

# Program Type Recognition for Compiler Optimization

Tzu-Han Hung

Princeton University, USA  
thhung@cs.princeton.edu

Jiunn-Yeu Chen, Wu Yang

National Chiao Tung University,  
Taiwan  
jychen, wuuyang@cs.nctu.edu.tw

Wei Chung Hsu

University of Minnesota, USA  
hsu@cs.umn.edu

## Abstract

*Today's compilers have 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 to compile all their programs. On the other hand, some researchers focus on customizing the optimizations for each program. The former method may hurt the performance of some programs since non-optimal options are applied on them. The latter approach takes care of the variations of programs but suffer from the scalability problem because the optimization space is often huge. This makes it 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 translator and great performance improvement can be achieved.*

## 1. Introduction

Research and development in optimizing compilers have come up with numerous code improvement techniques over the years. Initially, the focus was on code size reduction, such as redundancy elimination. With the effective exploita-

tion of ILP (Instruction Level Parallelism) in processors, more attentions have been paid to code motion, scheduling, speculation, and optimizations to improve cache hierarchy performance. However, many of the powerful code transformation techniques become double-edged sword – they could significantly improve, or degrade, the execution of the target program depending on the underlying microarchitecture.

To more effectively deploy many of the powerful code improvement techniques, compilers have been relied on profiling [11, 22, 15]. Initially, edge count profiles, which give the execution frequency of each basic block, have been commonly adopted. Recent years, some compilers have also exploited value profiles, cache profiles, and alias and dependence profiles.

Even with advanced profiling and whole program analysis, compilers may not be able to determine whether certain transformations will be effective for the target program. Therefore, many of the optimizations implemented in compilers become *optional* – external directives are sought for to guide the optimizer. One outcome of this trend is that many compilers have more than a hundred of options to drive the optimization process. Some researchers have even proposed intelligent search algorithms to locate the better performing combination of options for target programs [30, 10, 4, 29, 25]. In industry, some companies also offer scripts which automatically search through the space of combined options for better optimized code. In practice, application developers often rely on some packaged options, such as the -O2 and -Os compiler flags, to optimize their programs. The situation is more serious for the embedded system because of a diversity of processors and applications.

Besides, dynamic compilation systems are becoming more prevalent in recent years [6, 24, 8]. A dynamic compiler cannot afford expensive whole-program analysis and optimization space exploration. On the other hand, the dynamic compiler has runtime profiles handy and may use them to select more suitable optimizations. However, the overhead involved in required profiling may become excessive.

In this paper, we try to solve the code optimization problem by exploiting two insights. First, *it is more cost effective to package a set of optimizations for a class of programs instead of for one program*. If we can categorize programs into different classes according to their functionalities and characteristics, the optimizer can deploy the pre-packaged set of optimizations for the target program. This program 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 can be used for recognizing the type of a program*. We propose to develop a program type predictor using indicative dynamic attributes. The predicted type of the application program can be used to guide the optimizations. This approach may yield much lower profiling overhead, and could deploy effective optimizations sooner.

In summary, this paper makes the following contributions:

1. We propose a novel approach for optimization space exploration: the type of an application is predicted based on its runtime attributes, and the compiler (or a runtime optimizer) uses this type information to select a set of class-level optimizations for this program.
2. We have studied a combination of system-wide variables and experimented with several state-of-the-art machine learning algorithms and show it is possible to build a robust universal predictor over different platforms.
3. We demonstrate how the program type predictor can help a binary translator for embedded systems to select 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 optimizations.

The rest of the paper is organized as follows. Section 2 gives related work. 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 achieved. Finally, section 6 discusses several relevant issues and section 7 concludes this paper.

## 2. Related Work

Machine learning techniques have been actively studied to assist research in system optimization, mostly because today's computer systems are increasingly more complex [30, 10, 4, 29, 25, 18, 17]. With a large set of compiler optimizations and architectural configurations, it is challenging to understand all the interactions of the components in

a computer system and to deliver the best performance out of it. Therefore, some researchers find it useful to take advantage of modern statistical 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. [30] 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. On 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. [29] 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 and Eigenmann [25] 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 [17] 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.

This paper differs from prior work 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.
3. Most previous works use either static information, which is less indicative of application features, or huge amount of dynamic information, which requires the application run for a long time (or even through completion) for profile collection. Instead, our approach only needs a small amount of dynamic profiling information to build the prediction model and is more suitable for runtime systems. Some issues absent in previous research, such as the selection and collection of dynamic attributes, and early prediction of the program type, are emphatically discussed in this work.

### 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

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.

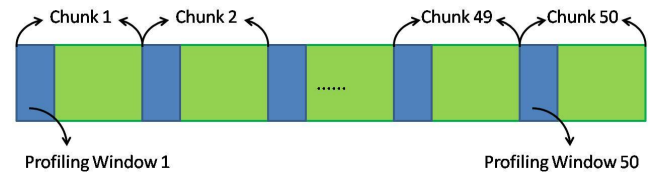
The profiling attributes of interest are categorized into three groups: micro-architectural statistics (M), instruction types (I), and condition codes (C). The micro-architectural statistics includes CPI, branch mis-prediction rate, cache miss rates, and queue utilizations. The instruction types include 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 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

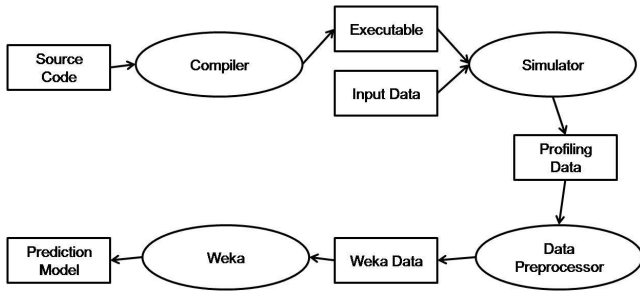
With the combination of compiler options, architecture parameters, and profiling attributes, there is a great number of program runs. For each program run, we divide the whole program execution into 50 equally-sized chunks (based on the instruction count) and perform the profiling for the first several, 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, we use *leave-one-out cross-validation* (LOOCV): for instance, to predict the category of a program, the profiles of this program are used as testing data, while the profiles of all the other programs in the benchmark suite are used as training data. The overall prediction accuracy is the average of accuracy of each program. The evaluation is thus fair since it mimics the situation when a new application is first seen by our prediction and compilation framework.



**Figure 1.** Program execution chunk and profiling window.

Profiling Attribute	Description	Profiling Attribute	Description
<i>Micro-Architecture Statistics (M)</i>			
CPI	Cycle per instruction	br_mispred	Branch mis-prediction rate
il1_miss_rate	Level 1 instruction cache miss rate	dll_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		
<i>Instruction Types (I)</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
<i>Condition Codes (C)</i>			
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_N	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) category.



**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 [31] 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 [21] 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) [23] 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 decision tree (DT) [27] determines which attribute to split on (according to the information gain of this splitting), creates branches and sub-trees,

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.

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) [13] 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) [28] 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) [20] neural networks simulate the way information is processed and propagated in the biological neural networks; the support vector machine (SVM) [26] 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 [16], 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 and present the accuracy of each machine learning algorithm. In each experiment, we change the different sets of parameters for building the models and measure the prediction accuracy to see the effect of these parameters. The prediction is accurate if the category of the input program (and its attributes) is correctly predicted. The prediction accuracy is the percentage of the profiling data that are correctly classified to their corresponding applications.

### 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, increasing the amount of information by combining multiple types of 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.

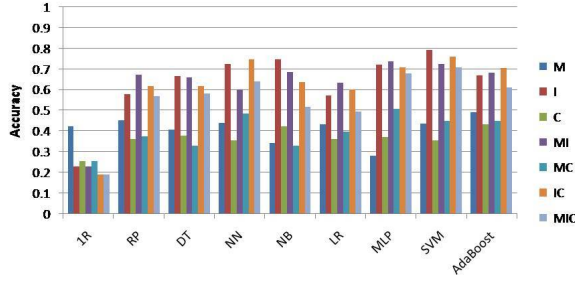
To make the attribute selection sensible, we run a category-blind attribute-ranking algorithm and use several classification algorithms to evaluate the relevance of each attribute.

Algo.	Attributes		
IR	inst_type_ARITH inst_type_MOVE cond_code_CS_CC inst_type_CMP	inst_type_SHIFT cond_code_GE_LT cond_code_FLAG_N	inst_type_LOGIC cond_code_FLAG_Z inst_type_MUL
RP	inst_type_STORE inst_type_ARITH cond_code_MI_PL lsq_occupancy	inst_type_LOGIC cond_code_CS_CC cond_code_AL	cond_code_FLAG_C ruu_occupancy dll_miss_rate
DT	cond_code_FLAG_V cond_code_GE_LT cond_code_MI_PL cond_code_CS_CC	inst_type_ARITH inst_type_LOGIC cond_code_GT_LE	cond_code_AL cond_code_EQ_NE inst_type_MOVE
NB	inst_type_SHIFT dtlb_miss_rate cond_code_GT_LE lsq_occupancy	inst_type_ARITH inst_type_BR_T cond_code_FLAG_N	inst_type_LOGIC inst_type_MOVE itlb_miss_rate
LR	inst_type_ARITH inst_type_MOVE cond_code_EQ_NE cond_code_HI_LS	inst_type_SHIFT cond_code_GT_LE cond_code_CS_CC	inst_type_LOGIC inst_type_LOAD cond_code_MI_PL
MLP	inst_type_MOVE cond_code_HI_LS cond_code_CS_CC cond_code_MI_PL	inst_type_SHIFT inst_type_LOGIC ifq_occupancy	br_mispred inst_type_BR_T cond_code_GT_LE
SVM	inst_type_SHIFT inst_type_BR_T inst_type_MUL dll_miss_rate	inst_type_MOVE inst_type_CMP cond_code_GE_LT	inst_type_LOGIC cond_code_HI_LS br_mispred
STD	inst_type_SHIFT inst_type_MOVE cond_code_AL dll_miss_rate	inst_type_ARITH inst_type_MUL cond_code_CMP	inst_type_LOGIC inst_type_BR_T br_mispred

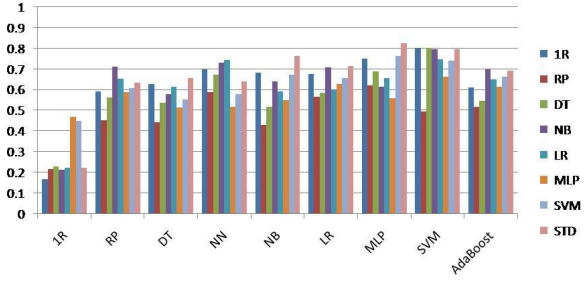
**Table 3.** The algorithmically-selected attribute sets and the manually-chosen standard attribute set (STD).

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

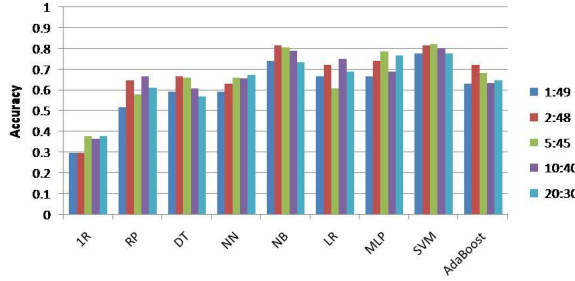
The second experiment evaluates the representativeness of the algorithmically-selected attribute sets and compare them with the standard attribute set. In Figure 3(b), the STD bars are usually higher than the others, meaning that the manually-chosen standard set can highly represent the program features 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



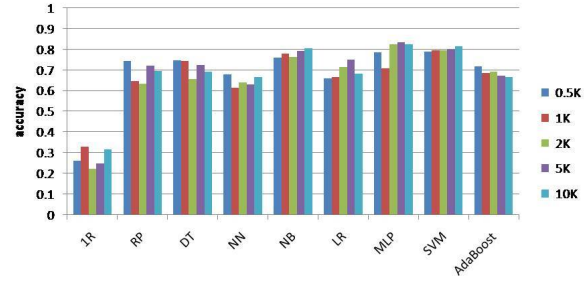
(a) Effect of artificially selected attributes.



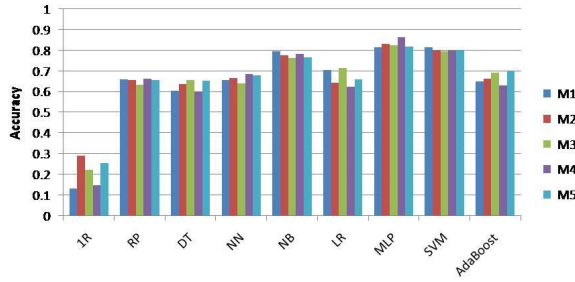
(b) Effect of algorithmically selected attributes.



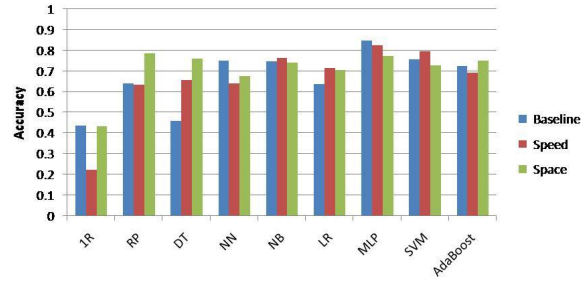
(c) Effect of chunk sequence.



(d) Effect of profiling window size.



(e) Effect of architecture parameters.



(f) Effect of compiler options.

**Figure 3.** Prediction accuracy for each experiment.

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 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 operations. 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.

## 5. Application

### 5.1. Binary Translation and Optimization

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

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 instruction 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 it reduces the overhead for both updating and checking operations. A dedicated register is used for storing the special semantics.

Table 4. Optimizations in the binary translator.

can translate executables for ARM to executables for another MIPS-like processor [12]. 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 optimizations

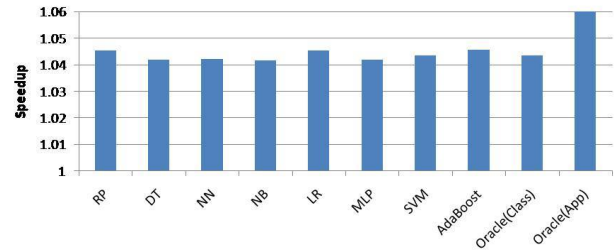
NEV and SCF do. The seven optimizations discussed above are introduced in [12] and summarized in Table 4.

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 prediction 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 prediction model to recognize the type of this program. Next, the class-to-optimization mapping is designed with domain experts' 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 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. The best optimizations for each application are decided in a brute-force manner, and therefore



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

this marks the upper bound of the improvements that can ever be achieved for the input programs.

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 optimization 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)`. This seemingly anomaly is due to the original artificial categorization. If an application has properties of several different categories, there might be a problem to assign this application to a single group. The results show that our experiment can capture and fix this anomaly, by recognizing the better categorization than the original one.

## 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 users can provide the type information of a program, either through command line arguments or executable annotations, the models for predicting the program type seem unnecessary. In reality, there are several reasons that make online prediction models essential to a more efficient runtime system. First, a program can have different characteristics throughout its execution, and this dynamic type changing usually cannot be easily captured by a static type assignment. For example, Skype, a popular VoIP application, may exhibit different dynamic behaviors through the whole execution. When a user is talking over the net, the Skype 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 entire execution, the application

developers or users may not have a concrete idea about the type of the program, especially from a compiler optimization's perspective. Finally, the runtime system may not want to trust the program type specified by the user since the user may provide inaccurate information to influence system resource allocation.

For the construction 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 were not satisfactory, 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 loops. Moreover, static attributes are less effective if the executable is statically linked with some large standard libraries so that the library code outweighs the user code when static information is collected. Besides, we also attempt to use clustering techniques, such as K-means [19] and EM [14] algorithms, for program type prediction. It is well-known that the accuracy of clustering algorithms is often influenced by the specified number of groups to be clustered. 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

Although many powerful compiler analysis and transformation techniques have been developed over the years, it is challenging to select a good combination of the optimizations for a target program. In this paper, we propose a novel way to select a good set of optimizations for target applications. First, the programs are categorized into different classes according to their functionalities and characteristics and the class-level optimization combinations is selected by domain experts. Then, the class of an input program is recognized by the machine learning models and the ideal class-level optimizations are applied to achieve a higher performance.

We have explored many state-of-the-art machine learning algorithms to build a program type predictor, using indicative attributes that could be practically provided by hardware performance monitors in most modern machines. In addition, 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 translator/optimizer for embedded systems, 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 program. Our results have shown that most algorithms studied in this work can achieve greater than 4% performance improvement, compared to the best suite-level optimizations.

## 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](http://www.eembc.org).
- [3] GCC, The GNU Compiler Collection. [gcc.gnu.org](http://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 internet 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] P. P. Chang, S. A. Mahlke, and W. W. Hwu. Using profile information to assist classic code optimizations. *Software Practice and Experience*, 21(12):1301–1321, December 1991.
- [12] 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.
- [13] W. W. Cohen. Fast effective rule induction. In *In Proceedings of the Twelfth International Conference on Machine Learning*, pages 115–123. Morgan Kaufmann, 1995.
- [14] 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.
- [15] J. A. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Transactions on Computers*, C-30(7):478–490, July 1981.
- [16] Y. Freund and R. E. Schapire. Experiments with a new boosting algorithm. In *International Conference on Machine Learning*, pages 148–156, 1996.
- [17] 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.
- [18] M. Haneda, P. M. W. Knijnenburg, and H. A. G. Wijshoff. Automatic selection of compiler options using non-parametric inferential statistics. In *PACT '05: Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, pages 123–132, Washington, DC, USA, 2005. IEEE Computer Society.

- [19] J. A. Hartigan and M. A. Wong. A k-means clustering algorithm. *Applied Statistics*, 28(1):100–108, 1979.
- [20] S. Haykin. *Neural Networks: A Comprehensive Foundation, 2nd edition*. Prentice Hall, Upper Saddle River, NJ, 1999.
- [21] R. C. Holte. Very simple classification rules perform well on most commonly used datasets. *Machine Learning*, 11(1):63–90, 1993.
- [22] W. W. Hwu, S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Warter, R. A. Bringmann, R. G. Ouellette, R. E. Hank, T. Kiyohara, G. E. Haab, J. G. Holm, and D. M. Lavery. The superblock: An effective technique for VLIW and superscalar compilation. *The Journal of Supercomputing*, 7(1):229–248, January 1993.
- [23] G. H. John and P. Langley. Estimating continuous distributions in bayesian classifiers. In *Proceedings of the Eleventh Conference on Uncertainty in Artificial Intelligence*, pages 338–345. Morgan Kaufmann, 1995.
- [24] 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.
- [25] 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.
- [26] 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.
- [27] J. R. Quinlan. *C4.5: programs for machine learning*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993.
- [28] S. S. le Cessie and J. van Houwelingen. Ridge estimators in logistic regression. *Applied Statistics*, 41(1):191–201, 1992.
- [29] 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.
- [30] 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.
- [31] 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.