that gives us good performance on one machine usually works well on other machines.

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. However, a closer look at the results for two of their optimizations indicate that most of 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.

Fursin *et al.* [16] discuss an extension to GCC called Milepost GCC, which can provide optimization strategies according to an objective function that user seeks to improve. Milepost GCC uses static features of a program to make predictions of good optimizations to apply. In this work, we used dynamic features of a programs, which have been shown to be better predictors [7] of of good optimization sequences. In contrast to this work, Fursin *et al.* [16] do not evaluate different methods of modeling the problem of how to predict good optimization sequences.

Cavazos *et al.* [8] 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 losing any performance. Recently, Cavazos *et al.* [9] 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.

8. CONCLUSION

In this paper, we introduce and evaluate a novel modeling technique, the tournament predictor, for predicting good compiler optimizations to apply to an "unseen" program. Moreover, we compare this modeling technique to two state-of-the-art modeling techniques, namely the *sequence* and *speedup* predictor. We build our models by using two machine learning techniques, regression and SVMs. These models predict good code optimization sequences to apply given a program's performance counter characterization. We first evaluated the prediction models on a large set of kernels with leave-one-out cross validation. From our experimental results, speedup and tournament predictors performed better than sequence predictor. The tournament predictor out-performed both the sequence and speedup predictor at 1 evaluation and 10 evaluations, and in 10 evaluations achieving 75% improvement over `-Ofast` in Open64 compiler. We trained the tournament predictor on kernels to evaluate a set of embedded domain applications from MiBench with "seen" and "unseen" sequences. We achieved good performance improvement with 10% and 6% for "seen" and "unseen" sequences respectively, which is 65% and 50% of the maximum speedup found for the 500 random optimization sequences evaluated.

For future work, we expect to apply additional machine learning algorithms to construct our prediction models. We will also extend our testbed to different domains of applications, different compilers, and a larger optimization space.

9. REFERENCES

[1] Weka 3: Data mining software in java. http://www.cs.waikato.ac.nz/ml/weka.

[2] Open64, inc. http://www.open64.net, 2006.

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

[4] F. Agakov, E. Bonilla, J. Cavazos, B. Franke, G. Fursin, M. O'Boyle, J. Thomson, M. Toussaint, and C. Williams. Using machine learning to focus iterative optimization. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 295–305, 2006.

[5] L. Almagor, K. D. Cooper, A. Grosul, T. J. Harvey, S. W. Reeves, D. Subramanian, L. Torczon, and T. Waterman. Finding effective compilation sequences. In *Proceedings of the 2004 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, pages 231–239, New York, NY, USA, 2004. ACM Press.

[6] J. Cavazos, C. Dubach, F. Agakov, E. Bonilla, M. F. O'Boyle, G. Fursin, and O. Temam. 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.

[7] J. Cavazos, G. Fursin, F. V. Agakov, E. V. Bonilla, M. F. P. O'Boyle, and O. Temam. Rapidly selecting good compiler optimizations using performance counters. In *CGO*, pages 185–197, 2007.

[8] J. Cavazos and J. E. B. Moss. Inducing heuristics to decide whether to schedule. In *Proceedings of the ACM SIGPLAN '04 Conference on Programming Language Design and Implementation*, pages 183–194, Washington, D.C., June 2004. ACM Press.

[9] J. Cavazos and M. O'Boyle. Method-specific dynamic compilation using logistic regression. In *Proceedings of the ACM SIGPLAN '06 Conference on Object Oriented Programming, Systems, Languages, and Applications*, Portland, Or., October 2006. ACM Press.

[10] K. D. Cooper, P. J. Schielke, and D. Subramanian. Optimizing for reduced code space using genetic algorithms. In *Workshop on Languages, Compilers, and Tools for Embedded Systems*, pages 1–9, Atlanta, Georgia, July 1999. ACM Press.

[11] F. de Mesmay, Y. Voronenko, and M. Püschel. Offline library adaptation using automatically generated heuristics. In *International Parallel and Distributed Processing Symposium (IPDPS)*, 2010.

[12] C. Dubach, J. Cavazos, B. Franke, M. F. O'Boyle, G. Fursin, and O. Temam. Fast compiler optimisation evaluation using code-feature based performance prediction. In *Proceedings of the ACM International Conference on Computing Frontiers*, May 2007.

[13] C. Dubach, T. M. Jones, E. V. Bonilla, G. Fursin, and M. F. O'Boyle. Portable compiler optimization across embedded programs and microarchitectures using machine learning. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture (MICRO)*, December 2009.

[14] B. Franke, M. O'Boyle, J. Thomson, and G. Fursin. Probabilistic source-level optimisation of embedded programs. In *Proceedings of the 2005 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, pages 78–86, New York, NY, USA, 2005. ACM Press.

[15] M. Frigo and S. G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005. special issue on "Program Generation, Optimization, and Platform Adaptation".

[16] G. Fursin, Y. Kashnikov, A. Memon, Z. Chamski, O. Temam, M. Namolaru, E. Yom-Tov, B. Mendelson, A. Zaks, E. Courtois, F. Bodin, P. Barnard, E. Ashton, E. Bonilla, J. Thomson, C. Williams, and M. O'Boyle. Milepost gcc: Machine learning enabled self-tuning compiler. *International Journal of Parallel Programming*, 39:296–327, 2011. 10.1007/s10766-010-0161-2.

[17] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. Mibench: A free, commercially representative embedded benchmark suite. In *IEEE 4th Annual Workshop on Workload Characterization*, Austin, TX, December 2001.

[18] M. Haneda, P. M. W. Knijnenburg, and H. A. G. Wijshoff. Automatic selection of compiler options using non-parametric inferential statistics. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, pages 123–132, Washington, DC, USA, 2005. IEEE Computer Society.

[19] T. Kisuki, P. M. W. Knijnenburg, and M. F. P. O'Boyle. Combined selection of tile sizes and unroll factors using iterative compilation. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, page 237, Washington, DC, USA, 2000. IEEE Computer Society.

[20] P. Kulkarni, S. Hines, J. Hiser, D. Whalley, J. Davidson, and D. Jones. Fast searches for effective optimization phase sequences. In *Proceedings of the ACM SIGPLAN '04 Conference on Programming Language Design and Implementation*, pages 171–182, New York, NY, USA, 2004. ACM Press.

[21] J. Lau, M. Arnold, M. Hind, and B. Calder. Online performance auditing: using hot optimizations without getting burned. *SIGPLAN Not.*, 41(6):239–251, 2006.

[22] A. Monsifrot, F. Bodin, and R. Quiniou. 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*, pages 41–50. Springer-Verlag, 2002.

[23] P. Mucci. Papi – the performance application programming interface. http://icl.cs.utk.edu/papi/index.html, 2000.

[24] Polybench. http://www-roc.inria.fr/ pouchet/software/polybench.

[25] M. Puschel, J. Moura, J. Johnson, D. Padua, M. Veloso, B. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo. 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] M. Stephenson, S. Amarasinghe, M. Martin, and U.-M. O'Reilly. Meta optimization: Improving compiler heuristics with machine learning. In *Proceedings of the ACM SIGPLAN '03 Conference on Programming Language Design and Implementation*, pages 77–90, San Diego, Ca, June 2003. ACM Press.

[27] R. C. Whaley and J. J. Dongarra. Automatically tuned linear algebra software. In *SC '98: Proceedings of the 1998 ACM/IEEE conference on Supercomputing*, pages 1–27, Washington, DC, USA, 1998. IEEE Computer Society.