# Multi-objective Exploration for Practical Optimization Decisions in Binary Translation

SUNGHYUN PARK, University of Michigan
YOUFENG WU, JANGHAENG LEE, and AMIR AUPOV, Intel Corporation
SCOTT MAHLKE, University of Michigan

In the design of mobile systems, hardware/software (HW/SW) co-design has important advantages by creating specialized hardware for the performance or power optimizations. Dynamic binary translation (DBT) is a key component in co-design. During the translation, a dynamic optimizer in the DBT system applies various software optimizations to improve the quality of the translated code. With dynamic optimization, optimization time is an exposed run-time overhead and useful analyses are often restricted due to their high costs. Thus, a dynamic optimizer needs to make smart decisions with limited analysis information, which complicates the design of optimization decision models and often causes failures in human-made heuristics. In mobile systems, this problem is even more challenging because of strict constraints on computing capabilities and memory size.

To overcome the challenge, we investigate an opportunity to build practical optimization decision models for DBT by using machine learning techniques. As the first step, *loop unrolling* is chosen as the representative optimization. We base our approach on the industrial strength DBT infrastructure and conduct evaluation with 17,116 unrollable loops collected from 200 benchmarks and real-life programs across various domains. By utilizing all available features that are potentially important for loop unrolling decision, we identify the best classification algorithm for our infrastructure with consideration for both prediction accuracy and cost. The greedy feature selection algorithm is then applied to the classification algorithm to distinguish its significant features and cut down the feature space. By maintaining significant features only, the best affordable classifier, which satisfies the budgets allocated to the decision process, shows 74.5% of prediction accuracy for the optimal unroll factor and realizes an average 20.9% reduction in dynamic instruction count during the steady-state translated code execution. For comparison, the best baseline heuristic achieves 46.0% prediction accuracy with an average 13.6% instruction count reduction. Given that the infrastructure is already highly optimized and the ideal upper bound for instruction reduction is observed at 23.8%, we believe this result is noteworthy.

CCS Concepts: • **Computer systems organization → Embedded systems**; *Compiler*; Binary Translation.

Additional Key Words and Phrases: Compiler, Machine learning, Binary Translation, Loop Unrolling

**ACM Reference Format:**
Sunghyun Park, Youfeng Wu, Janghaeng Lee, Amir Aupov, and Scott Mahlke. 2019. Multi-objective Exploration for Practical Optimization Decisions in Binary Translation. *J. ACM* 37, 4, Article 111 (October 2019), 19 pages. https://doi.org/10.1145/1122445.1122456

Authors' addresses: Sunghyun Park, sunggg@umich.edu, University of Michigan, 2260 Hayward St, Ann Arbor, Michigan, 48109; Youfeng Wu, youfeng.wu@intel.com; Janghaeng Lee, janghaeng.lee@intel.com; Amir Aupov, amir.aupov@intel.com, Intel Corporation, 3600 Juliette Ln, Santa Clara, CA, 95054; Scott Mahlke, mahlke@umich.edu, University of Michigan, 2260 Hayward St, Ann Arbor, Michigan, 48109.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2019 Association for Computing Machinery.
0004-5411/2019/10-ART111 $15.00
https://doi.org/10.1145/1122445.1122456

This article appears as part of the ESWEEK-TECS special issue and was presented in the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES) 2019

## 1 Introduction

Today, many important optimization decisions are made by heuristics, which often depends on the developers' expertise. With an expert level of understanding of the system and a huge amount of effort, programmers can create effective models that capture architectural characteristics [40]. However, due to the increasing complexity of architectural design, it becomes much harder to build effective human-made models. Additionally, the subtle interactions between software optimization phases add to the exploration space. As a result, heuristics often fail to make good decisions. Stephenson and Amarasinghe showed that the loop unrolling heuristics in Open Research Compiler (ORC) only achieve 16% prediction accuracy for optimal unroll factor [34].

To make effective optimization decisions, researchers build decision models by applying machine learning techniques [12, 28, 34, 35]. Particularly, given its system-wide impact, researchers have studied how to improve the optimization decision for loop unrolling. By relaxing loop-carried dependencies, properly applied loop unrolling can increase Instruction Level Parallelism (ILP) and bring more opportunities for subsequent optimization phases, resulting in significant performance improvement. However, it may cause code bloat or large numbers of register spills when it is applied too aggressively. For the best use of loop unrolling, Stephenson and Amarasinghe suggested defining loop unroll factor prediction as a classification problem [34]. By employing supervised classification techniques, they successfully improve prediction accuracy as well as loop performance over a baseline heuristic. These previous works are built by using statically known information and inserted into a static compiler. Since they target a static compiler, the overhead (computation and memory) of the built model is not considered.

On the other hand, hardware/software (HW/SW) co-design has been extensively studied for mobile and embedded systems to achieve better performance or reduce design cost [1, 7, 38, 39]. A DBT system is a key component in such a co-design process. Since the DBT system conducts optimization during translation [9], the optimization decision has a direct impact on the quality of the translation. Thus, smart optimization decisions are necessary to provide high translation quality. To make better decisions, we examine an approach to tailor machine learning optimization decision models for a code optimizer in DBT. Particularly, we target a mobile processor that supports high-performance applications but still operates in a constrained environment (e.g., a mobile processor for an autonomous vehicle). Unlike previous works [22, 28, 34] in a static compiler, the prediction overhead (e.g., memory usage, performance overhead, and energy consumption) of a decision model may restrict the usage of a complex or large decision model in the mobile system.

As the first step, we choose *loop unrolling* as a representative optimization and investigate five different multi-class classification techniques and their diverse configurations to build its effective decision model. This is a multi-objective exploration observing the relationship between prediction overhead and accuracy. We build our approach with the industrial strength DBT infrastructure. Our code optimizer has an optional optimization for loop unrolling, called *smart unrolling*, that can remove redundant branches [27]. Since smart unrolling can have performance side effects, the built model must make an additional decision on whether to apply smart unrolling. Furthermore, unlike previous works [22, 28, 34] in a static compiler, some useful but expensive analyses like dataflow analysis (DFA) may not be available due to their high overhead in binary translation. Instead, new opportunities to utilize dynamic information (e.g., loop trip count, taken probability of side exits) collected during runtime are studied. These turned out to be important features for unroll factor classifiers.

For experiments, we collected 17,116 unrollable loops from 200 real-life programs and benchmarks in various domains. By employing all available features that might be crucial for loop unrolling decision, we suggest the best classification algorithm for our infrastructure given both prediction

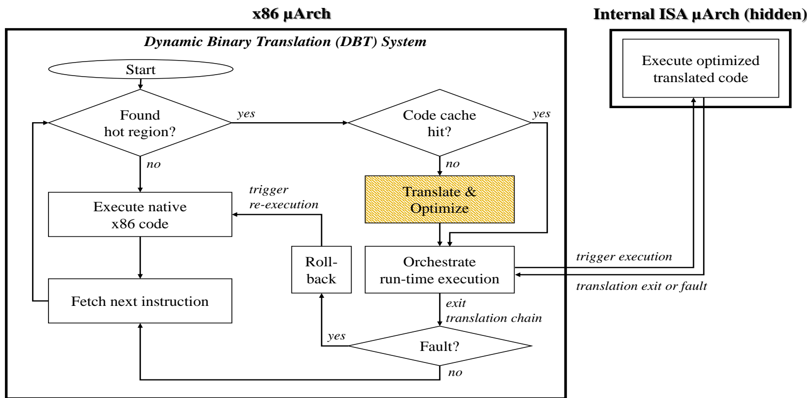



Fig. 1. Big picture of our HW/SW co-designed CPU with a DBT system. It constains two different microarchitectures that each supports its own ISA. The highlighted box illustrates the translation process by a binary translator and a dynamic optimizer.

accuracy and cost. Then, we identify its significant features to prune the feature set and evaluate the model built by using selected important features. As a result, the best affordable classifier that is within the memory/time budgets for the decision process shows a 74.5% of prediction accuracy for optimal unroll factor and realizes an average 20.9% reduction in dynamic instruction count during the steady-state translated code execution when the ideal upper bound for instruction reduction is 23.8%. For comparison, the best current heuristic shows a 46.0% prediction accuracy with an average of 13.6% instruction count reduction.

The major contributions of this work are as follow:

- We show how machine learning techniques can improve loop unrolling decisions for dynamic binary translation on the mobile processor, which is a more challenging environment than static compilers used in previous works [22, 28, 34]. The demanding environment requires a more careful model selection with multiple objectives considered which is not necessary for a static compiler. New opportunities to employ dynamic information is also examined. This approach is instruction set independent and can be extended to other optimization decisions.
- We investigate the relationship between prediction overhead (i.e., time, memory) and accuracy for diverse classification algorithms and their different configurations. As a result, the best classification algorithm is discovered. Then, by applying feature space pruning technique, we provide its major features and suggest the best affordable decision model for our infrastructure given the specific budgets allocated for the decision process.
- We compare the current heuristics in the industrial strength infrastructure and the proposed machine learning based approach. Given that the infrastructure is already highly optimized, the heuristics are well crafted and provide a challenging baseline for comparison.

## 2 Background

### 2.1 Our Infrastructure with Binary Translation

Figure 1 sketches our HW/SW co-designed CPU. Although this CPU only accepts the binary code written in x86, it incorporates a hidden microarchitecture that supports additional Instruction Set Architecture (ISA) internally. By having a DBT system that translates legacy code into internal ISA and orchestrates execution between two different architectures, the internal ISA and its

architectural design can stay invisible from the outside and be innovated without worrying about backward compatibility. To manage program execution, our infrastructure employs the region-level atomic execution similar to [36]. The dynamic binary translation system profiles the execution and constructs optimization regions to translate x86 instructions in the hot traces into internal ISA instructions that will be executed on its optimized microarchitecture. To maximize the performance benefit, it is crucial to produce a high quality of translated code. Therefore, during translation, the dynamic optimizer applies various code optimizations [36], including loop unrolling, to improve the translation quality.

#### 2.1.1 Target Loops

- Reducible (Single entry) loop.
- Innermost loop. The outer loop is only considered when the innermost loop is fully unrolled.
- Both *counting* and *non-counting* loops with a loop invariant trip count [14].

**2.1.2 Smart Unrolling** Our dynamic optimizer has an aggressive version of loop unrolling which is an optional optimization when loop unrolling is enabled. By targeting counting loops, the technique tries to transform the loop structure and exit condition to minimize the number of branches [27]. For example, the optimization can eliminate loop exit branches in copies of a loop body. Also, the optimization phase inserts a run-time check outside of the loop to prevent exceptions (e.g. overflow).

Smart unrolling can have side effects. When the run-time check fails, deoptimization will be triggered to revert the code and degrade the performance. Also, applying smart unrolling to a loop with a low trip count may result in performance degradation due to the overhead of the transformed structure. Thus, smart unrolling should be applied carefully to realize performance improvement.

**2.1.3 Current Heuristics** There are two unrolling heuristics in our dynamic optimizer: *Optimistic* and *Conservative*. By default, *Optimistic* is used to optimize identified hot traces. When speculation with *Optimistic* fails repeatedly [15], *Conservative* is used to generate optimized code with less aggressive optimization. Both heuristics make a decision based on identical information such as trip count, the expected number of post-unroll instructions, etc. The only difference between the two is two parameters: the maximum number of post-unroll instructions and the maximum unroll factor. *Conservative* has lower values than *Optimistic*. Each heuristic predicts an unroll factor that is lower than its upper limits. The maximum unroll factor is set to 8 for *Optimistic* and 4 for *Conservative*, respectively. Also, although smart unrolling may have side effects, current heuristics always apply it on counting loops with the optimistic expectation.

### 2.2 Supervised Multi-class Classification

Supervised classification is a process identifying which set of classes (or labels) a new observation belongs to, based on the learning from training data. This section describes five representative classification techniques as background to deliver their main concepts with the pros and cons.

**2.2.1 k-Nearest Neighbors (kNN)** kNN classifies a new observation based on the majority voting from $k$ number of closest neighbors in training data. The idea is straightforward: find the most similar case from the training database and assign the same label to the new observation. Thus, in our case, kNN will search the most similar loops from the training data and assign the dominant label among them. Without having any form of generalization process on training dataset, training data will be saved and populated directly during the inference. Since kNN scans data points in the training set to find the closest neighbors, its prediction cost is proportional to the size of the training data set. Also, its learned model is unable to be interpreted and give any intuition to the system designers.

**2.2.2 Supporting Vector Machines (SVM)** SVM makes a classification based on the decision boundaries. During training, the technique constructs decision boundaries that separate the training data points. Rather than focusing on minimizing prediction errors on the training set, it tries to minimize the expected generalization loss based on the probabilistic assumption for unseen data. Naturally, SVM is resistant to overfitting [32]. To divide data points under different classes while minimizing the generalization loss, SVM chooses each decision boundary that is farthest away from the observed training data among all possible boundaries. Thus, each selected decision boundary is also called the *maximum margin separator* and the points closest to the separator is called *support vector*. However, it takes a long time to train this model and the learned model is not comprehensible. Fundamentally, *SVM* manages multiple two-class decision boundaries to conduct multi-class classification. Therefore, the prediction cost may become expensive as the number of boundaries increases.

**2.2.3 Decision Tree** A decision tree is a function that makes a decision by conducting a sequence of tests [32]. Each node in a tree checks the value of one of the input features and guides to the next node until the final decision at the leaf node. During training, a decision tree learns what will be tested at each node. If a feature is numerical, the threshold will also be determined for each node. Fundamentally, the decision tree consists of nested conditional statements. Therefore, unlike other machine learning techniques, the learned model is able to be interpreted and easy to visualize. This is a noteworthy property in the sense that the learned model can give system designers insights that they may have been missing. In addition, the worst case for the prediction cost is proportional to the maximum height of the tree. Note, the computation at each node is quite cheap since it is usually just a simple comparison. Thus, if it is able to build an effective tree with the control of the maximum height, the model can be highly practical. However, the technique may suffer from overfitting.

**2.2.4 Random Forest** Random forest manages a multitude of decision trees and makes a prediction by aggregating the predictions from trees in the forest (e.g., majority voting). Since the random forest is essentially a group of decision trees, the learned model is also easy to visualize and is able to give information about the relation between the feature set and the prediction. In general, the technique is able to convey high accuracy and efficiently handle a large dataset with high dimensionality of feature space. Also, the estimate of generalization loss will be computed during training, which can be used to enhance overfitting. However, when the forest grows a multitude of large trees in parallel, it may require a lot of memory [23]. This may cause an overhead to the memory system and increase the inference time affecting the translation time in our case.

**2.2.5 Artificial Neural Network (ANN)** ANN is a learning method that mimics the brain activity mathematically. The model consists of multiple layers: an input layer, hidden layers, and an output layer. Each layer contains a large number of neurons and each neuron is connected to other neurons in the other layer. In this paper, a fully-connected network, whose each neuron is connected to all neurons in the next layer, is assumed. In general, the neural network is known for its outstanding classification accuracy compared to the traditional learning techniques. However, it may need a large network and a large training dataset to achieve satisfactory accuracy [42]. The inference can be very compute-intensive, particularly with a large network. In addition, the decision-making process is like a "black box" so that the internal mechanism would hardly give any intuition to the system designers.

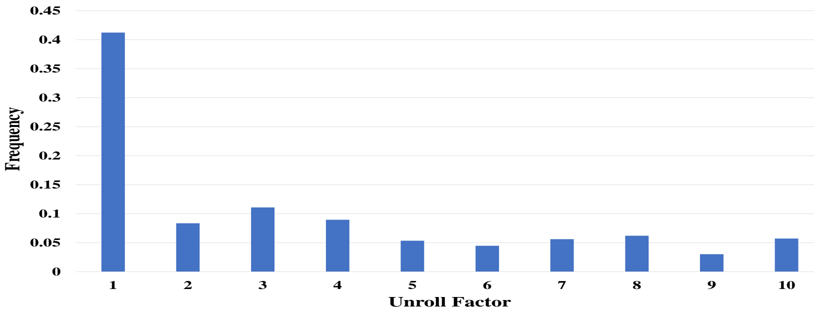



Fig. 2. Distribution of optimal unroll factors.

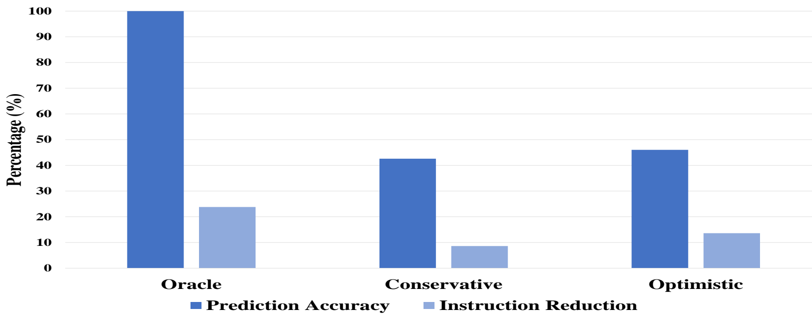



Fig. 3. Evaluation of current heuristic design. Prediction accuracy for optimal loop unrolling decision and its impact on the dynamic instruction count in the resulting optimized code are measured.

## 3 Motivation

Our current DBT system employs heuristic-based decision models designed by industry experts. Given the difficulty of creating effective models that describe subtle interaction between optimization phases while satisfying hard environmental constraints, we notice the pitfall that may exist: hand-made heuristic models might be biased by the designer's experience or insight. Therefore, we recognize the need to not only evaluate their effectiveness, but also develop the automatic and systematic approach to building decision models to lessen the burden required for their design process (e.g., analysis, tuning). Without an automatic method, system designers may need to repeatedly invest a huge amount of efforts to tune their optimization decision whenever other system component(s) are updated.

To investigate the feasibility of such an approach, we initiate the study by focusing on loop unrolling given its system-wide impact. By examining all possible configurations for the loop unrolling decision, the optimal unroll factor for each unrollable loop is identified and the performance of the current heuristic designs (i.e., *Conservative* and *Optimistic*) are measured. Section 5 explains our methodology in more detail. To evaluate if the upper bound of the unroll factor is reasonably set in the current heuristic design (e.g., 8 for *Optimistic*), we explore unroll factors ranging from 2 to 10.

Figure 2 shows the histogram of optimal factors across all collected loops. An optimal unroll factor of 1 represents the case that loop unrolling should not be applied. Other than the leftmost bar, there is no dominant unroll factor. This implies that the optimal unrolling factor varies depending

on the loop characteristics and thus, the unrolling decision should be made carefully. Also, heuristics are unable to predict optimal factors of 9 and 10, which accounts for 8.7% of loops. This suggests the necessity for the higher upper bound on unroll factors.

Figure 3 depicts the effectiveness of current heuristic designs. Note, *Oracle* represents an ideal model that always makes the optimal decisions. Thus, loop unrolling can realize 23.8% of instruction count reduction at most. However, the best heuristic adopted by a dynamic optimizer in our DBT system only shows 46% of accuracy with 13.6% of instruction reduction in the translated code. This suggests the opportunity for improvement with a better decision model.

## 4 Challenges and Opportunities

This section explains why previous works with a static compiler [22, 28, 34] cannot be applied directly to dynamic binary translation for a mobile system. New challenges and opportunities in adapting machine learning based optimization decision models to our environment are threefold:

- **Limited analysis support and increased complexity to make optimization decisions:** Some useful information that is concluded important for a static compiler [34] are not available at the loop unrolling phase in our dynamic binary translation due to their high analysis overhead: dataflow analyses, live range size, instruction fan-in, critical path length, etc. Additionally, the optimization phases in the dynamic optimization are more tightly coupled to each other than a static compiler [20]. Furthermore, in our dynamic optimizer, there is an additional decision for smart unrolling. These add a complication to the analysis and broaden the exploration space for optimization decisions. Therefore, the optimization decision model in a dynamic optimizer should be able to make more complicated decisions with restricted information. A key question is: Can an accurate machine learning model be created without high overhead analyses and handle the additional complexities presented in dynamic binary translation?
- **Strict restrictions on the overhead for an optimization decision:** Since optimization time is an extra run-time overhead in dynamic optimization, the cost for optimization decisions also incur run-time overhead [21]. Consequently, a dynamic optimizer needs to make smart decisions carefully to achieve a good balance between cost and overall performance benefit. To find a good balance, system designers often set strict time/memory constraints on each optimization phase. In our experiment, SVM and nearest neighbor, which are recommended for the static compiler [34], showed good accuracy improvement compared to the baseline heuristics. These techniques, however, presented $8,095\times$ and $660\times$ slower decision-making with significant extra memory requirements. Key questions to address are: Are these methods affordable for a dynamic binary translation in the mobile system? Can a better decision model be built for our environment?
- **The availability of dynamic information:** In dynamic optimization, a new opportunity to use dynamic information arises. Since translation is triggered during runtime, dynamic information, which is more accurate than profiled information for a static compiler [34], is available. However, collecting dynamic information also creates run-time overhead. Thus, the kinds of information that can be collected are limited. A critical question is: What type of dynamic information would be informative for the loop unrolling decision and also affordable in terms of time and storage overheads?

To answer these questions, we explore diverse machine learning techniques to evaluate the feasibility of each approach. Since the classifier will be tested to categorize a new observation based on the achieved knowledge from the learning, generating meaningful training data is a key process to build a good classifier. We describe our data generation process in Section 5 and assess each

| **General loop property** |
|---|
| Number of side exits, Number of outer loops, ... |
| **Constraints from binary translation** |
| Proportion of post-optimization instructions compared to its quota, ... |
| **Benefit: opportunity for the following phases** |
| Number of invariant loads, ... |
| **Benefit: ILP → Unavailable** |
| ~~Number of parallel computations in loop, ...~~ |
| **Side effect: code size** |
| Number of static instructions, ... |
| **Side effect: register pressure → Unavailable** |
| ~~Live range size, Number of uses/defs, ...~~ |
| **Dynamic information** |
| Trip count, Taken probability of side exits, ... |
| **Instruction mix** |
| Ratio of static loads/stores/branches, ... |
| **Smart unrolling** |
| Size of immediate operand in induction variable, ... |

Table 1. A subset of features for loop unroll decision under different categories. The ratio of each static operation is computed by dividing the number of each operation by the number of static instructions. Note, unavailable features (e.g., live range size) in our infrastructure are crossed out. The total of 34 features are extracted and used for the experiment.

classifier in Section 6. Then, Section 7 illustrates our feature selection technique for the machine learning algorithm.

## 5 Data Generation

For each unrollable loop, a set of features and its optimal unroll factor are extracted and combined to generate data for the classifier. Details for data generation process is described in the following sections.

### 5.1 Feature Extraction

For an accurate decision, characteristics of a loop must be captured properly. Thus, important loop information that can affect the decision for loop unrolling is introduced as a feature. For a fair comparison with current heuristics, we use the information that is already available at the optimization phase without an additional profiling or heavy analysis.

For feature selection, we define 9 categories of features across various loop information as presented in Table 1. To manage the quality of translated code regions, our DBT infrastructure puts certain restrictions (e.g., an upper limit for the expected number of post-optimization instructions) in each optimization phase. Thus, the constraints from the binary translation are introduced as features. To estimate the impact of loop unrolling, we also define categories indicating the benefit/side effect. However, the information for instruction level parallelism or register pressure cannot be used for our experiment since it requires dataflow analysis which is unavailable in our loop unrolling phase. In addition, the information for loop characterization, smart unrolling, dynamic information, etc. is employed as a feature. In total, 34 features are extracted and used for the experiment. These 34 features include all 8 features employed by heuristics.

### 5.2 Optimal Factor Exploration

For supervised learning and prediction accuracy measurement, the optimal factor for each set of features should be identified. Thus, we follow an exhaustive approach. Each benchmark trace is executed multiple times with all possible optimization decisions. In this work, unroll factor prediction should make several decisions: (1) Whether loop unrolling should be applied. When loop unrolling is expected to bring performance degradation, the decision model should not enable it. (2) Whether smart unrolling should be applied. (3) Unroll factor. In terms of the unroll factor, we explore the range from 2 to 10. Therefore, each benchmark is executed 19 times with different configurations in loop unrolling:

$$N_{nounroll} + N_{smart} * N_{factor} = 1 + 2 * 9 = 19$$

Each $N_{nounroll}$, $N_{smart}$, $N_{factor}$ represents the number of cases when unrolling is disabled, cases for smart unrolling (Enable/Disable), and cases for unroll factor (2 to 10) respectively. As we investigate 19 configurations, there exist 19 labels that the classifier considers. This exploration space is more than twice compared to previous work [34].

During each run, the dynamic optimizer identifies unrollable loops and dumps their feature sets. Then, the optimizer forces the given configurations on the loops and measures their performance. In this paper, the number of dynamic instructions is measured and used as the performance metric. We discuss our approach further in Section 6. Different performance metrics can be employed depending on the objective of the system design.

After execution, the performance numbers between different configurations are compared to find optimal configurations for each unrollable loop. When the numbers draw, the smaller unroll factor is preferred as it has a smaller side effect (e.g. code size). Secondarily, if the numbers between smart unrolling and regular loop unrolling with the same unroll factor draw, smart unrolling is picked with the optimistic expectation: when the speculative assumption for smart unrolling holds, it would outperform regular loop unrolling. The identified optimal unroll factors for each feature set will be used as training data for supervised learning and testing data for the evaluation.

## 6 Evaluation

### 6.1 Experimental Setup

Classifiers with five different supervised multi-class classification methods are built by using the Python scikit-learn library [29]. To improve the performance, the library implements its core algorithm with Cypthon [8], a package for C-Extension, and compiles it with -O3 optimization level. The selected classification methods include nearest neighbor and SVM which are recommended in previous study for the static compiler [34]. To observe the relationship between prediction cost and its benefit, we explore different configurations for decision tree and random forest. In the library, their configuration can be controlled indirectly by providing the maximum depth and the maximum number of leaf nodes. The parameter setting for each classifier is described in Table 2. For instance, by setting $k = 3$, *Nearest Neighbor* conducts majority voting from three most similar loops. The classifiers which are not covered in the table use the default setting. Due to the compute-intensive nature of neural network algorithm, we inspect a simple network to investigate its applicability.

For performance evaluation, we base our approach on the industrial strength DBT infrastructure and examine the quality of the translated code (i.e., internal ISA). In our experiment, both instruction sets run with a simulator that models our new SW/HW co-designed processor. Since only a functional simulator was available, alternatively, the number of dynamic instructions is measured and used as the performance metric. Although this approach has the restriction as it may not show the overall impact of the optimization directly, researchers often adopt dynamic instruction

| Classifier | Parameter Setting |
|---|---|
| Nearest Neighbor | k = 3 |
| * Decision Tree_0 | - |
| Decision Tree_1 | max.depth = 20, max.leaf = 500 |
| Decision Tree_2 | max.depth = 15, max.leaf = 200 |
| Decision Tree_3 | max.depth = 10, max.leaf = 50 |
| * Random Forest_0 | trees = 10 |
| Random Forest_1 | trees = 5, max.depth = 10, max.leaf = 50 |
| Neural Net | network size = 34 * 100 * 19 |

Table 2. Configurations for the classifiers. Annotated (*) configurations are generated without any restriction on both the maximum depth and the maximum number of leaf nodes.

count as an approximation for the execution time [18, 31]. For example, Ravindar and Srikant use the dynamic instruction count to estimate Worst-Case Execution Time (WCET) [31]. For the same reason, we could not directly measure the overhead of classifiers by embedding them into our dynamic optimizer. Instead, performance improvement analysis and overhead analysis for each prediction technique are conducted separately. Thus, our performance improvement analysis shows the enhancement in the optimized translated code after the execution is stabilized. For overhead analysis, time and memory usage per prediction are examined for each decision model on an Intel Core i7-4700MQ mobile processor assuming the x86 architecture in our CPU design. For a fair comparison with classifiers, we implemented both heuristics algorithms (i.e., *Conservative*, *Optimistic*) in Python with C-Extension and configured with the equivalent optimization level to measure their overhead on the identical conditions.

By following a similar approach with SimPoint [30], we gathered data of 17,116 unrollable loops in the hot traces from 200 real-life programs and benchmarks in various domains. The representative benchmark suites and the breakdown of collected loops for each domain are as follows: embedded/enterprise/games (FPMark [3], Geekbench [4], SYSmark [5], TabletMark [6], 3DMark [2], etc.:49%), performance (SPEC CPU ’06/’17 [10, 19]: 41%), machine learning (MLbench [25], etc.:10%). Diverse operating system environments are also considered: Windows, Linux, Mac, Android, etc.

Throughout the experiment, we conduct the *stratified 5-fold cross-validation* [41]. On average, we train each classifier with 13,652 loops and test it with 3,423 unseen loops. The loops are divided randomly and there is no intersection between them.

### 6.2 Prediction Accuracy

The accuracy is measured by counting the total number of correct classification in comparison to the optimal label out of all test data. For a better understanding of the classifier, we define four different classes for accuracy so that a single prediction can be evaluated in four different perspectives. Each class has a unique definition for "correct classification" as follows:

- ***Exact***: Accuracy for exact prediction in all unrolling, smart unrolling and unroll factor decisions.
- ***Unroll***: Accuracy only for the loop unrolling decision.
- ***Smart***: Accuracy only for the smart unrolling decision.
- ***Dist_n***: Compare the unroll factor only and accept the difference in factors within ’n’. It ignores the decision for smart unrolling.

In the dynamic optimization, the decision of whether to apply the optimization has more importance than the one in the static compilation as the optimization process is a part of a run-time overhead. Therefore, we measure the accuracy of each optimization decision in loop unrolling:

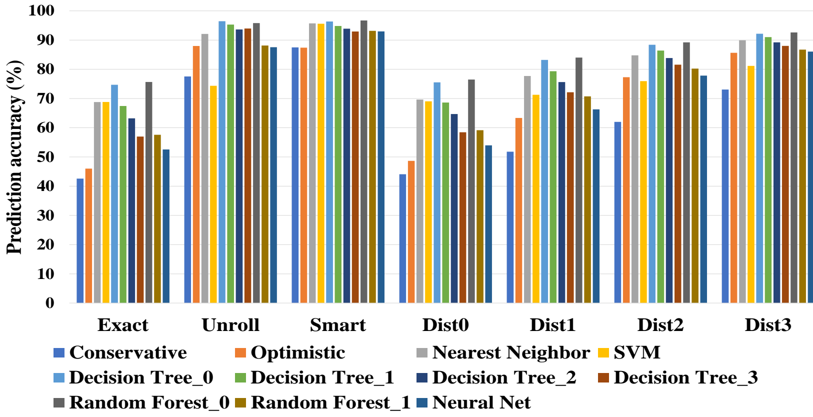
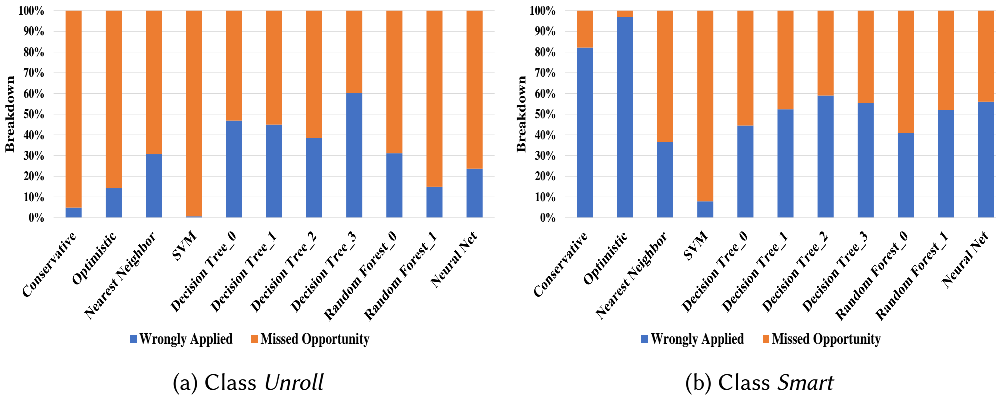



Fig. 4. Prediction accuracy with various classes.



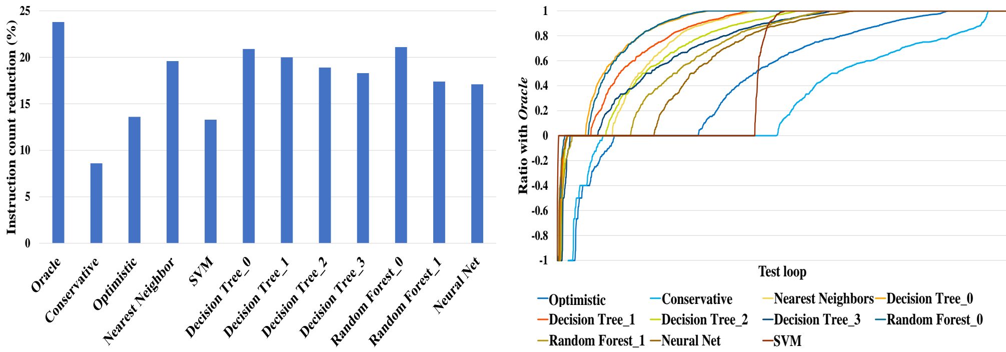

(a) Class *Unroll*

(b) Class *Smart*

Fig. 5. The breakdown of wrong predictions.

*Unroll*, *Smart*. Also, comparison of *Dist_n* gives the indication of how far the prediction goes from the optimal label in terms of the unroll factor.

Figure 4 presents the accuracy of heuristics and classifiers. For *Exact* class, *Conservative* and *Optimistic* show 42.6% and 46.0% of accuracy respectively. Overall, all machine learning techniques outperform heuristics. Especially, *Decision Tree_0* and *Random Forest_0* show the best accuracy which is around 75%. They are the biggest configurations for decision tree and random forest algorithm respectively. Their accuracy goes down with smaller configurations. For *Unroll* class, heuristics work pretty well. *Optimistic* has 88.0% accuracy, which is better than *SVM*, *Random Forest_1*, and *Neural Net*. Interestingly, *SVM* shows the lowest accuracy among all predictors, even lower than *Conservative* although it shows the decent accuracy for *Exact* class. Other classifiers, such as *Decision Tree_0*, surpass the accuracy of heuristics. For *Smart* class, all machine learning models outperform heuristics by presenting the accuracy above 90% even though both heuristics shows desirable accuracy. Although heuristics' optimistic expectation generally holds, this suggests an opportunity for improvement in their current design. For *Dist_n* class, most models increase their accuracy as the allowance in the difference of factors grows. Particularly, the increases in *Optimistic*, *Decision Tree_2*, *Random Forest_1*, and *Neural Net* are notable. This implies they make a good deal of sub-optimal predictions that are close to the optimal labels. However, the accuracy




(a) Geomean of instruction reduction for test loops. (b) The ratio of optimized loops compared to *Oracle*.

Fig. 6. Averaged instruction count reduction and the ratio of optimized loops compared to *Oracle* for each decision model.

change in *SVM* is not significant compared to others. This might be related to its noteworthy low accuracy for class *Unroll*.

To understand the misprediction for *Unroll* and *Smart*, we collect its sources of prediction failures. A prediction failure falls into either one or the other: *wrongly applied*, or *missed opportunity*. Figure 5 illustrates the ratio of wrongly applied and missed opportunities among mispredictions in the decision model. The breakdown for class *Unroll* is shown in Figure 5a. In general, most decision models have turned out to have lots of missed opportunity which suggests a room for performance improvement. Especially, *SVM* has a conspicuous amount of missed opportunity resulting in significantly low accuracy for class *Unroll*. The classifier disables loop unrolling more than necessary, resulting in overly conservative predictions. This can also explain the observation of *SVM* in class *Dist_n*. Given that *SVM* shows high prediction accuracy in *Exact* class, the classifier is expected to have a very high accuracy to figure out when it should not apply loop unrolling. Figure 2 backs up this phenomenon. For around 40% of loops in our benchmark traces, it is the best decision to disable loop unrolling. When comparing *Conservative* and *Optimistic*, the former makes more missed opportunity than the latter although it has less wrong applications of loop unrolling. This is expected when considering the nature of the conservative approach. On the other hand, Figure 5b shows the breakdown for class *Smart*. Notably, many decision models wrongly apply smart unrolling in many cases. Especially, both heuristics have a significantly high ratio of wrongly applied which is greater than 80% and 95% respectively. This implies overly optimistic expectation from heuristics would result in misuse of smart unrolling around 10% of the time in their predictions. The high misuse rate is induced by heuristics' overly optimistic expectation towards the speculative assumptions for smart unrolling. Meanwhile, *SVM* presents its conservative approach by showing a significantly low ratio of wrongly applied.

### 6.3 Performance Improvement In Translated Code

To evaluate the steady-state performance of the translated code, expected instruction reduction for each loop is computed by using collected data. Figure 6 illustrates the performance benefit of loop unrolling with each decision model. Figure 6a represents the average in instruction count reduction for test loops. The first bar, *Oracle*, shows the expected instruction reduction of a perfect predictor which always predicts optimal labels. Therefore, its number (23.8%) will be the ideal upper limit that the decision model can achieve. The best heuristic in our DBT infrastructure presents 13.6% of instruction reduction. All but *SVM* outperforms the best heuristic significantly.

This can be explained by *SVM*'s significant amount of missed opportunity for loop unrolling as presented in Figure 5a. Other classifiers notably outperform the best heuristic. In particular, *Decision Tree_0* and *Random Forest_0* show reductions of around 21%, which is close to the ideal upper limit. Smaller configurations like *Decision Tree_2*, *Decision Tree_3*, and *Random Forest_1* also show notable improvement. Figure 6b illustrates the ratio of loops that are close to the optimal performance. For each test loop, the instruction reduction of each classifier is divided by that of *Oracle*. *Oracle* can not have any negative value in the reduction since it would disable the loop unrolling when the side effect is shown. The calculated ratios are then placed in increasing order to clearly show how much proportion of loops are close to optimal. The points below zero represent the loops suffering from performance degradation due to the wrongly applied loop unrolling. When points are closer to one, corresponding loops are unrolled close to the optimal. In other words, the model with fewer points below zero suffers less from the side effect of loop unrolling while the model with more points close to one has more ideally optimized loops. In both heuristics, a greater number of loops suffer from the side effect of loop unrolling than machine learning techniques. Additionally, they have fewer loops unrolled by the optimal decision. Due to its conservative nature, *SVM* disables more loop unrolling than necessary for many loops. Thus, *SVM* has less loops near one while having many loops on zero. On the other hand, *Decision Tree_0* and *Random Forest_0* are observed to have ideally unrolled loops significantly more than others.

## 6.4 Prediction Overhead Analysis

Figure 7 and Table 3 illustrate the inference time and memory requirement for each decision model respectively. The memory requirements for main memory (e.g. DRAM) is computed by its entire model size while cache is calculated by its worst-case run-time memory footprint for an inference. As *Optimistic* and *Conservative* are essentially the same algorithm with different parameters, their inference times are almost identical. Notably, both inference time and memory requirement of *Nearest Neighbor* and *SVM* are significant. Since *Nearest Neighbor* populates the training data to make a prediction, the data should be stored and accessed during the inference. Thus, the large size of the training data may cause the high prediction overhead. On the other hand, *SVM* makes a prediction by using supporting vectors. Naturally, when the number of supporting vectors is high, the technique would suffer from high memory usage and slow inference speed. In our experiment, 11, 028 supporting vectors are built during training. This approach brings an excessive prediction cost for both memory and inference time (8, 095× slower than heuristics). *Random Forest_0*, *Random Forest_1*, and *Neural Net* present large prediction costs as well. Because the random forest algorithm maintains a group of trees, it requires sufficient memory space to store all tree nodes and computing power to process them to achieve satisfactory prediction speed. The compute-intensive nature of *Neural Net* arises its high prediction cost. Given that a simple network with a single hidden layer is assumed in this experiment, a more complex network is expected to have a higher prediction overhead that would be excessive for our dynamic optimizer. The decision tree model makes a prediction by conducting a sequence of simple tests on the nodes along with the path from the root node to the leaf node. Thus, when the tree has a reasonable height, the model exhibits low prediction overhead.

## 6.5 Choice of Classification Algorithm

Since the optimization time induces a run-time overhead in dynamic optimization, system designers often introduce various budgets on dynamic optimization as the design criteria. Also, due to the limited resource in a mobile processor, there is a restriction on the memory usage. In this section, we evaluate the feasibility of each classification methods by considering time/memory constraints and identify the best affordable method.

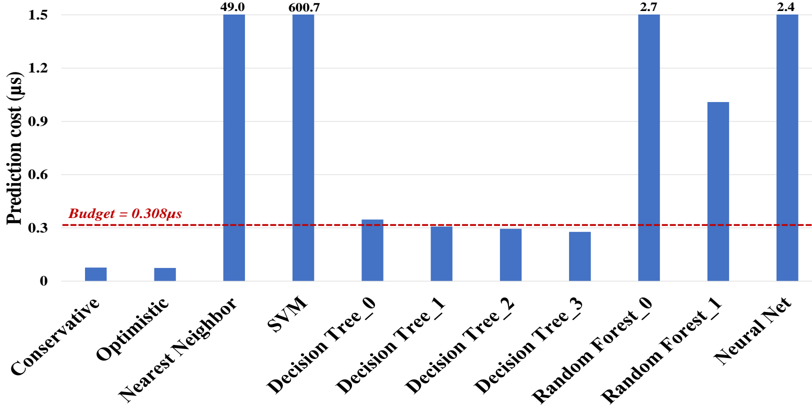



Fig. 7. Inference time for each model with given budget.

| Model | Main Memory (KB) | Cache (KB) |
|---|---|---|
| Nearest Neighbor | 1,391 | 1,391 |
| SVM | 1,500 | 1,500 |
| Decision Trees | < 130 | < 0.66 |
| Random Forests | < 1,122 | < 6.96 |
| Neural Net | 21 | 21 |

Table 3. Memory requirement for each model. The requirements for decision tree and random forest configurations are represented by their largest configurations.

Firstly, we examine the feasibility in terms of time. Based on the rule of thumb widely used by experts in the industry, dynamic binary translation could use no more than $10k * N$ instructions to optimize the region of $N$ instructions. Given that there are plenty of optimization phases and a decision model for the unroll factor is only a small fraction of loop unrolling phase, we assume 0.3% of the total budget as the time constraint for the decision model. Therefore, $30N$ instructions can be used for each unroll factor prediction. To calculate the time budget, we use the information of the processor used for the experiment: Intel Core i7-4700MQ mobile processor (Haswell) with 2.4GHz and its averaged IPC (1.621) [24]. Also, the averaged region size (40) [16, 36] is used for $N$. As a result, 0.308 µs is introduced as the time constraint for each inference and used to evaluate the applicability of each decision model. Figure 7 visualizes the inference cost with the time constraint. Overall, all machine learning techniques but the decision tree algorithm present the inference time above the given budget. *Nearest Neighbor* and *SVM* especially show excessively high prediction overhead. Although the previous work [34] suggests that both techniques would be a good choice for unroll factor prediction in a static compiler, their prediction overheads are exorbitant in our dynamic optimizer. Also, the high inference time of *Neural Network*, which is over the budget, implies that even a simple network has excessive computation overhead for our infrastructure. While *Random Forest_0* presents the highest prediction accuracy, it turned out the decision model is not affordable due to its high prediction cost significantly above the time constraint. *Decision Tree_0*, which has the comparable prediction accuracy to *Random Forest_0*, shows faster inference speed compared to the other classifiers. But its prediction cost is slightly above the constraint. The smaller decision trees, *Decision Tree_1* and *Decision Tree_2*, exhibit relatively high prediction accuracy with fast prediction satisfying the given budget. This implies the necessity to consider the trade-off between prediction accuracy and cost in terms of the tree size.

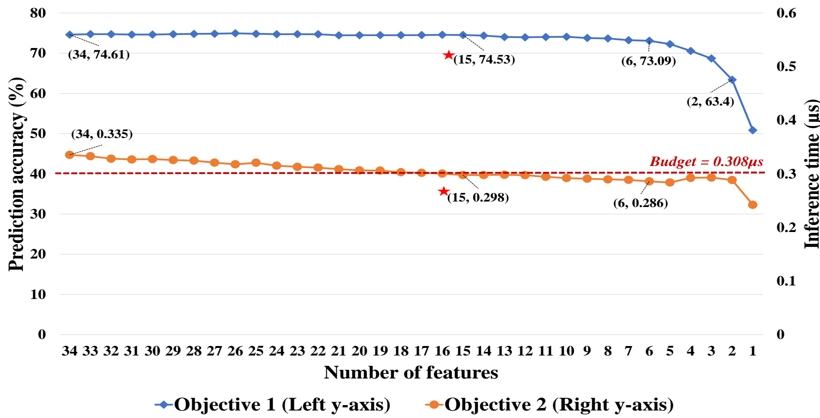



Fig. 8. The change in prediction accuracy and inference time during greedy feature selection process. With Top 15 features (marked with stars), the decision tree model can show an almost identical level of prediction accuracy to the model with all 34 features while satisfying the time constraint.

We also assume 0.3% of the total memory budget as the memory constraint for the decision model. Table 3 represents memory requirements for storing (main memory) and running (cache) each model. Our assumed mobile processor can store all models without difficulty. However, the memory constraint on cache is challenging. Since our mobile processor contains 1 MB of L2 cache, the constraint for cache becomes 3 KB. Thus, only decision tree and random forest with small configuration can meet the restriction.

Overall, the decision tree is the only machine learning based model that satisfies both time/memory budgets on our infrastructure. It is also one of the models with the highest prediction accuracy. Thus, the decision tree would be a good choice as a cost-effective decision model for loop unrolling on our infrastructure. Additionally, the decision tree is an interpretable model which can give insight to system designers. The learned knowledge (e.g., how each feature is treated) from the decision tree can be used to improve the human-made heuristic algorithm.

## 7 Redundant Feature Pruning

To identify the important features and possibly reduce the number of features, we modify the greedy feature selection algorithm [34]. Every iteration, our greedy method drops the least significant feature by examining variation in prediction accuracy when each feature in the feature set is eliminated. Starting from a full feature set (i.e., 34 features in our case), the same process is repeated until no feature is left in the set.

Figure 8 presents how our greedy feature selection technique gradually narrows down the feature set for the decision tree algorithm which is identified as the best classification technique for our DBT system in the previous section. The reduced feature set affects tree size and availability of test that the model can perform at each tree node. Mostly, both prediction accuracy and inference time are observed to decrease as the number of available features gets smaller. Since the decision tree makes a prediction at a leaf node by passing through a series of tests along the path, reducing average depth for leaf nodes brings the improvement in the inference time. The prediction accuracy sustains a similar level until only 6 features are employed and drops significantly after that point. Particularly, the decision tree model can present the almost identical level of prediction accuracy with the model employing a full feature set by using only 15 features (less than 0.1% of difference),

| Top 15 features |
|---|
| **General loop property** |
| • The number of operands |
| • Step of induction variable ✓ |
| • The viability of non-linear loop conversion ✓ |
| **Constraints from binary translation** |
| • Proportion of post-optimization blocks compared to its quota ✓ |
| • Proportion of post-optimization instructions compared to its quota ✓ |
| **Benefit: opportunity for the following phases** |
| • The number of loop invariants |
| • The number of loop invariant loads ✓ |
| **Side effect: code size** |
| • Number of duplicated exit blocks |
| • Number of instructions in duplicated exit blocks |
| **Dynamic Information** |
| • Trip count ✓ |
| • The taken probability of side exits |
| **Instruction mix** |
| • The ratio of static stores |
| • The ratio of memory operations |
| **Smart unrolling** |
| • The size of immediate operand from induction variable ✓ |
| • The size of operand for the loop condition ✓ |

Table 4. Top 15 features for decision tree. Check marks (✓) indicate that the corresponding features are considered in the current heuristic design: 8 out of the top 15 features are employed by current heuristics.

while satisfying the time constraint. Also, the built model requires 0.61KB of space in cache which fits within our memory budget. Thus, in this work, Top 15 features are recommended to use.

Table 4 shows those selected features. Overall, features under various categories defined in Table 1 are chosen. This suggests loop unrolling requires information from diverse aspects to make a good decision. Note, our binary translation puts constraints on upper limits for the expected number of post-optimization instructions and basic blocks. Therefore, the proportion of post-optimization instructions/basic blocks compared to its quota plays an important role in unrolling decision. Top 15 features also include information regarding loop invariants and the number of duplicated instructions/blocks. Each information provides an indication for benefit and side effect from loop unrolling respectively. Dynamic information, such as trip count and the taken probability of side exits, also takes a crucial role by helping the dynamic optimizer to capture the actual iteration count at runtime. On the other hand, the size of the immediate operand from the induction variable and the operand size for the loop condition are meaningful for the decision for smart unrolling. It helps the decision model to estimate the chance of the run-time check failure. The rest of the information helps the optimizer to characterize each loop better.

Interestingly, all 8 features employed in the current heuristic design are chosen as Top 15 features. The significant improvement in the machine learning based model comes from the difference in how each feature is treated (e.g., importance of each feature, threshold for each feature) and the missing features in the heuristic design. By using their own statistical approaches, machine learning based approach is capable of finer tuning in high dimensional space than the hand-made

model. Furthermore, the approach can suggest important features that are missed by experts. Our investigation points out that information, such as the taken probability of side exits and the ratio of memory operations, should be considered for the accurate loop unrolling decision. Note, these 7 missing features are readily available or collected with the negligible overhead at the loop unrolling phase.

## 8 Related Works

There have been efforts to build an optimization decision model by hand [33, 40]. Although these results show quite impressive improvements, it is expected to become continuously harder for compiler designers to create an effective model by hand due to the increasing design complexity. The failures with the software pipelining heuristic which is designed to avoid the side effect from its overly aggressive usage are reported [17, 26]. Also, Stephenson and Amarasinghe [34] showed that their baseline compiler predicts optimal unroll factor only 16% of the time. To improve optimization decision, researchers started to employ machine learning techniques. Particularly, given its system-wide impact, loop unrolling is widely studied. Monsifrot et al. proposed an auto-generation of heuristics for a target processor by using the decision tree algorithm [28]. They built a binary classifier that decides whether to apply loop unrolling or not while leaving unroll factor determination to the existing heuristic. To include the unroll factor in the automation process, Stephenson and Amarasinghe [34] suggested to consider this problem as a multi-class classification problem and solved it by introducing machine learning techniques. Their best classifier shows 65% of prediction accuracy which leads to a 5% speedup (software pipelining disabled) and 1% speedup (software pipelining enabled) over the SPEC 2000 benchmark suite. While the macroscopic approach is similar, our work assumes a dynamic binary translation in the mobile system which is more challenging environment than a static compiler assumed in previous works. As the prediction cost occurs in run-time overhead, five different machine learning techniques are evaluated with varying configurations to identify the cost-efficient machine learning algorithm and show the relation between classifier size and prediction accuracy. Leather et al. suggested the automatic feature generation mechanism for the decision tree model designed to predict the unroll factor [22]. They define the feature space by a grammar and automatized the exploration by using genetic programming and predictive modeling. This approach can be applied to our work to isolate the best set of features for the machine learning-based decision model.

The application of machine learning techniques is also explored for the dynamic compilation. In a Java Just-in-time (JIT) compiler, Cavazos and Moss [11] improved the program speed by employing supervised learning to predict whether blocks would benefit from instruction scheduling optimization. For the blocks expected to gain no advantage from the instruction scheduling, their approach bypasses the optimization to improve the compilation time at runtime. Cavazos and O'Boyle [12] proposed an automatic tuning method for function inlining in a Java JIT compiler. They designed a genetic algorithm that searches a large space of parameter values efficiently. In addition, the best optimization configuration is explored in JIT compiler. Hoste et al. [20] proposed the multi-objective evolutionary search algorithm to find Pareto-optimal in terms of compilation time and execution speed for JIT compiler. As a result, they gained up to 40% of improvement in compilation time and 19% of improvement in steady-state performance over the default setting of Jikes RVM. On the other hand, and O'Boyle [13] used logistic regression to determine which optimization should be applied to each method based on its features in Jikes RVM and achieves 29% of speedup compared to -O2 optimization level.

The comprehensive survey in the field of machine-learning based compilation is well described in the work by Wang and O'Boyle [37].

## 9 Conclusion

To improve the optimization decision model in binary translation, we propose a statistical and automatic approach by employing machine learning techniques. Focusing on loop unrolling, our work examines the performance and feasibility of machine learning models. We evaluate our approach with the industrial strength infrastructure and 17,116 unrollable loops collected from various real-life programs and benchmarks. By considering both prediction accuracy and cost, the decision tree model is identified as the best classification model. Then, through the greedy feature selection method, its significant features are discovered. By using them, we successfully build the effective best affordable decision model that satisfies the given time/memory budgets and greatly outperforms the baseline heuristics by making better optimization decisions.

## References

[1] 2019-02-08. *Intel core i7 embedded processor.* https://ark.intel.com/products/series/122593/8th-Generation-Intel-Core-i7-Processors#@embedded
[2] 2019-06-02. *3DMark.* https://www.3dmark.com/
[3] 2019-06-02. *FPMark.* https://www.eembc.org/fpmark/
[4] 2019-06-02. *Geekbench.* https://www.geekbench.com/
[5] 2019-06-02. *SYSmark.* https://bapco.com/products/sysmark-2018/
[6] 2019-06-02. *TabletMark.* https://bapco.com/products/end-of-life-products/tabletmark/
[7] Felice Balarin, Paolo Giusto, Attila Jurecska, Michael Chiodo, Harry Hsieh, Claudio Passerone, Ellen Sentovich, Luciano Lavagno, Bassam Tabbara, Alberto Sangiovanni-Vincentelli, et al. 1997. *Hardware-software co-design of embedded systems: the POLIS approach.* Springer Science & Business Media.
[8] Stefan Behnel, Robert Bradshaw, Craig Citro, Lisandro Dalcin, Dag Sverre Seljebotn, and Kurt Smith. 2011. Cython: The best of both worlds. *Computing in Science & Engineering* 13, 2 (2011), 31–39.
[9] Edson Borin, Youfeng Wu, Cheng Wang, Wei Liu, Mauricio Breternitz Jr, Shiliang Hu, Esfir Natanzon, Shai Rotem, and Roni Rosner. 2010. TAO: two-level atomicity for dynamic binary optimizations. In *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization.* ACM, 12–21.
[10] James Bucek, Klaus-Dieter Lange, et al. 2018. SPEC CPU2017: Next-Generation Compute Benchmark. In *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering.* ACM, 41–42.
[11] John Cavazos and J Eliot B Moss. 2004. Inducing heuristics to decide whether to schedule. In *ACM SIGPLAN Notices*, Vol. 39. ACM, 183–194.
[12] John Cavazos and Michael FP O'Boyle. 2005. Automatic tuning of inlining heuristics. In *Supercomputing, 2005. Proceedings of the ACM/IEEE SC 2005 Conference.* IEEE, 14–14.
[13] John Cavazos and Michael FP O'boyle. 2006. Method-specific dynamic compilation using logistic regression. *ACM SIGPLAN Notices* 41, 10 (2006), 229–240.
[14] Jack W Davidson and Sanjay Jinturkar. 1996. Aggressive loop unrolling in a retargetable, optimizing compiler. In *International Conference on Compiler Construction.* Springer, 59–73.
[15] James C Dehnert, Brian K Grant, John P Banning, Richard Johnson, Thomas Kistler, Alexander Klaiber, and Jim Mattson. 2003. The Transmeta Code Morphing/spl trade/Software: using speculation, recovery, and adaptive retranslation to address real-life challenges. In *Code Generation and Optimization, 2003. CGO 2003. International Symposium on.* IEEE, 15–24.
[16] Kemal Ebcioglu, Erik Altman, Michael Gschwind, and Sumedh Sathaye. 2001. Dynamic binary translation and optimization. *IEEE Transactions on computers* 50, 6 (2001), 529–548.
[17] Ramaswamy Govindarajan, Erik R Altman, and Guang R Gao. 1994. Minimizing register requirements under resource-constrained rate-optimal software pipelining. In *Proceedings of the 27th annual international symposium on Microarchitecture.* ACM, 85–94.
[18] John L Hennessy and David A Patterson. 2011. *Computer architecture: a quantitative approach.* Elsevier.
[19] John L Henning. 2006. SPEC CPU2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News* 34, 4 (2006), 1–17.
[20] Kenneth Hoste, Andy Georges, and Lieven Eeckhout. 2010. Automated just-in-time compiler tuning. In *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization.* ACM, 62–72.
[21] Chandra Krintz and Brad Calder. 2001. Using annotations to reduce dynamic optimization time. *ACM Sigplan Notices* 36, 5 (2001), 156–167.

[22] Hugh Leather, Edwin Bonilla, and Michael O'Boyle. 2009. Automatic feature generation for machine learning based optimizing compilation. In *Proceedings of the 7th annual IEEE/ACM International Symposium on Code Generation and Optimization*. IEEE Computer Society, 81–91.

[23] Andy Liaw, Matthew Wiener, et al. 2002. Classification and regression by randomForest. *R news* 2, 3 (2002), 18–22.

[24] Ankur Limaye and Tosiron Adegbija. 2018. A Workload Characterization of the SPEC CPU2017 Benchmark Suite. In *Performance Analysis of Systems and Software (ISPASS), 2018 IEEE International Symposium on*. IEEE, 149–158.

[25] Yu Liu, Hantian Zhang, Luyuan Zeng, Wentao Wu, and Ce Zhang. 2018. MLbench: benchmarking machine learning services against human experts. *Proceedings of the VLDB Endowment* 11, 10 (2018), 1220–1232.

[26] Josep Llosa, Mateo Valero, E Agyuade, and Antonio González. 1998. Modulo scheduling with reduced register pressure. *IEEE Transactions on computers* 6 (1998), 625–638.

[27] Uma Mahadevan and Lacky Shah. 1998. Intelligent loop unrolling. US Patent 5,797,013.

[28] Antoine Monsifrot, François Bodin, and Rene Quiniou. 2002. A machine learning approach to automatic production of compiler heuristics. In *International conference on artificial intelligence: methodology, systems, and applications*. Springer, 41–50.

[29] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. 2011. Scikit-learn: Machine learning in Python. *Journal of machine learning research* 12, Oct (2011), 2825–2830.

[30] Erez Perelman, Greg Hamerly, Michael Van Biesbrouck, Timothy Sherwood, and Brad Calder. 2003. Using SimPoint for accurate and efficient simulation. In *ACM SIGMETRICS Performance Evaluation Review*, Vol. 31. ACM, 318–319.

[31] Archana Ravindar and YN Srikant. 2011. Relative roles of instruction count and cycles per instruction in WCET estimation. In *ACM SIGSOFT Software Engineering Notes*, Vol. 36. ACM, 55–60.

[32] Stuart J Russell and Peter Norvig. 2016. *Artificial intelligence: a modern approach*. Malaysia; Pearson Education Limited,.

[33] Vivek Sarkar. 2000. Optimized unrolling of nested loops. In *Proceedings of the 14th international conference on Supercomputing*. ACM, 153–166.

[34] Mark Stephenson and Saman Amarasinghe. 2005. Predicting unroll factors using supervised classification. In *Proceedings of the international symposium on Code generation and optimization*. IEEE Computer Society, 123–134.

[35] Mark Stephenson, Saman Amarasinghe, Martin Martin, and Una-May O'Reilly. 2003. Meta optimization: improving compiler heuristics with machine learning. In *ACM SIGPLAN Notices*, Vol. 38. ACM, 77–90.

[36] Cheng Wang and Youfeng Wu. 2013. TSO_ATOMICITY: efficient hardware primitive for TSO-preserving region optimizations. In *ACM SIGARCH Computer Architecture News*, Vol. 41. ACM, 509–520.

[37] Zheng Wang and Michael O'Boyle. 2018. Machine Learning in Compiler Optimization. *Proc. IEEE* (2018).

[38] Markus Willems, Volker Bursgens, Thorsten Grotker, and Heinrich Meyr. 1997. FRIDGE: An interactive code generation environment for HW/SW codesign. In *1997 IEEE International Conference on Acoustics, Speech, and Signal Processing*, Vol. 1. IEEE, 287–290.

[39] Wayne H Wolf. 1994. Hardware-software co-design of embedded systems. *Proc. IEEE* 82, 7 (1994), 967–989.

[40] Kamen Yotov, Xiaoming Li, Gang Ren, Michael Cibulskis, Gerald DeJong, Maria Garzaran, David Padua, Keshav Pingali, Paul Stodghill, and Peng Wu. 2003. A comparison of empirical and model-driven optimization. *ACM SIGPLAN Notices* 38, 5 (2003), 63–76.

[41] Xinchuan Zeng and Tony R Martinez. 2000. Distribution-balanced stratified cross-validation for accuracy estimation. *Journal of Experimental & Theoretical Artificial Intelligence* 12, 1 (2000), 1–12.

[42] Guoqiang Peter Zhang. 2000. Neural networks for classification: a survey. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)* 30, 4 (2000), 451–462.