A Logic Programming Approach to Predict Effective Compiler Settings for Embedded Software

CRAIG BLACKMORE, OLIVER RAY and KERSTIN EDER

Department of Computer Science, University of Bristol
Merchant Venturers Building, Woodland Road, Bristol, BS8 1UB, United Kingdom
(*e-mail:* {craig.blackmore, oliver.ray, kerstin.eder}@bristol.ac.uk)

submitted 29 April 2015; revised 3 July 2015; accepted 14 July 2015

Abstract

This paper introduces a new logic-based method for optimising the selection of compiler flags on embedded architectures. In particular, we use Inductive Logic Programming (ILP) to learn logical rules that relate effective compiler flags to specific program features. Unlike earlier work, we aim to infer human-readable rules and we seek to develop a relational first-order approach which automatically discovers relevant features rather than relying on a vector of predetermined attributes. To this end we generated a data set by measuring execution times of 60 benchmarks on an embedded system development board and we developed an ILP prototype which outperforms the current state-of-the-art learning approach in 34 of the 60 benchmarks. Finally, we combined the strengths of the current state of the art and our ILP method in a hybrid approach which reduced execution times by an average of 8% and up to 50% in some cases.

KEYWORDS: inductive logic programming, embedded system, compiler optimisation

1 Introduction

This paper applies Inductive Logic Programming (ILP) [Muggleton and De Raedt 1994] to the task of predicting compiler flags to minimise the execution time of software running on embedded systems. This is an important problem because the choice of compiler flags can have a very significant impact on the performance of software [Fursin et al. 2011]; but it is a hard problem because individual flags can interact in complex ways while the sheer number of them makes brute force search infeasible [Pallister et al. 2013b]. Furthermore, different programs and platforms benefit from different sets of optimisations.

We build on the work of Milepost [Fursin et al. 2011], which tested 1-Nearest-Neighbour (1NN) and decision tree methods for reducing the execution time of a target program given a feature vector describing its characteristics. One of the key contributions from the Milepost project is the Milepost GCC compiler which produces a Prolog-encoded structural description of C programs based on the Intermediate Representation (IR) used internally by GCC. While Milepost simply used

the IR to generate a predefined feature vector of 56 features for machine learning, we seek to apply ILP directly to the IR in order to automatically discover relevant features for the learning problem. A major advantage of targeting the IR directly is that it retains potentially valuable structural information for predicting compiler flags which is otherwise lost when flattening into a feature vector.

We aim to test our hypothesis that a relational first-order logic approach which uses IR can exploit structured relations in the IR to predict better configurations. In order to test our hypothesis we made the following contributions. We generated a data set by measuring the execution time of 60 benchmarks compiled with various combinations of 133 compiler flags on an embedded system development board. We extended the evaluation of the 1NN approach (which Milepost found worked the best) by applying it to our larger and more diverse training set. We tested two ILP methodologies for relating program features to good and bad flags: our first approach (ILP+FV) uses the Milepost feature vector supplemented with extra relations generated using Prolog queries; our second approach (ILP+IR) seeks to discover new features by analysing Milepost's Prolog-encoded intermediate representation (ILP+IR). In both cases we used the ILP system CProgol4.4 [Muggleton 1995] to learn associations between good and bad flags (those which should be enabled or disabled for good performance) and program features. We show how learning from IR in the ILP+IR method outperforms the Milepost 1NN approach. Our experimental data and Progol scripts are available online at `github.com/craigblackmore/logiflag`.

This study targets execution time, but in principle, the methodology is applicable to any measurable metric that is influenced by the compiler. Possible alternative metrics include memory usage, code size, compilation time and energy consumption as well as multi-objective goals such as reducing both time and energy.

2 Background

This section begins with a brief introduction to the task of finding performance enhancing compiler optimisations (Sec. 2.1) followed by an overview of the state-of-the-art Milepost project which proposed 1NN and decision tree based solutions to the problem (Sec. 2.2). Finally, there is a brief summary of Inductive Logic Programming (ILP) which is central to our methodology (Sec. 2.3).

2.1 Compiler Optimisation

The GCC compiler offers four standard optimisation levels `O0`, `O1`, `O2` and `O3`, each of which turns on an increasing set of flags believed to reduce execution time of an average program at the expense of code size and/or compilation time [GCC, the GNU Compiler Collection 2015]. Adding or removing flags from these standard settings can significantly influence performance for any given program. Finding the best flags is a challenging task due to the large number of available flags and the potentially complex interactions between them [Pallister et al. 2013b].

2.2 Milepost

The state-of-the-art Milepost study [Fursin et al. 2011] aimed to reduce software execution times by predicting performance enhancing configurations (sets of flags) using 1NN and decision tree algorithms. Training data was produced by flattening programs into a feature vector comprising 56 statistical aggregates that summarise the code. The study focused on optimising the most time consuming function of each program. Effective configurations were found for each program using a method called iterative compilation that measured performance using 1000 random compiler configurations. As this is very time consuming, Milepost investigated predictive approaches that could be trained on this data and then applied to unseen programs. These predictive approaches were tested on 22 programs from the cBench suite [Collective Benchmark 2012] using leave-one-out cross validation.

The 1NN algorithm gave the best predictions and worked as follows [Fursin et al. 2011]. Given test program X, the feature vector was used to find the closest training program Y. Then one of two methods was applied. Either the best known configuration for Y is selected as the prediction (1NN-best) or a configuration computed as most likely to perform well is chosen (1NN-prob). In Sec. 4.1 we re-evaluate the 1NN approach and show it performs less well on our larger data set.

We are particularly interested in the Prolog-encoded IR which Milepost extracts and uses to calculate its feature vector. We seek to learn directly from the Milepost IR, which retains more information than flattening programs into feature vectors. Moreover, the IR is based precisely on the internal representations over which the GCC optimisations themselves operate [Fursin et al. 2008].

The Milepost IR consists of 48 predicates which describe the structure and properties of the code. The control-flow is encoded by the following predicates: `bb_p(BB)` defines a basic block[1], `edge_src(E,BB)` defines an edge[2] `E` from basic block `BB` and `edge_dest(E,BB)` which defines an edge to basic block `BB`. The following facts encode an edge `ed0` from basic block `bb0` to basic block `bb1`:

```
bb_p(bb0). bb_p(bb1). edge_src(ed0,bb0). edge_dest(ed0,bb1).
```

As well as describing the structure of the program, the Milepost IR also contains a range of flags which describe properties of specific parts of the program. For example, a single statement may have several flags associated with it such as the following, which means that statement `st0` contains memory operations:

```
stmt_flag(st0,has_mem_ops).
```

In order to apply propositional machine learning, Milepost extracts a feature vector for each function using a set of Prolog rules which analyse the IR to produce counts and averages that describe various aspects of the program such as number

[1] A basic block is a sequence of instructions with a single entry point (the first instruction) and a single exit point (the last instruction).

[2] An edge connects a pair of basic blocks if one is the predecessor of the other. For example, if the last instruction of basic block BB0 can be followed immediately by the first instruction of basic block BB1, then BB1 is a successor of BB0 (and BB0 is a predecessor of BB1).

of basic blocks with a single successor (feature 2), number of conditional branches (feature 20) and number of calls with pointers as arguments (feature 42). A complete list of the 56 features is given in [Fursin et al. 2011].

The following code shows how feature 2 is calculated using two auxiliary predicates `edge_src_pr2/2` (which counts the number of edges N whose source is at basic block B) and `edge_src_pr2_sel1/1` (which determines whether B has exactly one successor):

```
edge_src_pr2(B,N)     :- bb_p(B), findall(E,edge_src(E,B),L),
                         count_lst(L,N).
edge_src_pr2_sel1(B) :- edge_src_pr2(B,N), N=1.
ft(ft2,N)             :- findall(B,edge_src_pr2_sel1(B),L),
                         count_lst(L,N).
```

2.3 Inductive Logic Programming

The aim of ILP [Muggleton and De Raedt 1994] is to learn hypotheses H which generalise relations between background knowledge B about a problem and positive examples E^+ of when a given relation holds and negative examples E^- of when it does not hold. More formally, the goal is to learn hypotheses H which, given B, cover all of the positive examples (coverage) and none of the negative examples (consistency), which can be written $B \cup H \models E^+$ and $B \cup H \not\models E^-$.

For example, in order to learn the relation "*Flag F is a bad flag for program P*", the background knowledge B might contain clauses describing structural aspects of each program and E^+ and E^- could include examples of programs for which flag F has a bad or good influence on performance respectively. The following is an example of how such a problem could be encoded in Prolog:

```
B   =  program(prog1). program(prog2). program(prog3).
       flag(flag1). ft(ft1). avg(ft1,0.6).
       ft(ft1,prog1,0.9). ft(ft1,prog2,0.7). ft(ft1,prog3,0.2).
       small_ft(P,Ft) :- ft(Ft,P,N), avg(Ft,Avg), N<Avg.
       large_ft(P,Ft) :- ft(Ft,P,N), avg(Ft,Avg), N>=Avg.
E+  =  badFlag(prog1,flag1). badFlag(prog2,flag1)
E-  =  :- badFlag(prog3,flag1).
```

In addition to this background theory, mode declarations are required to constrain the search space used for generalisation [Muggleton 1995]. There are two types of mode declaration: mode head (`modeh/2`) and mode body (`modeb/2`) which define the format of literals that may appear respectively in the head and body of any learned hypothesis. The first term of each mode declaration specifies the recall, which is the number of alternative instantiations permitted for the predicate described in term two. The second term defines the type of terms that may appear in the predicate and whether they are an input variable (`+`), output variable (`-`) or constant (`#`). In the above example, we might use the following mode declarations to state that

learned rules may consist of `badFlag/2` in the head and `ft/3`, `small_ft/2` and `large_ft/2` in the body:

```
:- modeh(*,badFlag(+program,#flag))?
:- modeb(*,ft(#ft,+program,#any))?
:- modeb(*,small_ft(+program,#ft))?
:- modeb(*,large_ft(+program,#ft))?
```

The asterisks (`*`) in the above declarations specify an arbitrary recall (which is set to 100 in Progol by default). The `modeh` declaration states that the head of a learned rule may contain `badFlag/2` with an input variable of type `program` as its first term and a constant of type `flag` as its second term. The first `modeb` states that `ft/3` with a constant of type `ft`, an input variable of type `program` and a constant of any type may appear in the body of hypotheses.

The generalisation process in Progol proceeds by selecting one positive example at a time and first constructing the *most specific clause* that explains the example. This is constructed by placing the positive example at the head and adding to the body, all true literals for which the `modeb` declarations are provable. In this case the most specific clause is as follows:

```
badFlag(P,flag1) :- ft(ft1,P,0.900), large_ft(P,ft1).
```

Now Progol constructs several hypotheses and scores them based on the number of positive and negative examples they cover. The aim is to cover as many of the positive examples as possible and none of the negatives. The highest scoring hypothesis is chosen, any positive examples it covers are retracted from the background theory and generalisation continues with the next remaining positive example.

In this case, Progol will return the following hypothesis which states that `flag1` is a bad flag for program P if feature 1 of P is large:

$$H \quad = \quad \texttt{badFlag(P,flag1) :- large\_ft(P,ft1).}$$

In the next section we augment the background knowledge to capture additional relations between features and to generalise the information contained within Prolog-encoded IR in order to allow more complex rules to be learnt when applied to real data (as shown in Sec. 4.2).

3 Method

Our aim is to exploit logic programming to develop a better approach for selecting effective compiler flags on embedded software. First we generate our own data based on the more recent Bristol/Embecosm Embedded Benchmark Suite (BEEBS) (Sec. 3.1) then we propose two new ILP approaches (ILP+FV and ILP+IR). In ILP+FV we replicate the Milepost approach within an ILP setting using rule learning instead of 1NN (Sec. 3.2). Next, in ILP+IR we introduce IR which provides a lossless representation of the code and contains potentially useful learning information that is not present in the feature vector but which ILP can target directly (Sec. 3.3).

3.1 Identifying Examples of Significant Flags

This study targets the STM32VLDiscovery development board which features a Cortex-M3 embedded processor. We produced training data based on 60 programs from the Bristol/Embecosm Embedded Benchmark Suite (BEEBS) [BEEBS 2015] which was developed in response to the lack of free open source benchmarks compatible with embedded systems [Pallister et al. 2013a]. We fixed the input for each program and we studied the effect of compiler flags on the whole program.

Iterative compilation was used to study the effects of 133 flags that are available when compiling for the Cortex-M3 with GCC 4.8. We generated 1000 random configurations by selecting `O1`, `O2` or `O3` with probability $p(\frac{1}{3})$ and then explicitly enabling or disabling each of the 133 flags with $p(\frac{1}{2})$. Each benchmark program was compiled with each configuration and the resulting execution times were recorded.

The most significant flags for each program were identified by analysing good configurations (which we defined as those with an execution time within 5% of the best configuration). A flag was identified as good for performance if it appeared in at least 75% of good configurations or bad if it appeared in no more than 25% of good configurations.[3] These parameters were determined by preliminary testing.

Good and bad flags were turned into negative and positive examples of predicate `badFlag/2` (we found better results by choosing which flags to disable with respect to `O3`, rather than selecting which ones should be enabled). For example, if flag x were good and flag y were bad for program a this would be represented as:

```
:- badFlag(a,x).            % good flag
badFlag(a,y).               % bad flag
```

Note that some flags were neither good nor bad for a given program and our thresholds are used to prevent these from adding noise to the training set.

3.2 Extracting Program Features for ILP+FV

In order to make a fair comparison with Milepost and focus on testing the effect of rule learning and IR, our ILP+FV method used Milepost GCC to extract the feature vector for the most time consuming function of each benchmark, which we found by profiling each program using the `gprof` tool on an x86 processor.[4] Profiling in this way enables learning to be concentrated on the most characteristic function of the program and filters out sections of the code which may have little effect on performance. Each feature vector was normalised by number of instructions (feature 24) and converted into a series of Prolog facts to enable interpretation by Progol.

Further predicates were created to allow Progol to summarise and compare between the features of each program using quartiles and averages. For example, `qt(P,Ft,Q)` gives the quartile Q (1, 2, 3 or 4) that feature Ft is in for program P.

[3] Sixteen programs with fewer than five good configurations were excluded from the training set as there were not enough data to approximate the effects of individual flags.

[4] As it is very difficult to obtain function-level execution time for the Cortex-M3, we profiled an x86 to estimate the most time consuming function. Each function should be called the same number of times on each architecture, but we acknowledge that relative function costs may vary.

3.3 Extracting Intermediate Representation for ILP+IR

Since it is not known whether the feature vector contains the best features for the learning task, our ILP+IR method seeks to automatically discover relevant features during learning rather than predefine them. In Milepost GCC [Fursin et al. 2011], the feature vector is generated by applying Prolog rules to the Milepost IR. We wish to learn directly from the IR because it provides a lossless representation of the code which may contain potentially relevant structural information that is not present in the feature vector. In general, any IR could be used for our background knowledge (for example, LLVM IR), but we favour the Milepost IR since it encodes precisely the internal data structures over which the GCC optimisations are applied.

To allow comparisons to be consistent with ILP+FV and Milepost, we extracted the Milepost IR for the most time consuming function of each benchmark and used it as the basis for our background knowledge. The original Milepost IR used a separate file for each function and the resulting Prolog clauses did not define the program or function to which they referred. We added program and function names to each predicate in order to guarantee uniqueness and enable the IR to be used in our ILP methodology. Suppose `edge_src(ed0,bb1)` describes an edge in function `f` of program `p`, then this would become `edge_src(p,f,ed0,bb1)`.

Some of the predicates within the Milepost IR were too specific to allow meaningful generalisations, therefore we added more general rules to group such cases. For example, `assign_class(P,F,st1,minus_expr)` and `assign_class(P,F,st2, plus_expr)` denote two statements `st1` and `st2` which contain a subtraction and addition respectively. Since addition and subtraction are handled the same in hardware, it was sensible to group the two classes by adding the following rules:

```
assign_addsub(P,F,S) :- assign_subcode(P,F,S,plus_expr).
assign_addsub(P,F,S) :- assign_subcode(P,F,S,minus_expr).
```

The IR also contains an `expr_code(P,F,E,C)` predicate which gives the class `C` of expression `E` (for example integer constant, variable declaration or array reference). We added the following rule to identify classes appearing more than once in a function `F`.

```
expr_code2(P,F,C) :- expr_code(P,F,E1,C), expr_code(P,F,E2,C),
                     E1 @< E2.
```

Other predicates contained numerical data such as probability `Pr` of an edge being taken, denoted by `edge_prob(P,F,E,Pr)`. We added rules to allow general comparisons between edge probabilities as follows:

```
edge_prob_low(P,F,E)  :- edge_prob(P,F,E,N), N < 0.5.
edge_prob_high(P,F,E) :- edge_prob(P,F,E,N), N >= 0.5.
```

3.4 Learning Rules for Bad Flags

In both ILP+FV and ILP+IR, Progol learns rules for `badFlag/2` which are used to predict which of the 133 flags should be switched off for good performance on

a given program. We construct the predicted configuration by enabling `O3` then explicitly disabling the flags identified as bad and enabling the remaining flags.

In ILP+FV, Progol learns rules for `badFlag/2` using the following mode declarations (Sec. 2.3) to restrict the search space:

```
:- modeh(*,badFlag(+program,#flag))?
:- modeb(*,large_ft(+program,#ft))?
:- modeb(*,small_ft(+program,#ft))?
:- modeb(*,non_zero(+program,#ft))?
:- modeb(*,ft(#ft,+program,0))?
:- modeb(*,qt(+program,#ft,#any))?
```

These allow learned rules to use the knowledge that a feature for a program is larger or smaller than average, non-zero, zero or within a particular quartile. To avoid over-fitting we did not allow rules to contain exact feature values (other than zero).

In ILP+IR, Progol learns from the Milepost IR, rather than the feature vector, using modeb declarations such as the following:[5]

```
:- modeb(*,edge_src(+program,-func,-edge,-bb))?
:- modeb(*,edge_dest(+program,-func,-edge,-bb))?
:- modeb(*,edge_prob_low(+program,-func,-edge))?
:- modeb(*,edge_prob_high(+program,-func,-edge))?
:- modeb(*,expr_code2(+program,-func,#any))?
:- modeb(*,assign_addsub(+program,-func,-stmt))?
```

These declarations allow rules to contain knowledge of edges between source and destination basic blocks, edges with a high or low probability of being taken, classes of expressions that appear more than once and assignments featuring addition or subtraction. The last four declarations feature some of our new predicates which group elements of the IR to aid the learning of general rules (Sec. 3.3).

3.5 Evaluating Predictions

In order to compare with the Milepost state of the art, we implemented the 1NN algorithms used by Milepost and applied them to our data set.[6] Leave-one-out cross-validation was used to evaluate how well each method predicted for unseen programs. For each program T_i in training set $T_1, \ldots, T_n$, program T_i was excluded from training and predicted for as if it were previously unseen.

The performance of `O3` was used as the baseline for comparing configurations as it is the best optimisation level available without significant additional effort. The execution time for configuration x relative to `O3` was calculated as follows:[7]

$$r = \frac{\textit{execution time of configuration } x}{\textit{execution time of } \texttt{O3}} \tag{1}$$

[5] The full set of mode declarations is available at `github.com/craigblackmore/logiflag`

[6] We did not use the Milepost software directly as it had not been trained for the Cortex-M3 and there were difficulties in supplying our data set to the system.

[7] If $r < 1$ then x ran faster than `O3`. Conversely, if $r > 1$ then x took more time than `O3`.

4 Results and Evaluation

We present our results in three parts: first we reassess the 1NN methods from Milepost which we tested on a larger, more diverse training set than the original evaluation in [Fursin et al. 2011] (Sec. 4.1). Secondly, we compare our ILP+FV and ILP+IR methods to the Milepost state-of-the-art (Sec. 4.2). Finally we show how a hybrid approach which builds on the individual strengths of ILP, 1NN and `O3` further improves predictive performance (Sec. 4.3).

4.1 Re-evaluation of Milepost Results

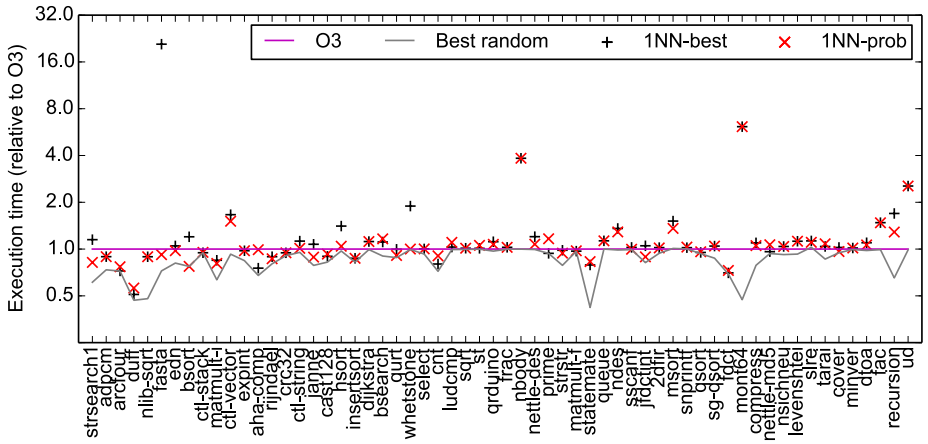

Fig. 1. 1NN-best, 1NN-prob and best of 1000 random configurations

We used leave-one-out cross validation to test the 1NN-best and 1NN-prob approaches of Milepost [Fursin et al. 2011] on our data set which contains nearly three times as many benchmarks as the original study. The 1NN-prob method outperformed 1NN-best in 37 out of 60 benchmarks (Fig. 1), but both methods actually increased the average execution time relative to `O3` by 54% and 17% respectively. Milepost [Fursin et al. 2011] also found that 1NN-prob outperformed 1NN-best on their data set, but in contrast they achieved an average speed-up of 10% using 1NN-prob on an embedded platform. In the rest of the paper we will compare our methods to the best result achieved by 1NN-best and 1NN-prob, which we will refer to as 1NN-both. The average for 1NN-both is a 15% increase in execution time.

We believe that 1NN performed worse in our study because our training programs are more diverse than those used in the Milepost study. We analysed the diversity by comparing the Euclidean distance between the normalised feature vector of each program. Figure 2 shows that the set of cBench programs used by Milepost contains clusters of very similar programs (the lighter the dot, the closer the two programs). In fact, the `blowfish` encryption and decryption programs have identical source code but are included as two different benchmarks `blowfish_e` and `blowfish_d`

(as is the case for `rijndael`). If we reduce the initial set of 22 benchmarks to allow only one variation of each program, then this gives a total of 13 programs. In contrast, the BEEBS programs used in our experiments were designed specifically to cover a wide range of features and consequently the programs are much more distant (Fig. 3). Some of the BEEBS programs were in fact derived from cBench including `blowfish` and `rijndael`, however, these only appear once in the set and

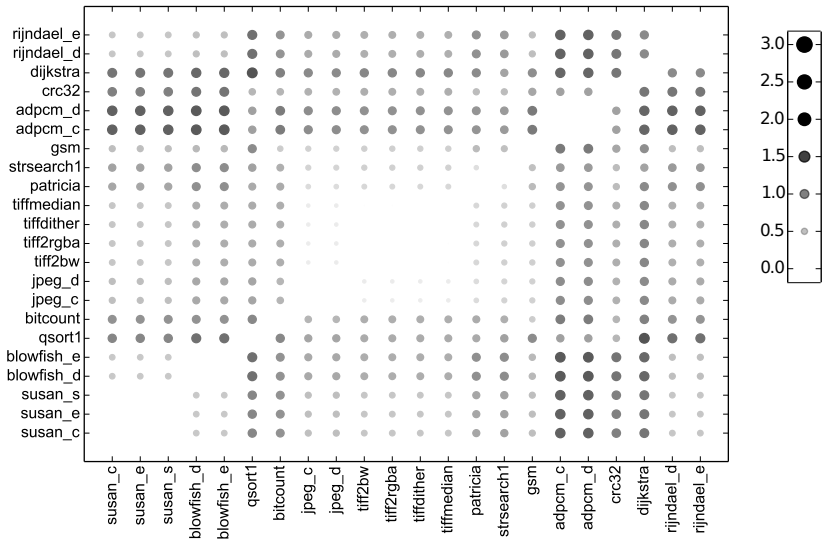

Fig. 2. Euclidean distance between normalised feature vector of each cBench program used in the Milepost study. Note: a similar graph appears in Fig. 9 of [Fursin et al. 2011].

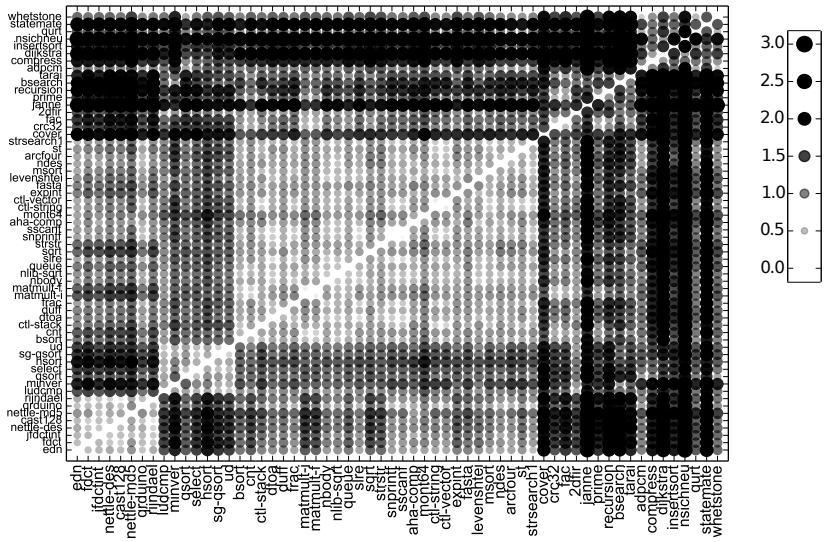

Fig. 3. Euclidean distance between normalised feature vector of each BEEBS program

overall BEEBS contains only three pairs of very similar programs (`ctl-stack`, `ctl-string`), (`matmult-int`, `matmult-float`) and (`trio-sscanf`, `trio-sprintf`).

One way to improve our results would be to add extra training programs in order to cover a wider range of optimisation scenarios, however, this may require a huge number of programs which would increase training times dramatically. Alternatively, we could search for patterns within the program structure that may help to predict effective configurations, as in our ILP methodology.

4.2 ILP+FV and ILP+IR

Our ILP+FV method outperformed both 1NN approaches used by Milepost in 25 out of 60 benchmarks. We further improved the ILP method by training on the more expressive IR rather than the feature vector. The ILP+IR method outperformed ILP+FV and 1NN-both in 34 out 60 benchmarks each. Furthermore, ILP+FV and 1NN-both increased the average execution time of `O3` by 11% and 15% respectively while ILP+IR was in line with `O3`.

Figure 4 shows that each predictive method performed well on different programs and that no one of these approaches was best across all benchmarks. Therefore a better strategy might be to apply a number of predictive approaches to the target program and select the configuration which gives the best result. We test the gains of such a hybrid approach in Sec. 4.3. We believe that 1NN performs well when the training set contains a program that is similar enough to the test program. In cases where there is no such similar program, then ILP can still pick out important characteristics that are common between programs in order to make good predictions.

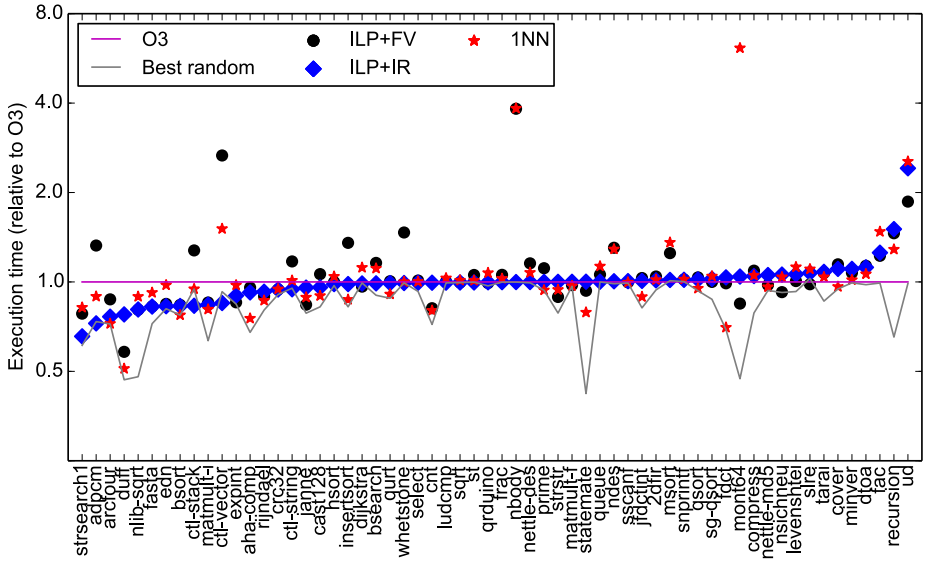

Fig. 4. ILP+FV, ILP+IR, 1NN-both and best of 1000 random configurations

The ILP+IR method was the only predictive approach to outperform random iterative compilation on any of the benchmarks. The execution times achieved by ILP+IR for `ctl-stack` and `ctl-vector` were over 5% better than the best result found after testing 1000 random configurations (which takes over 2.5 hours per benchmark compared to ILP+IR which takes under 10 seconds to predict and evaluate its selected configuration once trained).

One of the advantages of ILP is that it produces human-readable rules which show why certain flags were predicted. The rules can be easily translated into English and the IR should be familiar to GCC developers. For example, the following rule:

```
badFlag(P,'-fguess-branch-probability') :-
    stmt_code(P,F,S,gimple_cond),expr_code2(P,F,real_type).
```

can be translated as *"the -fguess-branch-probability flag should be disabled if function F of program P contains a conditional statement S and at least two floating point expressions"*. The guess branch probability optimisation targets conditional statements, therefore it is plausible that any program affected by it should contain a conditional statement. Secondly, the Cortex-M3 hardware does not have a floating point unit [ARM 2006], instead it relies on the compiler to emulate floating point functionality using integer operations, which is slower and therefore floating point operations are more expensive than integer operations and mispredictions would have more of an impact.

Interestingly, the above rule uses the `expr_code2(P,F,C)` predicate which we introduced in Sec. 3.3 to capture expression classes that appear more than once in a given function. Several of the other rules also use this predicate including `badFlag(P,'-fschedule-insns2') :- expr_code2(P,F,array_ref)` which states that *"the -fschedule-insns2 flag should be disabled if function F of program P contains at least two array references"*. Here ILP has found a potentially useful feature in the IR which is not present in the flattened feature vector. In fact, none of the attributes in the feature vector refer to array references.

Our rules can also identify potential gaps in the training set. For example, further analysis is required to determine if the rule `badFlag(P,'-fschedule-insns') :- expr_int_size(P,F,E,16)` (or *"-fschedule-insns should be disabled if program P of function F contains a 16-bit integer"*) is feasible. This could be achieved by writing a program for which this flag improves performance in order to add a counter-example to the training set.

Our ILP method also identified 19 flags which should always be disabled for the target platform, which were learned as facts such as `badFlag(P,'-fsched-spec')` and `badFlag(P,'-funroll-loops')`.

4.3 Hybrid Approach

Given the observation that ILP, 1NN and `O3` performed better on different benchmarks, we evaluated a hybrid method which tests the configurations given by ILP, 1NN and `O3` and selects the best result. This hybrid approach reduced the average execution time by 8% compared to `O3`.

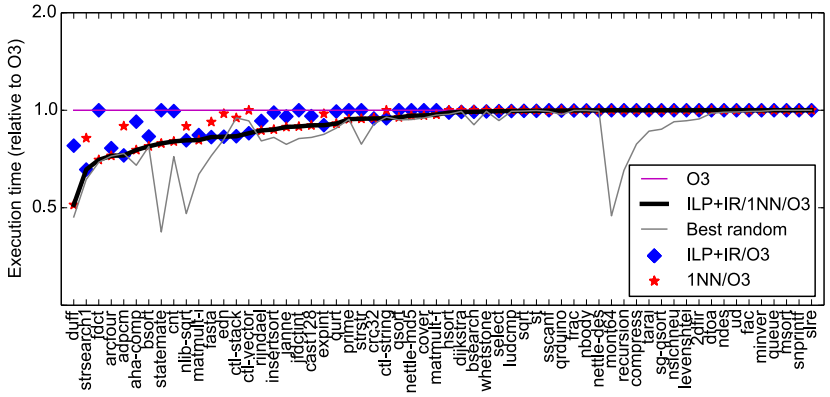

Fig. 5. ILP+IR/1NN-both/O3 hybrid and best of 1000 random configurations

Over half of the programs gained a positive benefit from the hybrid approach (Fig. 5), but there remains a small set of programs for which none of the methods were able to reduce their execution times even though the random approach shows a speed-up is possible. This may be due to lack of knowledge about certain flags in the data set. For example, further analysis of the `fdct` program showed that most of its speed-up can be obtained by disabling the `-ftree-fre` flag, but there exists no other program in the training set for which this was identified as a bad flag. Therefore, when `fdct` is removed during leave-one-out cross validation, none of the remaining examples allow a rule for this flag to be learned. In future we plan to add new training programs to increase the number of examples for such flags.

5 Conclusion and Future Work

We introduced and evaluated a novel ILP-based method for predicting effective compiler flags which outperformed the state of the art. We showed that learning from declarative IR rather than a predefined feature vector significantly improved the performance of our method. We have also demonstrated the human-readable rules that our method can produce. Our future work will use weighted examples to express the significance of each flag. In addition, we seek to further exploit the expressivity of ILP to learn dependencies between flags and also add rules that categorise flags by the type of optimisation they apply. This study assumed a fixed ordering of optimisations (which is decided internally by GCC's pass scheduler). In future, we will consider the order in which optimisations are applied (which is also important [Kulkarni and Cavazos 2012]) using the LLVM compiler infrastructure [LLVM 2015] which allows user-specified orderings. In principle, our approaches are applicable to any metric that can be measured and which the compiler can influence. Furthermore, the methodology could be applied to any compiler that offers control over which optimisations are enabled or disabled. In future work, we will target the multi-objective goal of reducing execution time and energy consumption, which is crucial on embedded systems.

References

ARM. 2006. Cortex-M3 technical reference manual (Revision: r1p1).

BEEBS 2015. `http://beebs.eu/`. [Accessed 02/07/2015].

Collective Benchmark 2012. `http://ctuning.org/cbench/` [Accessed 05/03/15].

Fursin, G., Kashnikov, Y., Memon, A. W., Chamski, Z., Temam, O., Namolaru, M., et al. 2011. Milepost GCC: Machine learning enabled self-tuning compiler. *International Journal of Parallel Programming 39,* 3, 296–327.

Fursin, G., Miranda, C., Temam, O., Namolaru, M., Yom-Tov, E., Zaks, A., et al. 2008. Milepost GCC: machine learning based research compiler. In *GCC Summit.*

GCC, the GNU Compiler Collection 2015. `http://gcc.gnu.org/`. [Accessed 02/04/2015].

Kulkarni, S. and Cavazos, J. 2012. Mitigating the compiler optimization phase-ordering problem using machine learning. *ACM SIGPLAN Notices 47,* 10 (Oct.), 147–162.

LLVM 2015. `http://llvm.org/`. [Accessed 02/07/2015].

Muggleton, S. 1995. Inverse entailment and progol. *New Generation Computing 13,* 3-4, 245–286.

Muggleton, S. and De Raedt, L. 1994. Inductive logic programming: Theory and methods. *The Journal of Logic Programming 19*, 629–679.

Pallister, J., Hollis, S. J., and Bennett, J. 2013a. BEEBS: Open benchmarks for energy measurements on embedded platforms. *arXiv:1308.5174v2 [cs.PF].*

Pallister, J., Hollis, S. J., and Bennett, J. 2013b. Identifying compiler options to minimize energy consumption for embedded platforms. *The Computer Journal.*