Reframing of Classification and Regression Tasks for Predicting the Effects of Compiler Settings on Multiple Embedded Systems

Craig Blackmore[1], Oliver Ray[1], Meelis Kull[1], Md. Geaur Rahman[2], Peter Flach[1], and Nicolas Lachiche[2]

[1] Department of Computer Science, University of Bristol, United Kingdom
{craig.blackmore, oliver.ray, meelis.kull, peter.flach}@bristol.ac.uk
[2] ICube Laboratory, University of Strasbourg, France
{grahman, nicolas.lachiche}@unistra.fr

Abstract. Compiler settings can have a significant impact on the performance of software but the task of finding effective configurations is time-consuming due to the large number of optimizations available and complex interactions between them. Furthermore, effective configurations are dependent on the target program and architecture. Previous work used prior knowledge about the performance of similar training programs in order to predict good configurations but these methods required retraining for each new architecture. In this paper we identify interesting classification and regression tasks for evaluating the performance of compiler configurations on two embedded system architectures. We show how a model learned for one architecture is not directly applicable to the other architecture and we discuss potential ideas for modeling the shift between architectures which in future could allow for the reuse of a single model on many platforms rather than retraining for each new system.

Keywords: compiler optimization, reframing, machine learning

1 Introduction

Compiler optimizations can have a significant impact on the performance of software, however, finding effective compiler configurations (sets of optimizations) is a challenging task due to the large number of optimizations available and complex interactions between them [21].

In earlier work, the state-of-the-art Milepost study [7] searched for performance enhancing configurations using a method called iterative compilation, which tests a target program with several configurations and evaluates the performance of each. Since this is a time-consuming task that potentially needs to be repeated for each target program and architecture, Milepost investigated 1NN and decision tree based approaches which sought to predict effective configurations for a target program given a feature vector describing its structure.

Such approaches required a new predictive model to be learned for each target platform.

Given the cost of training a new model for each platform, it would be highly valuable to be able to learn a model for one platform and reuse it on several other platforms. This process of adapting a model trained in one context in order to target another context is known as reframing [3]. Reframing differs from transfer learning [22] in its emphasis on learning versatile models which can be easily adapted to different contexts, and applying shifts to the inputs and outputs of those models to better adapt them to a new context.

In this paper we study the effects of compiler optimizations on two embedded systems platforms (namely, the 32-bit ARM Cortex-M3 and 8-bit AVR ATmega328P, which we will refer to as P1 and P2 respectively). We produced datasets which can be used to test approaches designed to reuse the results of learning in one context to another context, such as transfer learning [22], context adaptation [8], and reframing [3]. We analyse the task primarily from a reframing point of view by considering possible input and output shifts.

We identify several shifts in context between the two platforms with regard to their hardware specifications, the methodology for producing their datasets and the models learned in each context. We show how both Support Vector Machine (SVM) and Inductive Logic Programming (ILP) approaches gave accuracies higher than simply predicting the majority class for P1, but no classifier performed well on P2.

Given that we can learn a model for P1 but not P2, we argue that a successful reframing approach would not only allow the reuse of the P1 model on P2, but also the creation of a classifier that can predict for P2 even though it was not possible to learn such a classifier directly from the P2 data.

2 Background

This section gives a brief introduction to the task of reframing (Sec. 2.1), an overview of state-of-the-art machine learning efforts to predict effective compiler optimizations (Sec. 2.2) and a description of the target platforms and benchmarks used in our experiments (Sec. 2.3).

2.1 Reframing

Traditional data mining and machine learning algorithms make use of previously collected training data to build models and then make predictions on future data using the models [6,15]. Most of the traditional algorithms assume that the training and test data are drawn from the same distributions and the same domain space. However, it is natural that the distributions of the training and test data may change if the data are collected from different locations. When the distributions of the training and test data change, the algorithms may not perform well or may produce misleading information. In such situations, reframing between the domains of interest can be useful.

Reframing is an approach which deals with context changes between training and deployment environments [1, 3, 16]. The common context changes include dataset shift, task change and representation change [2,14,16,25]. Let, M be the model built from the training data, θ be the context such as dataset shift, X be the deployment data and D_a be the additional data (labelled or unlabelled) that may be available during deployment. If Y be the expected output then reframing can be defined as a function $R(.)$ as follows [12]:

$$Y \leftarrow R(X, M, \theta, D_a)$$

2.2 Compiler Optimizations

The GCC compiler toolchain [9] provides standard optimization levels (O1, O2, O3) which apply an increasing set of optimizations in an attempt to improve execution time at the expense of compile time and/or code size. In practice, these standard levels are often far from optimal, as demonstrated by the Milepost [7] project which used a method called iterative compilation to evaluate the performance of 1000 random configurations of compiler flags on 22 programs.

Since iterative compilation is a time-consuming process, Milepost [7] also explored 1NN and decision tree based methods for *predicting* rather than *searching* for effective configurations. The classifiers were trained using a feature vector which contains a set of 55 attributes that describe various aspects of the code such as number of basic blocks[3] and number of conditional branches. In 1NN, the closest training program (based on feature vectors) is used to predict a configuration for the target program. In the decision tree approach, the feature vector is used to estimate the probability that a configuration will perform within 95% of the maximal speed-up for the target program.

The Milepost features were selected by empirical experiments [19], however, it is not known whether they are the best features for the task. An alternative approach proposed by [23] does not rely on features that describe program structure but instead uses the execution times of training programs as attributes. We test a related approach in our regression task in Sec. 6.

2.3 Target Platforms and Benchmarks

There are several differences between the two platforms that we targeted in this study. Platform 1 (P1) is the ARM Cortex-M3 32-bit microprocessor [4] and Platform 2 (P2) is the AVR ATmega328P 8-bit microprocessor [5] (which belongs to a family of chips used on Arduino development boards). The Cortex-M3 is more complex than the ATmega328P and features a longer pipeline, larger flash and RAM and branch speculation (Table 1). In addition, some GCC flags

[3] A basic block is a sequence of instructions for which there is a single entry point (at the start) and a single exit point (at the end). When a basic block is entered, all of its instructions are guaranteed to be executed.

Table 1. Overview of hardware specifications for Platform 1 and 2

Platform	Processor	Clock Speed	Flash	RAM	Pipeline Stages	Branch Speculation	Floating Point Unit
P1	Cortex-M3 32-bit	24Mhz	128KB	8KB	3	Yes	No
P2	ATmega328P 8-bit	20Mhz	32KB	2KB	2	No	No

are only supported on P1 and not P2 (e.g. scheduling and section anchor optimizations). Neither platform has a floating point unit, so instead the compiler generates code which emulates floating point functionality using the hardware's integer unit.

In order to profile the effects of different compiler optimizations on the two platforms we used the Bristol/Embecosm Embedded Benchmark Suite (BEEBS[4]) [20] which was created by the MAchine Guided Energy Efficient Compilation (MAGEEC[5]) project conducted by the University of Bristol in partnership with industry compiler experts Embecosm[6]. The BEEBS programs were produced in response to the lack of freely available open source benchmarks compatible with embedded platforms, which often have tight resource limits.

3 Experiment 1: Standard Optimization Levels

In Experiment 1 we sought to evaluate the effects of the GCC standard optimization levels (O1, O2, O3) and determine whether O3 really does give the best execution time out of the three. We recorded the execution time for each O1, O2 and O3 when applied to 60 programs on P1 and 61 on P2. There were 48 programs common to the two platforms.

Figure 1 confirms that O3 is indeed generally the best of the three optimization levels. Therefore, we will use O3 as the baseline for comparing the performance of configurations tested in Experiment 2 (Sec 4).

4 Experiment 2: Iterative Compilation

In Experiment 2 we used iterative compilation to search for configurations that performed better than O3. We generated the dataset for P1, whereas the data for P2 were obtained from an independent experiment run externally by Embecosm. As a result, there are several differences between the two datasets which could impact on the ability to learn a model for one platform and apply it to the other. The differences are summarized in Table 2 and discussed in this section.

[4] The latest version of BEEBS is available at `http://beebs.eu`
[5] `http://mageec.org`
[6] `http://www.embecosm.com`

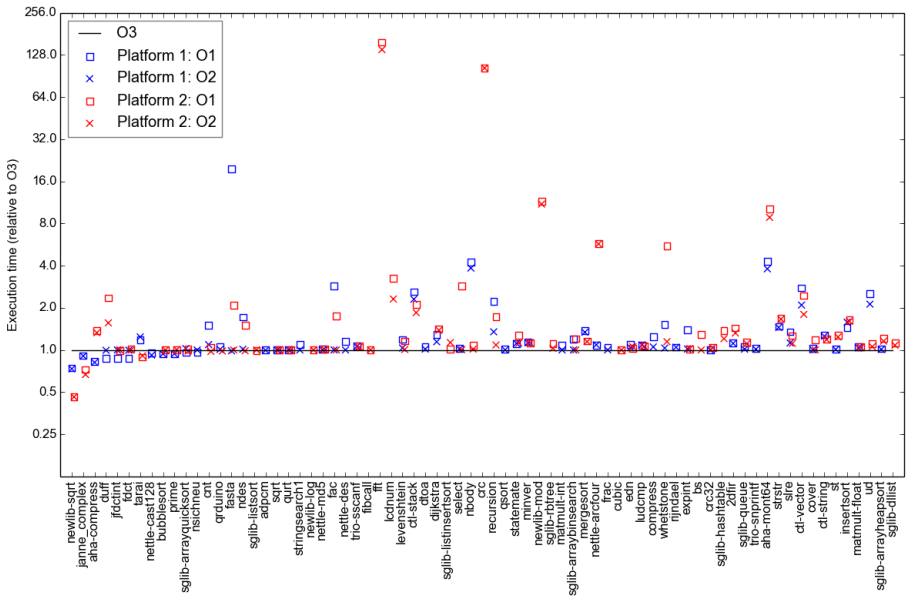


Fig. 1. Experiment 1: execution time of O1, O2 and O3 on Platform 1 and 2

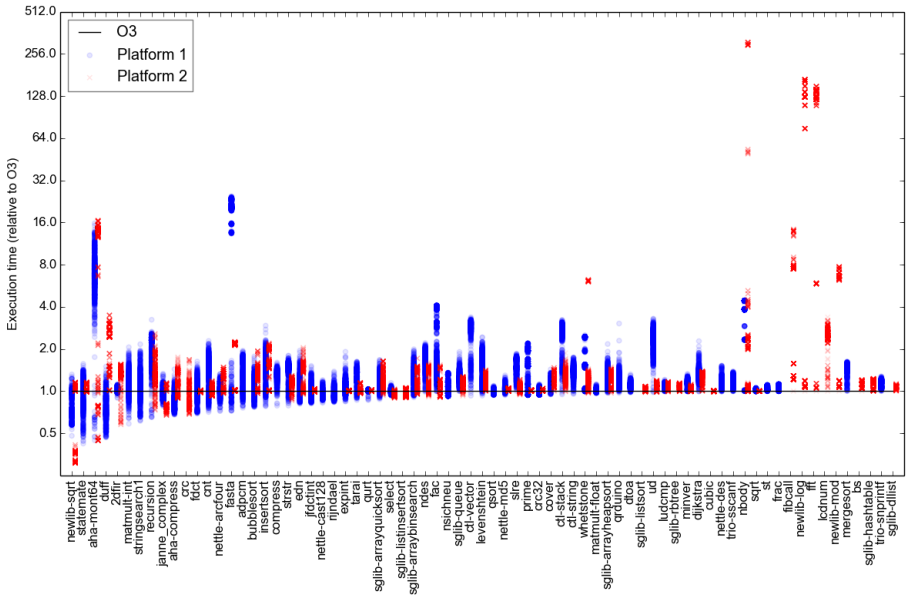


Fig. 2. Experiment 2: execution time for Platform 1 (1000 random configurations of 133 flags) and platform 2 (1024 configurations of 25 passes (generated using Fractional Factorial Design))

Table 2. Differences between P1 and P2 datasets in Experiment 2

Platform	Programs	Configs	Config Type	Config Generation	Unit
P1	60	1000	Flags	Random	Seconds
P2	54	1024	Passes	FFD	Cycles

The experiments for P1 used random iterative compilation to explore the effects of 133 compiler flags on the execution time of 60 programs from BEEBS. A set of 1000 random configurations was generated by selecting O3 and then enabling or disabling each of the 133 flags with 50% probability. The raw data consists of the execution time (in seconds) for each program when compiled with each of the 1000 random compiler configurations

In contrast, the P2 experiments explored the effects of 25 compiler *passes* on 54 programs. A compiler *flag* may enable or disable many compiler *passes*, and the mapping between flags and passes is highly complex. There were a total of 156 passes to choose from, however, preliminary experiments were used to select a subset of the 25 most influential passes on which to focus. Finally, a set of 1024 configurations of the 25 compiler passes were generated using Fractional Factorial Design (FFD) [10] (rather than random generation as in P1), which produced a balanced set of configurations in which each pair of passes appears together the same number of times.

In addition, the datasets for P1 and P2 contain 60 and 54 programs respectively, of which 41 are common to both platforms. The fact that P1 was measured in seconds and P2 was measured in CPU cycles has little impact since all of the machine learning tasks in this paper are based on execution time relative to O3 rather than using absolute values.

The results for Experiment 2 (Fig. 2) show that it is possible to outperform O3 in approximately half of the programs for P1. On P2, there were fewer programs for which O3 could be improved upon. In Sec. 5, we further analyze the results to describe an interesting classification task and suggest how it may lead to a suitable reframing problem.

5 Classification

After analyzing the data from Experiment 2 (Sec. 4) we identified a classification task which seeks to determine whether O3 is at least 10% slower than the best known configuration (O3 slow). This is an interesting task since it is valuable to know whether to simply use O3 for given program or whether it is worth investing extra time in order to find a more optimal configuration.

A visual inspection of the raw data (Fig. 2) for the two platforms strongly suggests that when O3 is slow on P2, then it is also slow on P1. Therefore a potentially very useful task might be to first learn a classifier to predict for which benchmarks it is possible to perform much better than O3 on P1 and then, on top of this, learn a way of predicting for which of those benchmarks it is also

possible to perform much better than O3 on P2. We view this as a reframing problem in which the model for P1 could be used to find the subset that applies to P2. This approach is potentially highly suited to our data since the classes are very well-balanced for P1 (Table 3).

Table 3. Class distribution (%)

Platform	O3 fast	O3 slow
P1	47	53
P2	78	22

Having identified the interesting task of predicting whether it is possible to perform significantly better than O3 on a given program, we tested four machine learning algorithms to establish whether it was possible to learn a suitable model. We tested J48 Decision Tree, 1-Nearest-Neighbor (1NN), Support Vector Machine (SVM) numerical methods as well as an Inductive Logic Programming (ILP) approach (described in Sec. 5.1) which sought to find logical rules that specify the types of programs for which O3 is slow. The former three algorithms were tested in Weka [11] and the latter was implemented in the CProgol4.4 [17] ILP system.

The Milepost [7] feature vector was used to provide attributes for each classifier. The feature vector consists of 55 attributes which describe various characteristics of a program such as number of basic blocks and number of conditional branches. A full list of the 55 features is given in [7]. In this study we used the feature vector for the most time-consuming function of each program, which we identified by profiling with the `gprof` tool on an x86 processor. This allowed us to focus on the function which has the biggest impact on execution time and filter out functions which may have little effect on performance. We also normalized the feature vector relative to the number of instructions (feature 24) to help compare between programs of varying sizes.

This section continues with a description of our ILP approach (Sec. 5.1) and the results for each algorithm when trained on each platform using the whole training set and leave-one-out cross validation (Sec. 5.2). This is followed by an analysis of the shift between the two datasets by training a model for P1 and applying it to P2 and vice versa (Sec. 5.3). Finally, we demonstrate the intuitive rules which our ILP based approach is able to produce (Sec. 5.4).

5.1 Inductive Logic Programming Approach

The aim of ILP [17] is to find a set of hypotheses H that generalize relationships between background knowledge B about a problem and positive and negative examples E^+ and E^- of when a relation does or does not hold. In the case of this study, B consists of knowledge about program structure, E^+ consists

of examples of programs for which O3 is slow and E^- contains examples of programs for which O3 is fast.

In more detail, the background knowledge consisted of predicates of the following form:

```
ft(Ft, P, Val).
large_ft(P, Ft).
small_ft(P, Ft).
qt(P, Ft, Quartile).
non_zero(P, Ft).
```

These predicates can be described as follows: `ft/3` gives the normalized value for feature Ft of the most-time consuming function in program P, `large_ft/2` and `small_ft/2` determine whether the feature Ft for program P is above or below the average for that feature, `qt/3` gives the quartile that feature Ft falls within for program P and `non_zero` specifies whether feature Ft is non zero for program P.

Examples were encoded using the predicate `o3slow/1`. For example, if O3 were slow for program X but fast for program Y, this would be encoded as:

```
o3slow(X).
:- o3slow(Y).
```

Using B and E in the format described above, Progol is able to learn hypotheses such as `o3slow(A) :- ft(ft4,A,0)`. Given that Feature 4 is the number of basic blocks with more than two successors, such a hypothesis could easily be translated into English as follows: program A is slow if its most time-consuming function contains no basic blocks with more than two successors.

We prevented the learning of rules that contained individual feature values given by `ft/3` such as `ft(ft1, program1, 0.75)` in order to discourage overfitting. This was implemented using mode body declarations [18], which are a feature of the Progol system. We did, however, allow literals of the form `ft(Ft, P, 0)` because it is potentially very useful to know that a certain feature is absent from a program. We did not restrict the appearance of any of the other predicates in B.

5.2 Finding a Suitable Algorithm

For P1, only SVM and ILP gave an accuracy better than simply predicting the majority class (Table 4). Both gave an accuracy of 68% which is relatively low. Note that J48 and 1NN performed very well on the whole training set but poorly in cross-validation, therefore we conclude that they both over-fit the data. In fact, 1NN achieves 100% accuracy on the whole training set because it simply memorizes the correct class for each program.

For P2, none of the algorithms were able to outperform the majority class (Table 5). We attribute the difficulty in learning the model to a lack of good features to describe the programs. Indeed, it is not known whether the Milepost

Table 4. Platform 1 - accuracy (%) of model trained using the feature vector (relative to number of instructions) for the most consuming function

Algorithm	Whole training set	Leave-one-out
Majority class	53	53
J48	98	48
1NN	100	45
SVM	73	68
ILP	85	68

Table 5. Platform 2 - accuracy (%) of model trained using the feature vector (relative to number of instructions) for the most consuming function

Algorithm	Whole training set	Leave-one-out
Majority class	78	78
J48	93	69
1NN	100	65
SVM	81	76
ILP	78	78

features are the best for the task at hand. These findings inspired us to adapt a related method called data transposition [23] which does not rely on program features to make predictions (Sec. 6).

Low accuracy aside, we would favor the ILP approach for P1 as it gives human-understandable rules which provide insight into the learned model (Sec. 5.4). On the other hand, it is very difficult to establish the reasoning behind a model learned by a SVM.

5.3 Checking for a Shift

Even though a suitable model was not learned for P2 in Sec. 5.2, the reframing approach suggested in Sec. 5 was still worth pursuing. We tested whether any of the learned models from one platform was directly applicable to the other platform (Table 6). None of the P1 models gave an accuracy better than the majority class when applied to P2. In contrast, each P2 model achieved an accuracy greater than or equal to the majority class when applied to P1. There may be element of luck here however, since the classes for P1 were well-balanced and a random classifier that predicts each class with a 50% probability would also get close to the majority class. In conclusion, the model for one platform was not directly applicable to the other, therefore there appears to be a shift. In Sec. 7 we discuss possible ways in which to reframe the P1 model which will be pursued in future work.

Table 6. Accuracy of model trained on one platform and tested on the other

	Accuracy (%)	
Algorithm	P1 model tested on P2	P2 model tested on P1
Majority class	78	53
SVM	65	57
ILP	70	53

5.4 Rules learned by ILP

This section demonstrates the intuitive rules which ILP can produce. The following Prolog rules were learned in order to predict whether O3 is slow for given program A based on its features:

```
o3slow(A) :- ft(ft4,A,0), ft(ft7,A,0), ft(ft15,A,0), ft(ft19,
  A,0), qt(A,ft46,4).
o3slow(A) :- ft(ft4,A,0), ft(ft7,A,0), ft(ft15,A,0), qt(A,ft46,
  1), qt(A,ft54,1).
o3slow(A) :- ft(ft4,A,0), ft(ft15,A,0), ft(ft19,A,0), qt(A,ft34,
  4).
```

These can be translated into English as follows:

```
rule 1: no basic blocks with more than 2 successors and
        no basic blocks with more than 2 predecessors and
        no basic blocks with more than 500 instructions and
        no direct calls and
        a large number of occurrences of integer constant zero

rule 2: no basic blocks with more than 2 successors and
        no basic blocks with more than 2 predecessors and
        no basic blocks with more than 500 instructions and
        a small number of occurrences of integer constant zero and
        a small number of local variables that are pointers

rule 3: no basic blocks with more than 2 successors and
        no basic blocks with more than 500 instructions and
        no direct calls and
        a large number of unary operations
```

A clear pattern emerged from these rules in that it appears to be possible to significantly outperform O3 on programs whose most time consuming function is fairly simple (i.e. the function does not contain complex control flow between basic blocks and does not have very large basic blocks). Two out of the three rules also require that the program has no direct function calls. Future work will investigate whether there is some bottleneck in the more complex programs that stops them from performing much better than O3.

6 Regression

In Section 5 we found that there is a dataset shift between P1 and P2 when a classification model is built on P1 and tested on P2. In this section, we further explore the existence of a dataset shift between P1 and P2 through applying a regression algorithm on the datasets that contain execution times of a set programs. We use the Java implementation of the regression model called M5P model tree [24] which is publicly available in Weka data mining software repository [11].

The dataset that contains execution times in P1 has 60 programs, whereas the dataset that contains execution times in P2 has 54 programs, as discussed in Section 4. Note that there are some uncommon programs, in P1 and P2, that are excluded in experimentation since our objective is to test the dataset shift by building a regression model on P1 and evaluating it on P2, and vice-versa. Therefore, in the experimentation we use the 41 programs that are common in P1 and P2. Each dataset contains 1000 records, where a record represents a configuration. However, we remove some records having missing values and thereby create datasets having 993 records for P1 and 996 records for P2 without any missing values.

For the regression task we first prepare datasets by following the process of data transposition [23]. The programs are considered as the target column one by one for prediction. That is, if a program is considered as the dependent variable then the running times of the remaining 40 programs using the respective configurations are the independent variables. Therefore, for each platform we create 41 datasets from a single dataset as shown in Fig 3.

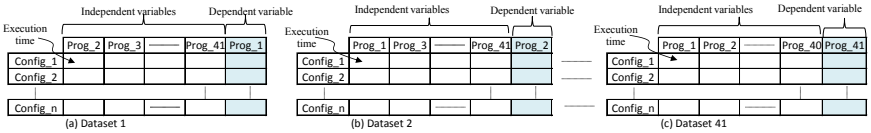
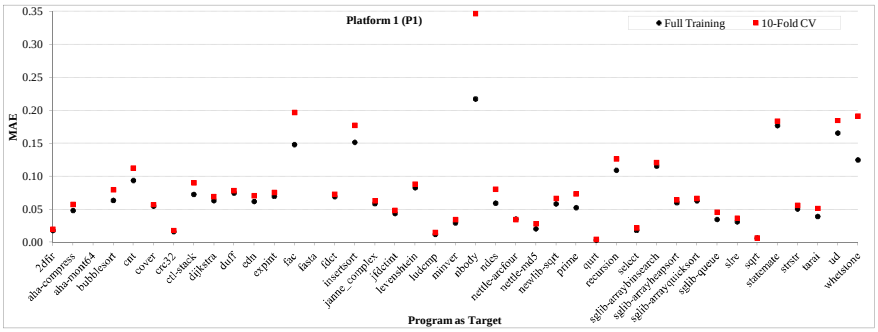


Fig. 3. Dataset preparation

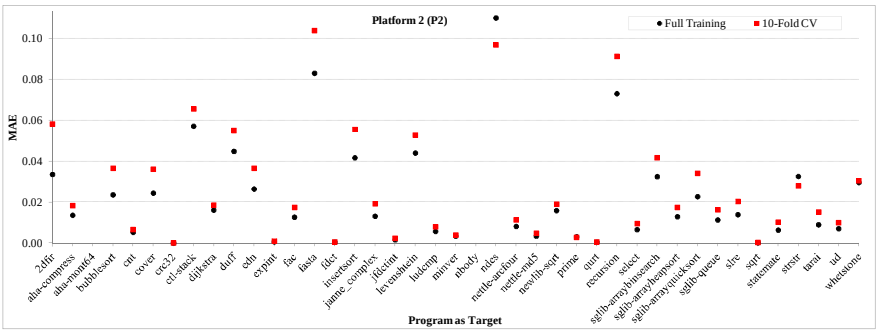
Now we build regression models using the M5P model tree on each dataset and evaluate the models based on the following regression tasks.

1. For each platform, a regression model is built and evaluated on the same dataset (Full-training).
2. For each platform, a regression model is built and evaluated on the same dataset but based on 10 fold cross validation (10-Fold CV).
3. A regression model is built on a platform and evaluated on another platform, where P1 on P2 means the model is built on P1 and evaluated on P2, and similarly P2 on P1 means the model is built on P2 and evaluated on P1. Note that the same program is considered as the target in both platforms.

We evaluate the prediction accuracy of M5P based on the Mean Absolute Error (MAE) [13,26] where a lower MAE value indicates a better result. Figure 4 presents a comparison of the prediction accuracy of M5P in terms of MAE between full-training approach (black circles) and 10-Fold CV approach (red squares). For platform P1 (see Fig. 4(a)), we can see that the MAE values of the full training and 10-Fold CV approaches are closer for all cases. So M5P is able to generate good predictions. We also see a similar indication for P2 as shown in Fig. 4(b).



(a) P1



(b) P2

Fig. 4. M5P regression accuracy in terms of MAE using full-training approach and 10 fold cross-validation approach. (a) P1 and (b) P2.

The experimental results on the full-training approach and the 10-Fold CV approach for a platform motivate us to further explore to see that whether there exists a dataset shift between P1 and P2 or not. We carry out experimentation in which we build a model on one platform and then evaluate model on another platform (i.e. regression Task 3).

We observe that the MAE of a model learned on one platform and evaluated on the other platform is always worse than the MAE of 10-Fold CV approach of the latter platform. For better understanding, we calculate the magnitudes of

the differences of the accuracies as shown in Fig. 5, where the magnitude of P1 on P2 and the magnitude of P2 on P1 are calculated as follows.

$Magnitude\ of\ P1\ on\ P2 = Log_{10}((MAE\ of\ P1\ on\ P2)/(MAE\ of\ 10-Fold\ CV\ on\ P2))$

$Magnitude\ of\ P2\ on\ P1 = Log_{10}((MAE\ of\ P2\ on\ P1)/(MAE\ of\ 10-Fold\ CV\ on\ P1))$

In Fig. 5 we can see that the magnitude is up to the order of 4. The results indicate the existence of a dataset shift between P1 and P2. In this situation, a reframing approach between P1 and P2 can be useful to improve the prediction accuracy. In Section 7, we discuss a few possible approaches to reframe the P1 model. However, they need to be carefully analyzed and evaluated in order to find the suitable one and we plan to carry out the analysis and evaluation in our future study.

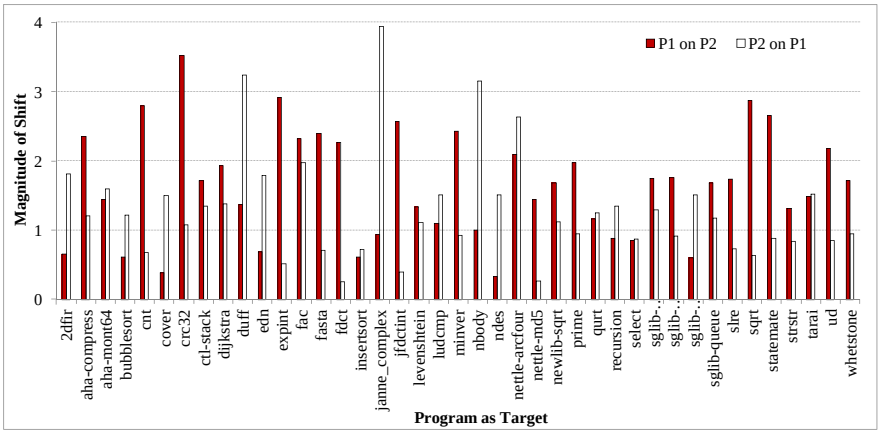


Fig. 5. Magnitude of shift

7 Discussion of Ideas for Reframing

We have identified a number of methods which may allow a P1 model to be reframed for P2. Firstly, we could look for patterns in the feature vector which may identify the subset of programs for which O3 can be improved upon significantly on P1 but not P2. Another idea is to exploit our knowledge of differences between the architectures. A selection of hardware specifications for each platform could be encoded as additional features. We also know that some optimizations (such as scheduling and section anchors) are only supported on P1 and not P2. Therefore, it is possible that some of these unsupported flags are required to achieve the gains that were only seen on P1. Furthermore, other optimizations that are available to both platforms may have a bigger effect on P1 than P2 (since the more complex hardware of P1 may be able to gain more benefit from

them than P2). In future work, we aim to establish whether there are certain flags which improve performance significantly on P1 but not on P2.

8 Conclusion

We have identified classification and regression tasks for predicting the performance of compiler settings and shown that execution times are dependent on the target program and platform. We focused on two platforms and showed that the model learned for one platform was not directly applicable to the other platform. Therefore, we argue that exploring reframing for this problem would be highly valuable for two reasons. Firstly, obtaining training data for each new platform is a time-consuming task, therefore it would be highly cost effective to reuse the model from one platform and apply it to many others. Secondly, we argued that although it was not possible to learn a model for P2 based on P2's training data, it may be possible to learn a model for P1 and apply reframing techniques in order to adapt the model to target P2.

We discussed several ways in which the P1 model may be reframed for P2 by exploiting differences between the two platforms including: identifying flags that are only supported on one platform, identifying flags that are only beneficial/detrimental to one platform, analyzing the subset of programs for which the P1 and P2 class differs and encoding hardware specifications as additional features. In future work we will investigate these options in order to develop a methodology for reframing on the tasks presented in this study.

Finally, this paper focused on execution time, however, the methods can be applied to any metric such as energy consumption, code size and compilation time or a multi-objective goal such as reducing both energy and execution time. Our future work will extend this study further to target energy consumption, which is a critical factor in embedded systems development.

References

1. Ahmed, C.F., Charnay, C., Lachiche, N., Braud, A.: Reframing on relational data. In: International Conference on Inductive Logic Programming (ILP), Nancy, France (2014)
2. Ahmed, C.F., Lachiche, N., Charnay, C., Braud, A.: Dataset shift in a real-life dataset. In: LMCE 2014: First International Workshop on Learning over Multiple Contexts (ECML-PKDD Workshop LMCE) (2014)
3. Ahmed, C.F., Lachiche, N., Charnay, C., Braud, A.: Reframing continuous input attributes. In: Proceedings of the 2014 IEEE 26th International Conference on Tools with Artificial Intelligence. pp. 31–38. IEEE (2014)
4. ARM: Cortex-M3 technical reference manual (Revision: r1p1) (2006)
5. Atmel: ATmega48A/PA/88A/PA/168A/PA/328/P datasheet (2014)
6. Baralis, E., Chiusano, S., Garza, P.: A lazy approach to associative classification. Knowledge and Data Engineering, IEEE Transactions on 20(2), 156–171 (2008)
7. Fursin, G., Kashnikov, Y., Memon, A.W., Chamski, Z., Temam, O., Namolaru, M., et al.: Milepost GCC: Machine learning enabled self-tuning compiler. International Journal of Parallel Programming 39(3), 296–327 (2011)

8. Ganin, Y., Lempitsky, V.: Unsupervised domain adaptation by backpropagation. arXiv preprint arXiv:1409.7495 (2014)
9. GCC: GCC, the gnu compiler collection. http://gcc.gnu.org/ (2015), [Accessed 02/04/2015]
10. Gunst, R.F., Mason, R.L.: Fractional factorial design. Wiley Interdisciplinary Reviews: Computational Statistics 1(2), 234–244 (2009)
11. Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P., Witten, I.H.: The weka data mining software: an update. ACM SIGKDD explorations newsletter 11(1), 10–18 (2009)
12. Jose, H.O., Ricardo, B.P., Kull, M., Flach, P., Chowdhury, F.A., Lachiche, N., Martynez-Uso, A.: Reframing in context: A methodology for model reuse in machine learning. AI Communications (submitted) (2015)
13. Junninen, H., Niska, H., Tuppurainen, K., Ruuskanen, J., Kolehmainen, M.: Methods for imputation of missing values in air quality data sets. Atmospheric Environment 38(18), 2895–2907 (2004)
14. Kull, M., Flach, P.: Patterns of dataset shift. In: LMCE 2014: First International Workshop on Learning over Multiple Contexts (ECML-PKDD Workshop LMCE) (2014)
15. Kuncheva, L.I., Rodriguez, J.J.: Classifier ensembles with a random linear oracle. Knowledge and Data Engineering, IEEE Transactions on 19(4), 500–508 (2007)
16. Moreno-Torres, J.G., Raeder, T., Alaiz-Rodríguez, R., Chawla, N.V., Herrera, F.: A unifying view on dataset shift in classification. Pattern Recognition 45(1), 521–530 (2012)
17. Muggleton, S., De Raedt, L.: Inductive logic programming: Theory and methods. The Journal of Logic Programming 19, 629–679 (1994)
18. Muggleton, S.: Inverse entailment and progol. New Generation Computing 13(3-4), 245–286 (1995), http://dx.doi.org/10.1007/BF03037227
19. Namolaru, M., Cohen, A., Fursin, G., Zaks, A., Freund, A.: Practical aggregation of semantical program properties for machine learning based optimization. In: Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems. pp. 197–206. ACM (2010), http://doi.acm.org/10.1145/1878921.1878951
20. Pallister, J., Hollis, S.J., Bennett, J.: BEEBS: Open benchmarks for energy measurements on embedded platforms. arXiv:1308.5174v2 [cs.PF] (2013)
21. Pallister, J., Hollis, S.J., Bennett, J.: Identifying compiler options to minimize energy consumption for embedded platforms. The Computer Journal (2013)
22. Pan, S.J., Yang, Q.: A survey on transfer learning. Knowledge and Data Engineering, IEEE Transactions on 22(10), 1345–1359 (2010)
23. Piccart, B., Georges, A., Blockeel, H., Eeckhout, L.: Ranking commercial machines through data transposition. In: Workload Characterization (IISWC), 2011 IEEE International Symposium on. pp. 3–14. IEEE (2011)
24. Quinlan, J.R., et al.: Learning with continuous classes. In: 5th Australian joint conference on artificial intelligence. vol. 92, pp. 343–348. Singapore (1992)
25. Quionero-Candela, J., Sugiyama, M., Schwaighofer, A., Lawrence, N.D.: Dataset shift in machine learning. The MIT Press (2009)
26. Rahman, M.G., Islam, M.Z.: Missing value imputation using a fuzzy clustering-based em approach. Knowledge and Information Systems pp. 1–34 (2015)