Tzu-Han Hung, Jiunn-Yeu Chen, and Wuu Yang, Wei Chung Hsu, Program Type Recognition for Compiler Optimization, 7th Workshop on Optimizations for DSP and Embedded Systems (ODES-7, March 22nd, 2009, Seattle, Washington), 2009.

Program Type Recognition for Compiler Optimization

Abstract

Today's compilers have too many optimization options, and it is difficult to understand all the details and the interactions between them. Therefore, many application developers simply use the well-known compiler flags (such as `-O2`, `-Os`*) to compile all their programs; at the other end of the extreme, some researchers focus on customizing the optimizations for each program. The former method is too naive as it treats all kinds of programs in the same way, and some programs might even have performance loss since the compilation options are in fact tuned for the others. The latter approach takes care of the variations of programs and tries to individually tune them to the best. However, since the optimization space is extremely huge, the scalability of this scheme is a problem and also makes this approach less popular for the emerging dynamic optimizing compilers.*

We attack this problem by exploiting two insights: first, it is feasible to categorize the programs into different classes according to their functionalities and characteristics; second, dynamic profiling information is useful for recognizing the type of a program. With both insights, we present a novel approach for program optimization: first, the programs are categorized into different classes and the class-level optimization decision is made by the domain experts; then, the class of an input program is recognized by the machine learning models and the optimal class-level optimizations are applied to obtain the best performance. We demonstrate that this approach can be applied to a binary translation/optimization system and great performance improvement can be achieved.

1. Introduction

There have been great research efforts and achievements on compiler optimizations, and yet in practice, people still find it hard to optimize a program, even with a powerful compiler at hand. Today's compilers have too many optimization options, and it is difficult to understand all the details and the interactions between them. Very often the application developers simply use some pre-defined well-understood compiler flags (such as `-O2`, `-Os`) to compile all their programs; at the other end of the extreme, some researchers focus on customizing the optimizations for each program. The former method is too naive as it treats all kinds of programs in the same way, and some programs might even have performance loss since the compilation options are in fact tuned for the other programs. The latter approach takes care of the variations of programs and tries to individually tune them to the best. However, since the optimization space is extremely huge, the scalability of this scheme is a problem.

[Copyright notice will appear here once 'preprint' option is removed.]

At the same time, dynamic compilation systems are becoming more prevalent [6, 19, 8]. A dynamic compiler cannot afford the expensive whole-program analysis and optimization space exploration. Even it has the access to runtime profiling data and may use this information to select suitable optimizations, the time for *cold start* (i.e., the time before the running program enters *hot code* and the dynamic compiler recognizes it) and the time for performing profiling sometimes offset the benefits from dynamic optimizations if huge amount of dynamic information is required.

In this paper, we try to solve the program optimization problem by exploiting two insights. First, instead of optimizing all the programs in a uniform way or highly customizing each program with its own set of optimizations, we find it feasible to *categorize the programs into different classes according to their functionalities and characteristics*. This categorization introduces the concept of class-level optimization (i.e., the most beneficial set of optimizations for this class of programs) and makes it possible to bring in the knowledge of domain experts and compiler writers and focus on a particular class of programs at a time. Second, *dynamic profiling information is useful for recognizing the type of a program*. This observation suggests the feasibility of building a program type predictor using indicative dynamic attributes. The prediction model built can then guide the dynamic compiler to perform appropriate optimizations, with two more benefits: since only a small amount of dynamic attributes is necessary, the cost for profiling is relatively low; since the model can recognize the program type at an early stage, the optimizations can be applied earlier to improve the overall performance.

In summary, this paper makes the following contributions:

1. With the notion of program type categorization and dynamic program characteristics, we propose a novel approach for optmization space exploration: an application to be compiled is run once, the profiling attributes are collected and used for predicting the class of this application, and then the best class-level optimizations are applied to optimize this application.
2. We experiment with several state-of-the-art machine learning algorithms and find some promising models for recognizing the program type. We also present insights into practical selection and collection of dynamic program characteristics.
3. We consider a comprehensive combination of system-wide variables, including the profiling window sizes, the architecture parameters, and the compiler options. The experimental results suggest that it is possible to build a universal predictor, having little accuracy loss across different platform settings, if a good prediction algorithm is used and appropriate profiling attributes are selected.
4. We demonstrate how the program type predictor can help a binary translator choose the most profitable optimizations to apply to each individual program, without the need to explore all possible candidates in a brute-force manner. Results have shown that most algorithms studied in this work can achieve greater than 4% performance improvement, compared to the best suite-level optmizations.

The rest of the paper is organized as follows. Section 2 summarizes the related work and compares the differences of this paper and previous research. Section 3 introduces the platforms and details for the experiments and section 4 presents and analyzes the experimental results. Section 5 demonstrates the application of program type prediction model to a binary translation system and shows the performance improvement achieved. Finally, section 6 discusses several relevant issues and section 7 concludes this paper.

2. Related Work

It is becoming more popular to use machine learning techniques to direct the research in the system optimization community, mostly because of the complexity of today's computer systems. With a huge set of compiler optimizations and architectural configurations, it is a challenging problem to understand all the interactions of the components in a computer system and to retrieve the best performance out of it. Therefore, some researchers find it useful to take advantage of modern statistics methods to help the study of computer systems.

A typical use of machine learning techniques in the compiler field is to predict the performance of a program and use this information to determine the best combination of optimizations for the program. For example, Vaswani et al. [25] use compiler flags and micro-architectural parameters to forecast the expected performance of a program, and they experiment with three standard models: linear regression, multivariate adaptive regression splines, and radial basis function neural networks. Cavazos et al. [10] design their own models that, given some dynamic features of a program such as instruction type distributions and hardware statistics, can output a probability for each transformation showing if this transformation should be applied. In these two works, all the programs in the testing set need to be run at least once so that the dynamic information can be gathered and feed into the prediction models. To the contrary, Agakov et al. [4] exploit only static features, specifically the instruction type distributions inside the loops, and use independent identically distributed model and Markov model to predict the benefits of certain loop optimizations. Instead of using machine learning techniques explicitly, some researchers look for the best set of optimizations with their own heuristics. Triantafyllis et al. [24] explore the optimization space using a decision tree-style algorithm; to reduce the search time, they perform pruning for some non-promising combinations of optimizations and use a heuristic-based static performance estimator to calculate the execution time of a program without actually running it. Pan et al. [20] propose several strategies to select appropriate candidates for performance measurement and compared them with others presented in previous work.

In the computer architecture community, SimPoint [15] is a prominent simulation tool that uses machine learning techniques. While traditional simulation takes an extremely long time and the users usually have to sacrifice the accuracy to reduce simulation time, SimPoint can produce precise simulation results with much shorter running time. The trick is that it constructs a whole picture of the complete execution of a program, by recognizing and weighting each phase. SimPoint intentionally neglects the use of any hardware-based statistics to avoid re-analyzing the performance for different architectural configurations. Instead, it records the frequency of each type of instructions that appear in a basic block. Then all the basic blocks, along with the instruction frequency information, are fed into by the K-means clustering algorithm to find the different phases of the program execution.

Overall, this paper differs from the prior works in the following aspects:

1. The goal of this work is mainly on recognizing the type of a program and demonstrating that this information is useful for compilers and other runtime systems.
2. Instead of designing and tuning ad hoc methods, we explore many state-of-the-art machine learning algorithms and discover some promising models that well fit this particular problem. Besides, we experiment with many dynamic program attributes, various profiling techniques, and different system parameters for building the prediction models and compare their effectiveness on recognizing the program type in order to understand the strength of the prediction models across different platforms.
3. Unlike most previous works, which exploit static information and evaluate the model statically, we use dynamic attributes to identify the program characteristics and address the feasibility of integrating the prediction models into a dynamic optimizing compiler. Therefore, some issues absent in previous research are emphatically discussed in this work: selection and collection of dynamic attributes, running time of the prediction algorithms, and early prediction of the program type.

3. Experiment Setup

This section describes the platform details, including the profiling attributes, compiler options, architecture parameters, and benchmarks used for the experiments. Then, it explains the methodology for profile collection and introduces the machine learning algorithms.

3.1. Platform Details

Machine Type	Description
M1 (extremely low-end)	Issue width: 1, BTB: 64 entries with 1-way, L1: 4K with 4-way, L2: None
M2 (low-end)	Issue width: 1, BTB: 128 entries with 2-way, L1: 8K with 4-way, L2: None
M3 (typical)	Issue width: 2, BTB: 256 entries with 2-way, L1: 16K with 4-way, L2: None
M4 (high-end)	Issue width: 2, BTB: 512 entries with 2-way, L1: 16K with 4-way, L2: 32K with 4-way
M5 (extremely high-end)	Issue width: 4, BTB: 1024 entries with 2-way, L1: 32K with 8-way, L2: 64K with 8-way

Table 2. Architectural configurations. The block size is 32 byte for all caches. There are separate L1 instruction and data caches, and L2 cache is unified, if present.

We choose the ARM [1] processor as our research platform, although we believe the results can be demonstrated on other platforms as well. We use GCC 2.95.2 [3] to generate ARM executables and adopt three optimization options: `-O0`, `-O2`, `-Os`. Among them, the *speed* compilation (`-O2`) means the program is optimized for execution time and is usually the default option for released software; the *space* compilation (`-Os`) means the program is optimized for code size and is often the preferable option if the program is to be deployed to an environment with stringent memory constraints, e.g., embedded systems.

The profiling attributes of interest are categorized into three groups: micro-architectural statistics (M), instruction types (I), and condition codes (C). The micro-architectural statistics includes CPI, branch mis-prediction rate, cache miss rates, and queue utilizations. The instruction types include ALU operations (arithmetic, logic, shift, comparison, multiplication, move), memory operations (load, store), and branch operations. The condition codes category includes the use of 16 condition codes (with relevant pairs

Profiling Attribute	Description	Profiling Attribute	Description
Micro-Architecture Statistics (M)			
CPI	Cycle per instruction	br_mispred	Branch mis-prediction rate
il1_miss_rate	Level 1 instruction cache miss rate	dl1_miss_rate	Level 1 data cache miss rate
itlb_miss_rate	Instruction TLB miss rate	dtlb_miss_rate	Data TLB miss rate
ifq occupancy	Occupancy of instruction fetch queue	ruu_occupancy	Occupancy of register update unit
lsq_occupancy	Occupancy of load/store queue		
Instruction Types (I)			
arithmetic	Arithmetic instructions (including add, sub, rsb, .etc)	logical	Logical instructions (including and, or, xor)
shift	Shift instruction (embedded in shifter operands)	comparison	Comparison instructions (including cmp, cmn)
multiplication	Multiplication instructions	move	Register-to-register data transfer
load	Memory-to-register data transfer	store	Register-to-memory data transfer
branch_taken	Conditional branches that are taken	branch_not_taken	Conditional branches that are not taken
Condition Codes (C)			
cond_code_EQ/NE	Use of condition code EQ/NE	cond_code_CS/CC	Use of condition code CS/CC
cond_code_MI/PL	Use of condition code MI/PL	cond_code_VS/VC	Use of condition code VS/VC
cond_code_HI/LS	Use of condition code HI/LS	cond_code_GE/LT	Use of condition code GE/LT
cond_code_GT/LE	Use of condition code GT/LE	cond_code_AL/NE	Use of condition code AL/NE
cond_flag_X	Update of condition flag N	cond_flag_Z	Update of condition flag Z
cond_flag_C	Update of condition flag C	cond_flag_V	Update of condition flag V

Table 1. Profiling information. There are 9 attributes in the micro-architectural statistics (M) category, 10 attributes in the instruction types (I) category, and 11 attributes in the condition codes (C) cetegory.

combined) and the update of 4 condition flags. Table 1 gives a complete list of these attributes. Besides, Section 4.1 defines a *standard* set of attributes, which is used as the default attribute set in our experiments. We use a modified simulator, based on SimpleScalar/ARM [9], to collect these profiling attributes. To understand the model sensitivity to architecture parameters, we perform the experiments with five different machine configurations, summarized in Table 2. These settings are adapted from the configurations of modern ARM 10 processors [1].

The EEMBC 1.1 suite [2] is chosen as the benchmark studied for two reasons. First, it is the standard benchmark for embedded systems, prevalent both in industry and research community. Second, there are six pre-defined categories of the programs in EEMBC, mainly categorized according to their functionalities: `8_16-bit`, `automotive`, `consumer`, `networking`, `office`, `telecomm`. For example, `cjpeg` (JPEG compression) and `djpeg` (JPEG decompression) are classified as `consumer` programs; `viterb` (Viterbi decoder) and `conven` (convolutional encoder) are classified as `telecomm` programs. We do not use the programs in the `office` class since they are dramatically different with other programs in the same category. A complete list of the programs and categories in the EEMBC suite can be found in [2].

3.2. Data Preparation

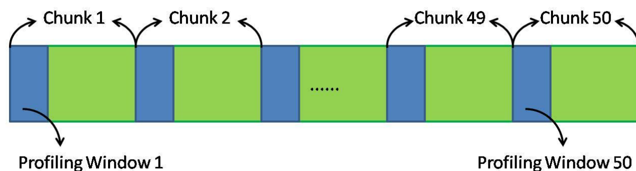


Figure 1. Program execution chunk and profiling window.

With the combination of compiler options, architecture parameters, and profiling attributes, ther is a great number of program runs. For each program run, we divide the whole program execution into 50 equally-sized chunks and perform the profiling for the first several, ranging from 500 to 10000, instructions in each chunk, as Figure 1 illustrates. The standard procedure, if not mentioned otherwise, is to compile the program with `-O2` flag, run the program on a typical machine setting (M3 in Table 2) with the default input data provided along with the benchmark, perform profiling for the first 2000 instructions within a chunk, and collect profiling attributes in the standard set (`STD` in Table 3). Then, the profiling data are used to construct prediction models. To evaluate the models, the *leave-one-out cross-validation* (LOOCV) is performed and the accuracy is reported. Note that there are 50 data instances for each program (since we collect profiles for the 50 chunks through the program run); when performing LOOCV, we exclude all the 50 data instances from the training data and use them for testing. Figure 2 depicts the standard experiment procedure.

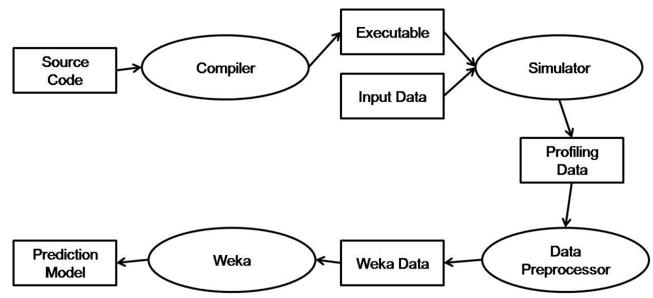


Figure 2. Standard experimental procedure.

3.3. Machine Learning Algorithms

To analyze the profiling data gathered during the program execution, we use Weka 3.4 [26] and various algorithms implemented in this machine learning toolkit. These machine learning algorithms are selected because of their different flavors. For example, the simple `1R` [18] algorithm, which relies on one single attribute for classification, is chosen for baseline comparison. The nearest neighbor (`NN`) [5] algorithm treats each attribute equally and uses Euclidean distance of the attributes to measure the similarity of each data instance; on the contrary, the naive Bayes (`NB`) algorithm assumes that the occurrences of the attributes are independent and calculates the probability that a given data instance belongs to each class. A tree-based classifier takes a divide-and-conquer strategy; C4.5 de-

cision tree (`DT`) [22] determines which attribute to split on (according to the information gain of this splitting), creates branches and sub-trees, and applies the same operation in turn for each sub-tree. Alternatively, a rule-based classifier uses a separate-and-conquer technique; The Ripper classifier (`RP`) [12] greedily attempts to construct a rule to cover as many as possible the instances within a class, separates out those are covered, and continues on those not covered.

Several modern machine learning algorithms are also explored: the logistic regression (`LR`) [23] can be used to predict a categorical type and is able to capture the non-linear relationship between the response (i.e., the prediction output) and the explanatory variables (i.e., the attribute input); the typical multi-layer perceptron (`MLP`) [17] neural networks simulate the way information is processed and propagated in the biological neural networks; the support vector machine (`SVM`) [21] treats the input data as vectors in an N-dimensional space (one dimension for each attribute) and finds a maximum-margin hyperplane (which is defined to be the hyperplane that maximizes the distances from the hyperplane to the closest data points) to separate the vectors into two classes, one on each side of the hyperplane. Finally, `AdaBoost` [14], a provably effective method, produces more accurate predictions by training a series of weak learners with differently weighted data instances (based on the evaluation results from previous rounds) and combining the predictions from them accordingly. In our experiment, the `AdaBoost` classifier uses C4.5 decision trees as weak learners.

4. Evaluation of Experimental Results

In this section, we perform several experiments, each with different variables and goals that try to build the models with varying assumptions and evaluate the models under varying circumstances.

4.1. Effect of Profiling Attributes

The first experiment evaluates the importance of the three categories of profiling information in listed Table 1, separated and combined. The results in Figure 3(a) presents several important observations. First, the micro-architectural information, which is based on the hardware counters and is usually provided by most modern processors, is not an indicative feature for predicting the program type; for `NB` and `MLP`, it is even the least indicative attribute. Second, the condition codes category, which is based merely on the control flow structure of a program and not on the other program behaviors, cannot fully represent the program characteristics. Third, the most indicative information is the instruction types, which cover most of the program behaviors and can mostly be collected by the instruction decoder (by inspecting the instruction opcode). Finally, the combination of the profiling information does not necessarily guarantee the improvement of prediction accuracy. In fact, the results are often negative if non-indicative attributes are added into the profiling attribute set. For instance, the MC bars in Figure 3(a) are generally lower than the M bars, or the C bars, or both. This strongly suggests that the profiling attributes should be carefully chosen in order to enhance the prediction accuracy.

To make the attribute selection sensible, we run a category-blind attribute-ranking algorithm and use several classification algorithms to evaluate the relevance of each attribute. Table 3 summarizes the attributes selected by the some machine learning algorithms. One can see that there are more attributes from the instruction types category, which confirms the observation from Figure 3(a) that instruction types is the most indicative information for recognizing the program type. Besides, some important attributes (e.g., inst_type_LOGIC, inst_type_SHIFT, inst_type_BR_T) are chosen by most of the algorithms. Therefore, from all the attributes selected by the classification algorithms, we carefully choose the ten

Algo.	Attributes		
1R	inst_type_ARITH	inst_type_SHIFT	inst_type_LOGIC
	inst_type_MOVE	cond_code_GE_LT	cond_flag_Z
	cond_code_CS_CC	cond_flag_N	inst_type_MUL
	inst_type_CMP		
RP	inst_type_STORE	inst_type_LOGIC	cond_flag_C
	inst_type_ARITH	cond_code_CS_CC	ruu_occupancy
	cond_code_MI_PL	cond_code_AL	dl1_miss_rate
	lsq_occupancy		
DT	cond_flag_V	inst_type_ARITH	cond_code_AL
	cond_code_GE_LT	inst_type_LOGIC	cond_code_EQ_NE
	cond_code_MI_PL	cond_code_GT_LE	inst_type_MOVE
	cond_code_CS_CC		
NB	inst_type_SHIFT	inst_type_ARITH	inst_type_LOGIC
	dtlb_miss_rate	inst_type_BR_T	inst_type_MOVE
	cond_code_GT_LE	cond_flag_N	itlb_miss_rate
	lsq_occupancy		
LR	inst_type_ARITH	inst_type_SHIFT	inst_type_LOGIC
	inst_type_MOVE	cond_code_GT_LE	inst_type_LOAD
	cond_code_EQ_NE	cond_code_CS_CC	cond_code_MI_PL
	cond_code_HI_LS		
MLP	inst_type_MOVE	inst_type_SHIFT	br_mispred
	cond_code_HI_LS	inst_type_LOGIC	inst_type_BR_T
	cond_code_CS_CC	ifq_occupancy	cond_code_GT_LE
	cond_code_MI_PL		
SVM	inst_type_SHIFT	inst_type_MOVE	inst_type_LOGIC
	inst_type_BR_T	inst_type_CMP	cond_code_HI_LS
	inst_type_MUL	cond_code_GE_LT	br_mispred
	dl1_miss_rate		
STD	inst_type_SHIFT	inst_type_ARITH	inst_type_LOGIC
	inst_type_MOVE	inst_type_MUL	cond_code_CMP
	inst_type_BR_T	br_mispred	dl1_miss_rate
	cond_code_AL		

Table 3. The algorithmically-selected attribute sets and the manually-chosen standard attribute set (`STD`).

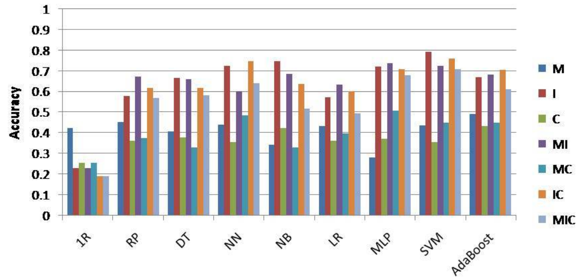
most indicative attributes and define the standard (`STD`) attribute set, which is used in the following experiments.

The second experiment evaluates the representativeness of the algorithmically-selected attribute sets and compare them with the standard attribute set. In Figure 3(b), the `STD` bars are usually higher than the others, meaning that the manually-chosen standard set can highly represent the program features and is useful for the prediction. Furthermore, the accuracies from Figure 3(b) are generally better than the accuracies from Figure 3(a), although the size of the attribute sets used in the second experiment are often smaller; this again proves that including redundant or non-representative attributes for model construction can cause accuracy degradation.

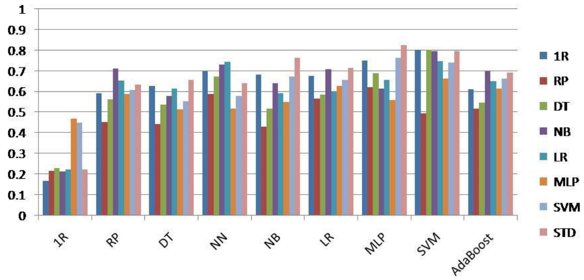
4.2. Effect of Profiling Techniques

In the previous experiments, all the execution chunks are considered to be mutually independent and equally important, although they are sequential in essence during the program execution. In fact, we find it unnecessary to address the order between the chunks since they are merely small pieces of the execution and might be very independent in terms of the whole program execution. Besides, putting the chunk order into consideration may be impractical since a runtime system, without knowing how long the program will run, has no way to conceptually divide the execution into 50 chunks, as we do to the profile data gathered after a complete run. These two observations suggest the treatment of the chunks without considering their order.

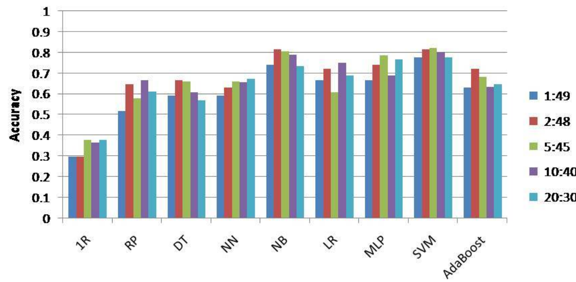
However, if the classification model is to be used in an online program type predictor, it is preferable that the type of a program can be predicted as early as possible and thus it is desired that



(a) Effect of artificially selected attributes.

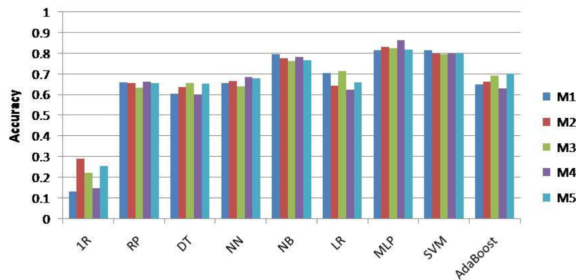


(b) Effect of algorithmically selected attributes.

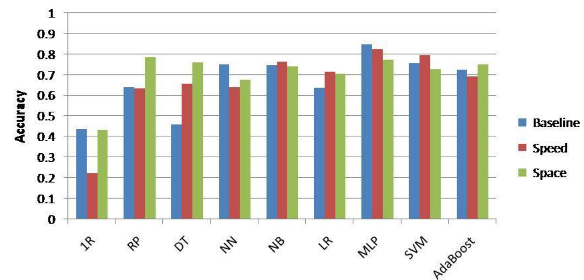


(c) Effect of chunk sequence.

(d) Effect of profiling window size.



(e) Effect of architecture parameters.



(f) Effect of compiler options.

Figure 3. Prediction accuracy for each experiment.

the former chunks (i.e., the first several chunks gathered right after program execution) can be indicative for the prediction's purpose, and this is the main focus of the third experiment. In Figure 3(c), the bar 1:49 indicates that the profiles gathered from the last 49 chunks are used as training data while the profiles gathered from the first chunk is used as testing data, and so on. Notice that the amount of training data is varying in this experiment since we split all the 50 chunks into training data and testing data. As expected, prediction for the first chunk has the lowest accuracy in general, even though all the other 49 chunks are used for model construction. This is mainly because that within the first chunk, some programs are still in their initialization phases (although we have attempted to avoid by skipping certain amount of instructions at the beginning) and the profiles gathered during this period can hardly reflect the typical characteristics of these programs. However, we can see satisfying prediction accuracy when the programs run to the second chunk. This implies that the online predictor could still foresee the type of the running program at a very early stage, using only a small amount of profiling data and having an insignificant profiling overhead.

The fourth experiment studies the profiling window sizes, ranging from 500 to 10000 instructions inside a chunk. The general trend is that with a larger profiling window, the observed behavior of a program is more uniform and the prediction could be more precise. However, a larger profiling windows implies that the hardware cost (for buffering the data) and the runtime overhead (for processing the data) are higher. Fortunately, the results in Figure 3(d) suggest that even the smallest profiling window (with 500 instructions) is enough for collecting profile data as promising as the data gathered from a much larger profiling window.

4.3. Effect of System Parameters

The fifth experiment focuses on the five different architecture parameters listed in Table 2. Figure 3(e) presents an interesting fact: prediction accuracies from all algorithms are surprisingly stable across all different architectural configurations. This confirms the results from Figure 3(a) that micro-architectural statistics itself is not an indicative information. Besides, this also implies that a model built upon a certain platform can be used, with only little loss of prediction accuracy, on other platforms, even though the underlying hardware implementations are dramatically different. This message is particularly useful for embedded systems where processors are often specialized and customized to fit the different

user needs and the varying hardware costs, and a universal predictor is extremely favorable.

The sixth experiment focuses on the three different compiler options: `baseline`, `speed`, `space`. As Figure 3(f) shows, prediction models are more sensitive to the compiler options than to the architecture parameters. The choice of optimization option greatly affects the instructions generated and executed, especially the ALU, memory and condition code oprations. The large discrepancy of prediction accuracies confirms the results from Figure 3(a) that the distribution of instructions and condition codes are more informative attributes than the micro-architectural statistics. However, for most predictors, the accuracies for `speed` and `space` are still close enough, making the models effective for both optimization options that are frequently adopted in the real-world software deployment.

4.4. Running Time of Machine Learning Algorithms

Algorithm	Training Time	Testing Time)
1R	155	7
RP	3177	20.7
DT	914.3	43.7
NN	15	20461.7
NB	172.3	711
LR	10078.7	140
MLP	136629	152
SVM	6277	95
AdaBoost	18316	627

Table 4. Training and testing time of each machine learning algorithm (in seconds).

Table 4 reports the time for performing model construction (i.e., training) and evaluation (i.e., testing), averaged in three runs. Both the training time and the testing time are measured by the modified Java programs implemented in Weka. Because of some extra overhead incurred by the Java virtual machine (e.g. garbage collection), the time measured does not reflect the actual running time for a real online predictor, typically implemented in C or some other more efficient languages. However, the results from Table 4 do demonstrate the relative running time of each machine learning algorithm.

The relationship between training time and testing time is quite irregular across different algorithms, due to the various learning mechanisms they employ. For example, `NN` essentially has no training time at all; it merely puts all the data instances into a two-dimensional table for later calculation in the testing phase. The testing time for `NN`, therefore, is extremely long since the testing instance is compared with all the training instances to see which training instance is the closest and the type of that nearest neighbor is considered to be the type of the testing instance. On the contrary, the `MLP` algorithm spends a long time on finding the weight values of the neural network that best fit the input/output pairs in the training data, but once the computation is completed, it is much faster to calculate the corresponding value of a testing instance. One thing worth noticing is that if the algorithm is used for online prediction, i.e., the testing instances are collected from runtime profiling and predictions occur during runtime, then the testing time, instead of the training time, is the major concern for choosing the most appropriate algorithm. In this sense, `MLP` is considered to be a better candidate than `NN` since `MLP` has a very short testing time, even though it is slow for `MLP` to build the model during the training phase. In fact, most modern machien learning algorithms tend to spend more time on optimizing the sophisticated models to enhance prediction accuracy, and therefore have a higher training-time-to-testing-time ratio. For example, the ratio is 898.9 for `MLP` and 71.9 for `LR`, while the ratio is 22.14 for `1R` and 20.94 for `DT`.

5. Application

5.1. Binary Translation and Optimization

Optimization	Description
Return address stack (RAS)	A software-based return address stack for speeding up the return instruction. A dedicated register is used for a pointer to the top of stack.
Independent N/Z/C/V flag (INF, IZF, ICF, IVF)	A separate register is reserved for the N/Z/C/V flag, making the access to N/Z/C/V flag cheaper (since no shifting is needed).
N-flag-equals-V-flag testing (NEV)	The clause `N=V` is frequently used for checking condition codes (e.g., `GE`, `LT`, `GT`, `LE`). A register is allocated to store the result of this clause to avoid repeated testing.
Special condition flags (SCF)	This optimization extends the previous one and exploits the special semantics of certain condition codes. For example, if an instruction updates the `Z`, `N`, and `V` flags, and the only insturction that uses these flags depend on the condition code `GT`, instead of storing the three condition flags, it is possible simply to remember if the test of `GE` should be successful or not. This transform is beneficial since is reduces the overhead for both updating and checking operations. A dedicated register is used for storing the special semantics. (Extensive data flow analysis is required to make this optmiization correct.)

Table 5. Optimizations in the binary translator.

This section presents an application of the program type recognition: optimizations for binary translators. For the purpose of demonstration, we use a static binary translator that can translate executables for ARM to executables for another MIPS-like processor. For the binary translator, most of the translation overhead comes from the handling of indirect branches and condition code operations and thus many optimizations are developed to eliminate the overhead of these instructions. First, the return address stack (`RAS`) is a mechanism to enhance the handling of return instruction. Without `RAS`, a return instruction is treated as a normal indirect branch and a dynamic table lookup is needed to find out the corresponding target address for the return point. With `RAS`, however, when a function is called, the pair of return addresses (for both source platform and target platform) is pushed onto a separate stack and the return instruction can take advantage of this information and immediately jump to the return address for target platform if appropriate. Besides, there are several optimizations focusing on the condition flags. On an ARM processor, there are four condition flags: `N` (negative), `Z` (zero), `C` (carry), `V` (overflow). One can emulate the condition flags on a MIPS-like processor, using one register to store all the four flags or allocate some flags on the lowest bits of other registers; optimizations `INF`, `IZF`, `ICF`, and `IVF` each allocates a register for storing a particular condition flag for fast access (since shift operations are not necessary for reading/writing the lowest bit). Moreover, one can exploit the logical structure and semantics of the condition codes to remove the redundant operations for updating and checking the condition flags, as optmizations `NEV` and `SCF` do. The seven optimizations discussed above are introduced in [11] and summarized in Table 5.

While these optimizations can effectively reduce the translation overhead and thus improve the performance of the translated program, each of them takes a dedicated register for its own use.

Hence, if there are some register constraints, it is important to determine which combination of these optimizations should be applied to a program to achieve the best performance. For example, when there are only four registers available to the binary translator, it must choose a set of four optimizations (out of totally 32 combinations of these optimizations) to perform for an input program. Searching for the optimal set of optimizations using brute-force method (i.e., trying all the possible combinations) is unappealing since it greatly increases the translation time. Instead, we use the predcition model developed previously to solve this problem.

5.2. Performance Improvement with the Prediction Models

Specifically, we use the machine learning-based prediction model to recognize the type of an input program and determine the best set of optimizations according to its type. First, programs are classified into different categories. In our experiment, the pre-defined categories in the EEMBC suite are adopted. With this categorization, we can run the input ARM program (compiled with `-O2` option) once, collect the profiling data, and use the prediciton model to recognize the type of this program. Next, the class-to-optimization mapping is designed with domain experts's knowledge and experience. For example, given the constraint that only four registers are available, the optimal set of optimizations for class `automotive` is {`RAS, IZF, IVF, SCF`}; for class `consumer`, {`IZF, ICV, NEV, SCF`}; for class `telecomm`, {`RAS, INF, IZF, IVF`}. With this mapping, we can decide the most beneficial optimizations to be performed for this input program during translation. Notice that this approach drastically reduces the translation time, especially the time for deciding what optimizations to be applied to the program, while still keeping the translation quality (i.e., the performance of the translated program), as long as the prediction model has a great accuracy.

Figure 4. Program improvement with the prediction models. The comparison baseline is the optimal suite-level optimizations.

Figure 4 presents the performance improvement of each prediction algorithm. The performance index for this comparison is the cycle count ratio of the input program and the translated program. The baseline is the best optimization configuration for the whole suite: {`RAS, INF, IZF, IVF`}. The `Oracle(Class)` represents that the class of an application can always be recognized correctly, and the optimal configuration for that class is applied. The `Oracle(App)` represents that each application can always be identified correctly, and the optimal configuration for that application is applied; this marks the upper bound of the speedup that can ever be achieved for this input program.

Figure 4 shows that, besides `lR`, all the other prediction models can help improve the performance without exploring the whole optimization space brute-forcefully; in fact, most of the predictors can help choose an optmization combination that is 4% faster than the suite-level configuration. Furthermore, the enhancements from three predictors (`RP`, `LR`, `AdaBoost`) are even greater than `Oracle(Class)`. The reason is that some applications are predicted to be another class and are applied with some even better optimizations because of this mis-prediction. This suggests that the categorization does not best fit the optimization decisions and can be enhanced to make the speedup closer to `Oracle(App)`, the upper bound of performance improvement.

6. Discussion

This section discusses several relevant issues that are not covered in the previous sections, including the applicability of user-provided information about program types, the use of static attributes for machine learning, and the experiments with clustering algorithms.

One might argue that if the user can provide the type of a running program, either through command line arguments or binary executable annotations, the models for predicting the program type seem redundant. In reality, there are several reasons that make an online prediction models essential to a successful runtime system. First, sometimes a program can have different characteristics throughout its whole execution, and this dynamic type changing usually cannot be easily captured by a static type assignment. Take Skype for example. When a user is talking over Skype, the application needs more machine resources performing audio processing, like the programs in the `telecomm` class; however, when the host becomes a *supernode*, the application is busy coordinating the traffic and directing the packets [7], like the programs in the `networking` class. Furthermore, even when a program acts similarly throughout its whole execution, the application developer or user may not have a good idea about the type of the program, especially from a compiler optimization's perspective. Finally, the runtime system may not be able to trust the program type specified by the user since the user may try to grab more machine resources by assigning an incorrect yet profitable program type.

For the consturction of machine learning models, we try to exploit static attributes (e.g., the static count of arithmetic instructions or condition code operations) of the programs as well. However, the results from the experiments are disappointing, as the static attributes do not take into account the loop effect during runtime, which is essential since programs usually spend most of their execution time in several main loops. Moreover, the static attributes are even less effective if the executable is statically linked with some huge standard libraries and therefore the library code outweighs the user code when static information is collected, even though most of the library functions are seldom or never executed. Besides, we also attempt to use clustering techniques, such as `K-means` [16] and `EM` [13] algorithms, for program type prediction. The well known vulnerability of clustering algorithms is that the specified number of groups to be clustered can greatly affect the results. In our experiment, even when the correct number of groups, which is five, is given to the clustering algorithms, the prediction accuracy is still low, ranging from 30% to 40%. Therefore, the prediction results using clustering techniques are not reported in this paper.

7. Conclusion

The problem of determining the best combination of compiler optimizations has perplexed the compiler writers and application developers for a long time. In this paper, we present a novel way to help optimize programs: first, the programs are categorized into different classes according to their functionalities and characteristics and the class-level optimization decision is made by the domain experts; then, the class of an input program is recognized by the machine learning models and the optimal class-level optimizations are applied to obtain the best performance.

To achieve this goal, we explore many state-of-the-art machine learning algorithms to build a program type predictor, using indicative attributes that could be practically provided by the profiling hardware of a reasonable modern processor. Besides, we study the representativeness of profiling attributes, the model sensitivity to system-wide parameters, and the running time of each prediction algorithm.

Finally, we apply this approach to a binary translation/optimization system and show that the binary translator can use the information of predicted program type to decide the most beneficial optimizations to perform for a particular input program, and the translated program is capable of obtaining the best performance improvement. Results have shown that most algorithms studied in this work can achieve greater than 4% performance improvement, compared to the best suite-level optmizations.

References

[1] ARM10 processors. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.set.arm10/index.html.

[2] EEMBC, The Embedded Microprocessor Benchmark Consortium. www.eembc.org.

[3] GCC, The GNU Compiler Collection. gcc.gnu.org.

[4] F. Agakov, E. Bonilla, J. Cavazos, B. Franke, G. Fursin, M. F. P. O'Boyle, J. Thomson, M. Toussaint, and C. K. I. Williams. Using machine learning to focus iterative optimization. In *CGO '06: Proceedings of the International Symposium on Code Generation and Optimization*, pages 295–305, Washington, DC, USA, 2006. IEEE Computer Society.

[5] D. W. Aha and D. Kibler. Instance-based learning algorithms. *Machine Learning*, 6(1):37–66, 1991.

[6] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: a transparent dynamic optimization system. In *Proceedings of the ACM SIGPLAN '00 conference on Programming language design and implementation*, pages 1–12. ACM Press, 2000.

[7] S. A. Baset and H. Schulzrinne. An analysis of the Skype peer-to-peer internel telephony protocol, Dec 2004.

[8] D. Bruening, T. Garnett, and S. Amarasinghe. An infrastructure for adaptive dynamic optimization. In *CGO '03: Proceedings of the international symposium on Code generation and optimization*, pages 265–275, Washington, DC, USA, 2003. IEEE Computer Society.

[9] D. Burger and T. M. Austin. The simplescalar tool set version 2.0. Technical Report 1342, Department of Computer Science, University of Wisconsin-Madison, June 1997.

[10] J. Cavazos, G. Fursin, F. Agakov, E. Bonilla, M. F. P. O'Boyle, and O. Temam. Rapidly selecting good compiler optimizations using performance counters. In *CGO '07: Proceedings of the International Symposium on Code Generation and Optimization*, pages 185–197, Washington, DC, USA, 2007. IEEE Computer Society.

[11] J.-Y. Chen, W. Yang, T.-H. Hung, H.-M. Su, and W. C. Hsu. A static binary translator for efficient migration of ARM-based applications. In *the 6th Workshop on Optimizations for DSP and Embedded Systems*, 2008.

[12] W. W. Cohen. Fast effective rule induction. In *In Proceedings of the Twelfth International Conference on Machine Learning*, pages 115–123. Morgan Kaufmann, 1995.

[13] A. P. Dempster, N. M. Laird, and D. B. Rubin. Maximum likelihood from incomplete data via the EM algorithm. *Journal of the Royal Statistical Society. Series B (Methodological)*, 39(1):1–38, 1977.

[14] Y. Freund and R. E. Schapire. Experiments with a new boosting algorithm. In *International Conference on Machine Learning*, pages 148–156, 1996.

[15] G. Hamerly, E. Perelman, J. Lau, B. Calder, and T. Sherwood. Using machine learning to guide architecture simulation. *J. Mach. Learn. Res.*, 7:343–378, 2006.

[16] J. A. Hartigan and M. A. Wong. A k-means clustering algorithm. *Applied Statistics*, 28(1):100–108, 1979.

[17] S. Haykin. *Neural Networks: A Comprehensive Foundation, 2nd edition*. Prentice Hall, Upper Saddle River, NJ, 1999.

[18] R. C. Holte. Very simple classification rules perform well on most commonly used datasets. *Machine Learning*, 11(1):63–90, 1993.

[19] J. Lu, H. Chen, R. Fu, W.-C. Hsu, B. Othmer, P.-C. Yew, and D.-Y. Chen. The performance of runtime data cache prefetching in a dynamic optimization system. In *MICRO 36: Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, page 180, Washington, DC, USA, 2003. IEEE Computer Society.

[20] Z. Pan and R. Eigenmann. Fast and effective orchestration of compiler optimizations for automatic performance tuning. In *CGO '06: Proceedings of the International Symposium on Code Generation and Optimization*, pages 319–332, Washington, DC, USA, 2006. IEEE Computer Society.

[21] J. C. Platt. Fast training of support vector machines using sequential minimal optimization. In *Advances in kernel methods: support vector learning*, pages 185–208, Cambridge, MA, USA, 1999. MIT Press.

[22] J. R. Quinlan. *C4.5: programs for machine learning*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993.

[23] S. S. le Cessie and J. van Houwelingen. Ridge estimators in logistic regression. *Applied Statistics*, 41(1):191–201, 1992.

[24] S. Triantafyllis, M. Vachharajani, N. Vachharajani, and D. I. August. Compiler optimization-space exploration. In *CGO '03: Proceedings of the international symposium on Code generation and optimization*, pages 204–215, Washington, DC, USA, 2003. IEEE Computer Society.

[25] K. Vaswani, M. J. Thazhuthaveetil, Y. N. Srikant, and P. J. Joseph. Microarchitecture sensitive empirical models for compiler optimizations. In *CGO '07: Proceedings of the International Symposium on Code Generation and Optimization*, pages 131–143, Washington, DC, USA, 2007. IEEE Computer Society.

[26] I. H. Witten and E. Frank. *Data Mining: Practical Machine Learning Tools and Techniques, 2nd Edition*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005.