# Automatic Feature Generation for Machine Learning–Based Optimising Compilation

HUGH LEATHER, University of Edinburgh
EDWIN BONILLA, NICTA and Australian National University
MICHAEL O'BOYLE, University of Edinburgh

Recent work has shown that machine learning can automate and in some cases outperform handcrafted compiler optimisations. Central to such an approach is that machine learning techniques typically rely upon summaries or features of the program. The quality of these features is critical to the accuracy of the resulting machine learned algorithm; no machine learning method will work well with poorly chosen features. However, due to the size and complexity of programs, theoretically there are an infinite number of potential features to choose from. The compiler writer now has to expend effort in choosing the best features from this space. This article develops a novel mechanism to automatically find those features that most improve the quality of the machine learned heuristic. The feature space is described by a grammar and is then searched with genetic programming and predictive modelling. We apply this technique to loop unrolling in GCC 4.3.1 and evaluate our approach on a Pentium 6. On a benchmark suite of 57 programs, GCCs hard-coded heuristic achieves only 3% of the maximum performance available, whereas a state-of-the-art machine learning approach with hand-coded features obtains 59%. Our feature generation technique is able to achieve 76% of the maximum available speedup, outperforming existing approaches.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Compilers

General Terms: Machine Learning, Performance

Additional Key Words and Phrases: Feature generation, genetic programming, program optimisation

**ACM Reference Format:**
Hugh Leather, Edwin Bonilla, and Michael O'Boyle. 2014. Automatic feature generation for machine learning–based optimising compilation. ACM Trans. Architec. Code Optim. 11, 1, Article 14 (February 2014), 32 pages.
DOI: http://dx.doi.org/10.1145/2536688



## 1. INTRODUCTION

The use of machine learning to automate compiler optimisation [Cavazos and O'Boyle 2006; Stephenson and Amarasinghe 2005; Beaty 1991] has received considerable research interest. Previously, compiler writers have had to manually tune their heuristics and often were faced with difficulties due to the complexity of the optimisations and interactions with the architecture and the rest of the compiler [Cooper et al. 2005]. The vast number of variables to consider makes tuning heuristics a daunting task. All of this work may need to be repeated whenever the architecture changes. Given the rapidly evolving nature of architecture, the time and effort required to achieve an

This work is supported under the European project EU FP6 STREP MILEPOST IST-035307.
Authors' addresses: Hugh Leather and Michael O'Boyle, University of Edinburgh, Informatics Forum, 10 Crichton St, Edinburgh, Midlothian, EH8 9AB; Edwin Bonilla, NICTA, Canberra Research Laboratory, Tower A, 7 London Circuit, Canberra City ACT 2601, Australia.

DOI: http://dx.doi.org/10.1145/2536688

acceptable optimising compiler is becoming a serious issue. Machine learned heuristics are attractive in that they not only automatically adapt to a new environment but, in practice, often outperform their human created counterparts [Stephenson et al. 2003].

The difficulty for compiler-based machine learning, however, is that it requires programs to be represented as a set of features that serve as inputs to a machine learning tool [McGovern and Moss 1999]. It is now the compiler writer's new task to extract the crucial elements of the program as a fixed-length feature vector. Given that features will be computed over the graphs and trees that form the compiler's internal representations of the programs, and that these structures are unbounded, there are an infinite number of these potential features, making the choice of the right features a nontrivial task. In some ways, we have pushed the problem from one of hand coding the right heuristic to one of hand coding the right features.

The interaction between features and a machine learning algorithm is complex. Features that are based on human intuition may not be the best features to choose. Features may not represent all of the relationships between the program and the desired outcome, or, even if they do, they may not work sufficiently well with the machine learning algorithm. In fact, the true quality of the features can only be found by directly asking the machine learning algorithm to make predictions using the features and seeing how good the predictions are [Kohavi and John 1997].

Previously, researchers in machine learning for compilers manually created lists of features that they believed to be reasonable [Cavazos and Moss 2004]. Many such works use feature selection to remove redundant or unhelpful features. However, no attempt has been made to search through the feature space, generating entirely new features along the way. In our approach,[1] we represent the space of features as a grammar. Each sentence from the grammar represents one feature. We search this space for good features that most improve the machine learning algorithm's performance. Our system is allowed to range over an infinite space of features with a smart Genetic Programming (GP) methodology. Although GP has been used before to search over the model space [Stephenson and Amarasinghe 2005], this is the first time that it has been used to generate features.

We evaluated our technique on an extensively studied problem: loop unrolling [Monsifrot et al. 2002; Aho et al. 1986]. Loop unrolling is an optimisation performed by practically every modern compiler, and we study its effect in a widely used open-source compiler: GCC. Furthermore, machine learning has been successfully applied to loop unrolling [Stephenson and Amarasinghe 2005], allowing direct comparison. Given the mature nature of this problem, it should be a challenging task for a new technique to show additional improvement.

However, across 57 benchmarks, we show that GCC's unrolling heuristic, on average, obtains just 3% of the maximum performance available. A state-of-the-art [Stephenson and Amarasinghe 2005] machine learning approach is able to increase this, automatically, to 59%. Using our feature search scheme, we are able to generate features and learn a model that achieves 76% of the maximum speedup available, outperforming prior approaches.

The remainder of this article is organised as follows. Section 2 shows how machine learning is presently applied for learning compiler heuristics and describes the problems with it. This is followed in Section 3 by an overview of our system. Section 4 explains how feature grammars are used, while Section 5 discusses the design challenges

[1]This article is an extended version of our previous work [Leather et al. 2009], providing additional details about the feature generation process that the prior work was unable to elaborate on due to space considerations. In particular, a full discussion of how feature grammars are created, and the difficulties that must be overcome, are missing from that earlier work.

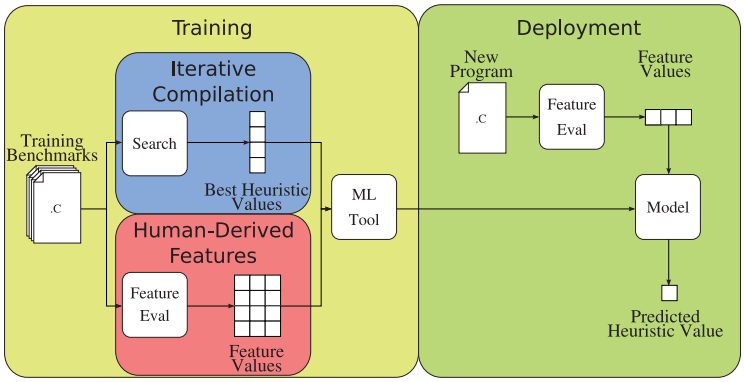



Fig. 1. Typical structure of machine learning in compilers. A number of example benchmarks are used in the training phase. For these, the best parameters are found for a given optimisation by iterative compilation. Additionally, expert written features are computed for each program. These data are used by a machine learning tool to create a predictive model. The model is used in deployment so that when given the features of a new program, it will predict the best heuristic value for it.

involved in creating a feature grammar. Section 6 explains the support required for searching the feature space. Section 7 outlines the particular autogenerated grammar used for loop unrolling in GCC. Our experimental setup, methodology, and results are presented in Sections 8, 9, and 10, respectively. This is followed by a brief summary of related work and some concluding remarks.

## 2. MANUAL FEATURE CREATION

The compiler writer must be aware of several pitfalls when writing features by hand, which are covered in this section.

*Supervised machine learning* involves learning a function that generalises data presented as a set of training examples. Given a set of pairs, $\langle \mathbf{x}_i, \mathbf{y}_i \rangle$, each of which is an example of some function $\mathbf{y} = f(\mathbf{x}, \mathbf{z})$, a supervised machine learning tool will generate a function $\mathbf{y} = f'(\mathbf{x})$ that is a best guess at replicating the original function $f$. The new function $f'$ is called a *predictive model*, or just *model*, since it tries to predict the output of the original function for a new input vector that was not present in the training examples. The input vector, $\mathbf{x}$, is called a *feature* vector, and the output vector, $\mathbf{y}$, is the *response*. It may be that not all pertinent information is measured; thus, the vector $\mathbf{z}$ is hidden from the machine learner. In the typical case of a compiler experiment, there will be one example, $\langle \mathbf{x}_i, \mathbf{y}_i \rangle$, for each training benchmark. In this discussion and throughout the article, a benchmark refers to the code regions targeted by the transformation rather than necessarily a whole program. For example, in loop unrolling, there will be one example for each loop in the training benchmarks. Each $\mathbf{y_i}$ will be the desired value of some optimisation parameter, discovered by searching for whichever value most improves the performance of the code. The features, $\mathbf{x_i}$, will be summaries of the program, derived from analyses of the internal representation (AST, CFG, DDG, etc.) of the program.

The typical structure of machine learning in compilers is shown in Figure 1. There are two phases: a training phase, performed off-line, in advance, in which example data are used to create a predictive model, and a deployment phase, in which that model is used to predict the proper parameters for optimisation heuristics. In the standard form of the training phase, a number of benchmarks are compiled in different ways and profiled to determine which is the best heuristic value for each—this search is called *iterative compilation*. The other data that must be collected during training are the

features of each program. These are derived from the compiler writer's own knowledge of the optimisation to tune and his hunches. Together, the features and matching target heuristic parameters are passed to a machine learning tool that builds the model. In deployment, the compiler writer's features are used once more to summarise the new program. From that, the model can predict, based on the examples given to it during training, what the right heuristic value should be for the new program.

The process demonstrated in this article permits not only searching for the best performance through iterative compilation but also searching for the best features.

### 2.1. Difficulties with Human-Created Features

*Irrelevant features.* It is easy to conceive of features that will have no relevance to any optimisation task. Features such as "the number of comments in the code" or "the average length of identifiers" would not help a machine learning algorithm. These examples are so obvious that no human would suggest them, but the same situation arises in more subtle cases where a sensible sounding feature simply has no bearing on the current optimisation.

More serious is the case when a feature is useful on its own but when added to an existing set of features does not show any additional improvement. This can happen when all of the useful information in a feature is already present in the other features—for example, if the feature is a linear combination of the others and a linear regression is performed.

The compiler writer can somewhat mitigate the damage of irrelevant features by applying feature selection to remove unnecessary features. However, some features may accidentally correlate with the training data, but not with test data; the selection algorithms will fail to remove these, which can mislead the machine learning tool.

*Classification clashes.* Two distinct programs may have the same feature vector but different best values for the heuristic; a machine learning algorithm will predict at least one of them wrongly. This does in fact happen in practice, as shown by Monsifrot et al. [2002]. The presence of these clashes is a clear indication that the features are inadequate, since they cannot distinguish examples from different classes. However, irrelevant features and noise can obscure classification clashes.

When clashes are found, they place an upper bound on the accuracy of the models created by the machine learning tool. It is possible that adding other features may help to separate the features by adding other dimensions.

*Classifier peculiarities.* A set of features that performs well for one machine learning algorithm might not be good for another [Kohavi and John 1997]. In other words, features are not independent of the learning technology.

*Beyond simple features.* Once the obvious features have been written, inevitably they do not completely represent the relationship between the programs and the desired heuristic values. The expert must then choose which additional features to implement. The sheer number of choices can be huge, and the expert will find that at each stage, as they increase the complexity of their features, they face the same difficult challenges as they did before, only now multiplied due the larger set of features with which they must work.

### 2.2. Motivating Example

To demonstrate that selecting the right features can have significant impact on optimisation performance, we look at an example of unrolling a loop. Loop unrolling can reduce the branch overhead of loop iterations and, by creating larger contiguous code

```
for (i=0;i<EXP_TABLE_SIZE-1;i++) {
    l->SpotExpTable[i][1] =
        l->SpotExpTable[i+1][0] -
        l->SpotExpTable[i][0];
}
```

(a)

| Method | Unroll | Cycles | Speedup | % of Max |
|---|---|---|---|---|
| Baseline | 0 | 406,424 | 1.0000 | 0% |
| Oracle | 11 | 328,352 | 1.2378 | 100% |
| GCC Default | 7 | 418,464 | 0.9712 | -12% |
| GCC Tree | 2 | 392,655 | 1.0351 | 14% |
| **Our Technique** | **11** | **328,352** | **1.2378** | **100%** |

(b)

Fig. 2. Loop from MediaBench (a) and speedups using various schemes (b). GCC's default heuristic selects an unroll factor of 7, causing a slowdown. Using GCC features and machine learning, an unroll factor of 2 is selected, giving a small improvement. Our technique correctly determines 11 as the best unroll factor.

| Feature Name | Value |
|---|---|
| ninsns | 10 |
| av_ninsns | 9 |
| niter | 6.14E17 |
| expected_loop_iterations | 49 |
| num_loop_branches | 1 |
| simple_p | 1 |

(a)

```
if( ninsns <= 63 )
  if( simple_p > 0 )
    if( num_loop_branches <= 3 )
      if( av_ninsns > 5 )
        if( niter > 6.1384926724882432E17 )
          if( expected_loop_iterations > 8 )
            if( niter <= 6.1428835034542899E17 )
              if( num_loop_branches <= 1 )
                unrollFactor = 2;
```

(b)

Fig. 3. The path through the learned GCC tree heuristic (b) for the example in Figure 2 and the features used in that path (a). The features are the variables that GCC's original heuristic examined when deciding its value. `ninsns` is the number of instructions in the loop; `av_ninsns` is the average number of GCC expects to execute each time round the loop (based on GCC's nonprofiled guesses about the expected iteration count); `niter` is the number of iterations if statically known ahead of time, or a large number if not; `expected_loop_iterations` is GCC's guess about the actual number of loop iterations, which is not profiled and distinct from the provable iteration count; `num_loop_branches` is the number of branches in the loop; and `simple_p` is 1 if the loop is simple, zero otherwise (a simple loop has only a single exit). The decision tree using these features incorrectly chooses an unroll factor of 2, leading to only 14% of the maximum available performance.

regions, grant later optimisations greater scope to improve the code.[2] On the other hand, loop unrolling can have detrimental effects, such as increasing register pressure and cache misses. In addition, GCC provides several variants of unrolling, some of which cannot be used with particular unroll factors, complicating the mapping between unroll factor performance. Choosing the best unroll factor is therefore nontrivial.

Consider the loop in Figure 2, selected from the `mesa` benchmark within *MediaBench*. If GCC's default loop unroll heuristic (labeled GCC Default in Figure 2(b)) is applied, it determines that the best unroll factor is 7. When executed on the Pentium, this achieves a slowdown of 0.97. However, if all loop unroll factors up to 15 are evaluated exhaustively, then the best unroll factor is found to be 11, resulting in a speedup of 1.24, as shown by the Oracle entry in Figure 2(b). If GCC's heuristic is replaced with a machine learning decision tree algorithm, whose features use the same information used by GCC's heuristic (as shown in Figure 3(a)), then it is possible to achieve a speedup of 1.04 or 14% of the maximum available. Figure 3(b) shows the path followed by the learned decision tree heuristic leading to the unroll factor of 2 being selected.

If, instead, our technique is used to search for the best set of features and to train a decision tree over those, then the best unroll factor of 11 can be selected automatically, giving the maximum speedup for the loop in Figure 2. The path of the decision tree selecting the unroll factor of 11 is also shown in Figure 4(b), and the features touched

[2]The unroll pass considered here is `pass_rtl_unroll_and_peel_loops`. There are approximately 70 passes that run after this pass.

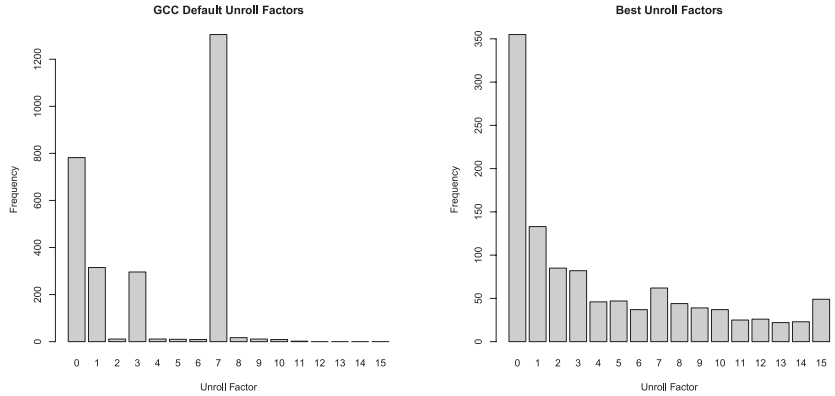
| Name | Value | Feature |
|---|---|---|
| f0 | 6.14 E17 | get-attr(@num-iter) |
| f1 | 308 | count...!is-type(wide-int) .. |
| f2 | 2 | count...is-type(basic-block)... |
| f3 | 5 | max... is-type(basic-block) . |
| f4 | 4 | count... is-type(array_type) |
| f5 | 0 | count... is-type(le) && ... |

(a)

```
if( f2 <=4 )
  if( f5 <= 0 )
    if( f0 > 8206 )
      if( f1 > 168 )
        if( f0  > 6.1E17 )
          if( f2 <= 3 )
            if( f1 <= 1247 )
              if( f4 > 1 )
                if( f3 > 4 )
                  if( f3 <= 6 )
                    unrollFactor = 11;
```

(b)

Fig. 4. Features for the learned heuristic (a) for the example in Figure 2 and the path used (b). The correct unroll factor, 11, is chosen, achieving 100% of the available performance. More complete features are given and described in Figures 31 and 32.




(a) (b)

Fig. 5. Histograms of unroll factors showing the frequency that GCC chooses different unroll factors (a) and the best unroll factors (b) found through iterative compilation of 2,778 loops. GCC has a very strong preference for unroll factors that lead to the number of loop bodies being a power of 2. The best unroll factors show only a slight preference for those factors.

are shown in Figure 4(a). This example shows that while performance of the heuristic can be improved by a machine learning approach, it may ultimately be limited by the features used. By searching the space of features, features better suited to the learning task at hand can be found.

Figure 5(a) and (b) shows the difference between the best unroll factor found through iterative compilation and the unroll factors chosen by GCC for the 2,778 used in the experimental setup of this article. The histograms show that GCC has a strong preference for "power of two" unrolling (unroll factor 7 indicates 7 copies of the loop body in addition to the original). The best possible unroll factors, however, are much more evenly spread, with only a small preference for "power of two" unrolling.

## 3. OVERVIEW

A high-level overview of the system is illustrated in Figure 6. The system is comprised of the following components: *training data generation*, *feature search*, and *machine learning*. The training data generation process extracts the compiler's Intermediate Representation (IR) of the program elements that the optimisation operated on (as described in Section 7) together with the optimal values for the heuristic that we wish

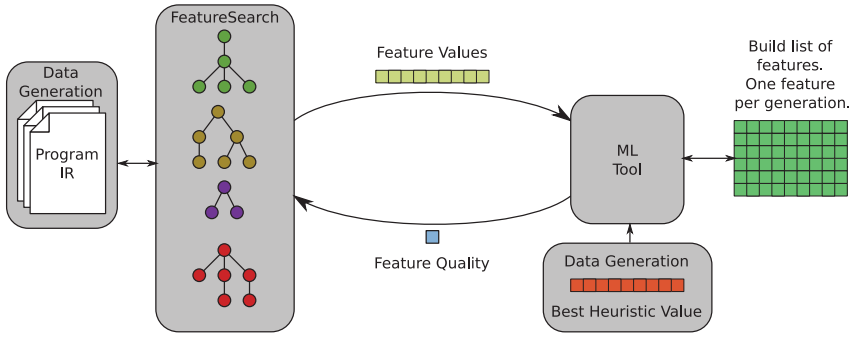



Fig. 6. Overview of the feature search system.

to learn. Once these data have been generated, the feature search component explores features over the compiler's IR and provides the corresponding feature values to the machine learning subsystem. The machine learning subsystem computes how good the feature is at predicting the best heuristic value in combination with the other features in the base feature set (which is initially empty). The search component finds the best such feature and, once it can no longer improve upon it, adds that feature to the base feature set and repeats. In this way, we build up a gradually improving set of features in a greedy fashion.

### 3.1. Data Generation

Similar to existing machine learning techniques, we must gather a number of examples of inputs to the heuristic and find out what the optimal answer should be for those examples. Each program is compiled in different ways, each with a different heuristic value. We time the execution of the compiled programs to find out which heuristic value is best for each program. Due to the intrinsic variability of execution times on the target architecture, we run each compiled program multiple times (100 runs) to reduce susceptibility to noise (see Section 8).

We also extract from the compiler the internal data structures that describe the programs. The feature search component will generate summaries of these data as candidate features. Typical data will be the abstract syntax tree, looping structures, use-def chains, and so forth. Whatever information can be extracted should be recorded, including whatever analyses are performed by any existing heuristics. This process is described in detail in Section 7.

These two aspects of data generation are depicted in Figure 6.

### 3.2. Feature Search

The feature search component, shown in Figure 6, maintains a population of feature expressions, represented as choice trees[3] (described in Section 6.1). The expressions come from a family described by a grammar derived automatically from the compiler's IR. Evaluating a feature on a program generates a single real number; the collection of those numbers over all programs forms a vector of feature values that are later used by the machine learning subsystem. The construction of grammars and their uses are discussed in Section 4.

The search component uses an evolutionary search over choice trees with the search operators described in Section 6.2. These allow genetic mutations and matings of the

[3]A choice tree represents a feature in a similar way to a parse tree, encoding the choices that selected the feature from the grammar.

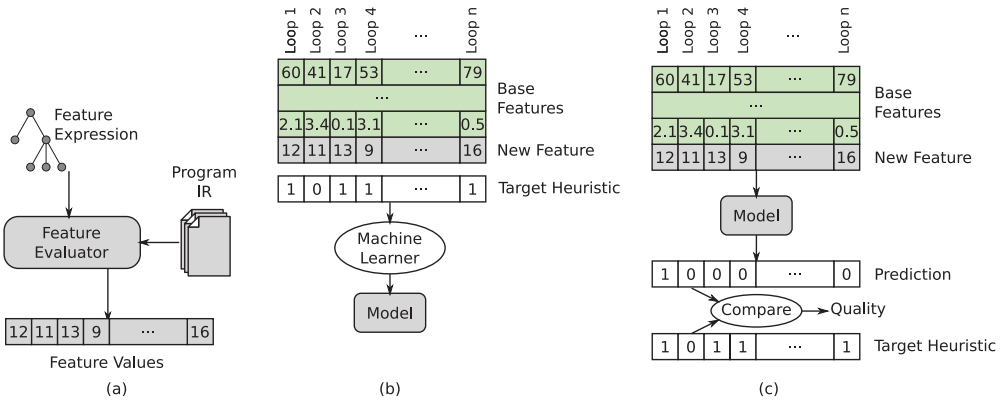



Fig. 7. Machine learning used to determine the quality of a feature.

choice trees. A population of choice trees is kept and sorted according to a fitness function. After the trees are ranked, a new population of the same size is created. Each member of the new population is created by mutations and matings or, rarely, by just simple copying of the members of the previous population. The selection process to determine which individuals will participate in creating the next generation is tournament selection. In tournament selection, a small number of individuals are picked uniformly at random, and from these, one is chosen such that the probability of its being the best is highest and the probability for each below it in the ranking is exponentially lower. This ensures that fitter individuals are more likely to contribute their genetic material to the new generation.

The model is built from the new feature and the previously fixed set of features. This means that the set of features is grown iteratively, with the next feature being the focus of the search, as described in the next section. The fitness of a feature is determined by appending it to the previously fixed features and calculating the speedup that would be obtained by using a machine learning model, based on those features, in place of GCC's unrolling heuristic. A feature that improves the speedup is better than another that does not.

### 3.3. Machine Learning

The machine learning subsystem of Figure 6 is the part of the system that provides feedback to the search component about the quality of a feature. As mentioned earlier, the system maintains a list of good base features that is initially empty. It repeatedly searches for the best next feature to add to the base features, iteratively building up the list of good features. The system stops when it has failed to add a new feature that improves the results. The final output of the system will be the list of good features at the end of the search. Figure 7 shows the details of the process.

To evaluate the quality of a new feature, we first compute the feature values across all programs, as shown in Figure 7(a). The program data might be the IR for loops of a number of benchmarks and a feature might be *the depth of the loop times the number of basic-blocks*. An evaluation system computes the feature expression on each program datum, yielding a vector of the resulting values.

We then combine this feature with the previous base features and ask a machine learning algorithm to learn a model that can predict the target heuristic value (shown in Figure 7(b)). Any machine learning tool can be used. However, since the tool will have to learn a model every time we need to compute the fitness of a feature, it must be fast; we might compute the fitness of several million features during our search process.

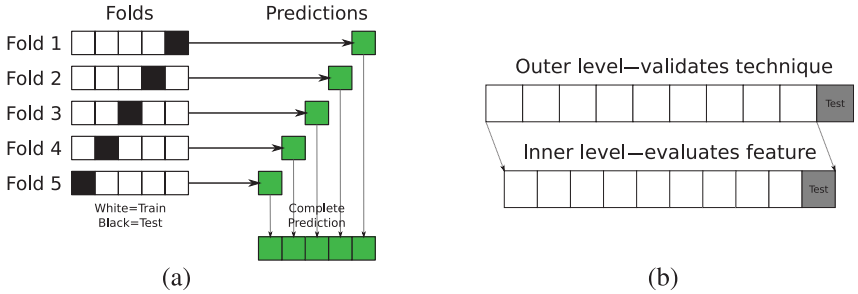



Fig. 8. Cross validation. Panel (a) demonstrates fivefold cross validation. The dataset is partitioned into five sets. For each of the five folds, a prediction is made for one of the sets with a model trained on the other four sets. The five predictions are gathered to create a complete prediction. Panel (b) shows the two levels of cross validation, with one test set held out in each. The outer level is used to evaluate the technique, and the inner one is used to evaluate a feature.

Some of the state-of-the-art machine learning tools, such as Support Vector Machines (SVMs), are very slow, sometimes by two orders of magnitude compared to simpler tools such as decision trees. In this work, then, we use C4.5 decision trees [Quinlan 1993], which are considerably faster. However, since like many search problems this could be trivially parallelised, the runtime of the slower tools could become tolerable.

Finally, we test the model's quality and report this back to the search component (shown in Figure 7(c)). In this work, we used the speedup of the prediction to be the fitness of the model for a feature. Prediction accuracy could have been used, amongst other possibilities, but we found that in this case the system may concentrate on improving accuracy for small loops at the expense of those that dominate execution time. Improving speedup is a much closer fit to the goal of compiler optimisation.

*3.3.1. Cross Validation.* Cross validation is used in machine learning to avoid overfitting to the training set. Overfitting means that the predictive model may be very good for the points in the training set but poor for other points. A typical cross validation scenario partitions the input data into a number of sets (say 10). One set is kept out and called the *test* set, the remainder (typically nine tenths of all data) is called the *training* set. A model is trained on the training set, and it predicts values for the test set. This is repeated for each partition so that each partition is the test set once and predictions are created for all of the input data. The total prediction (albeit composed from different models) is then evaluated for quality. This process is shown in Figure 8(a).

We have two levels of cross validation, shown in Figure 8(b)—an inner level used during the search that estimates how well individual feature sets are performing, and an outer level that judges how well the entire procedure has succeeded. The outer level is the more typical and is used to determine how well the feature generator works. This level means that the program data is partitioned, providing a training set to the whole feature search, as in Figure 7(a)–(c), and complete sets of features are created for each partition. The models learned from those sets are used to predict values for the test sets, and the whole prediction is used to evaluate how good the method is. This is no different from most machine learning experiments.

The inner level, on the other hand, is used during the feature search so that the search process can estimate how well each feature set is performing. The step shown in Figure 7(b) is actually cross validated (we use 10-fold cross validation, so 10 models are created for the 10 training set/test set pairs). Together, the multiple models combined build the prediction in Figure 7(c) over the several test sets.

At no point will the inner level see the outer level's test set. The inner level's training and test sets come from the outer level's training set only.

```
<expr> ::= <term> <op> <expr>
<term> ::= <id> | <num> | "(" <expr> ")"
<op>   ::= "+" | "*"
<id>   ::= ("a" | ... | "z")+
<num>   ::= ("0" | ... | "9")+
```

(a)

```
--- a = 10
--- b = 20
--- c = a * b + 12
--- d = a * (( b + c * c ) * ( 2 + 3
    ))
```

(b)

Fig. 9. A grammar for the simple language (a) and example statements from it (b).

*3.3.2. Parsimony.* In practice, the system uses information in addition to the quality metric described previously. It happens that the feature expressions learned by evolutionary search can quickly become very long. Two features can have the same results but have different lengths (e.g., if one feature is `loop.depth`, a more complicated feature with no more predictive power is `loop.depth+(1+2)×loop.depth`.

In order to address this problem, we adopt the well-known GP methodology of rewarding parsimony. If the objective function computed by the machine learning tool for two features gives the same value, then we determine that whichever feature expression is shorter is the better.

## 4. DEFINING THE FEATURE SPACE

Feature grammars are used to describe a feature space. A toy compiler language is presented first in Section 4.1, and features are defined for it. How features are computed against the compiler's IR is covered in Section 4.2. Implementation of advanced feature spaces that are difficult to define with a Context-Free Grammar (CFG) alone is covered in Section 4.3. Then, Section 4.4 shows how different areas of the feature space can be prioritised over others.

### 4.1. Features for a Simple Language

A toy example is presented to show how the process works. The toy language allows only sets of assignment statements; the left-hand side of each will be a variable name; the right-hand side will be an expression containing variables, constant integers, operators '+' and '*', and parentheses. A Backus Naur Form (BNF) for the simple language and example statements are shown in Figure 9.

A human compiler expert, when thinking of features to be computed over expressions, will most likely devise simple features like *the number of '*' operators in the expression* or *the depth of the expression*. He will implement and test these features and with luck will discover that they are somewhat helpful to machine learning but in all likelihood do not perform as well as he had hoped. He will then probably create small variations of his original features. For example, to expand on his count of multiply nodes, he may think that another feature could be to count those multiply nodes that have as their left child a '+' operator and whose right child is a constant. He can add this feature to his set and try his machine learning tool again.

There are an unbounded number of these features, however, as the compiler expert may choose to further refine his new feature by specifying the properties, recursively, for the children of the '+' node—for example, restricting the set of values for the constant or any number of complex modifications that could be imagined. The expert will repeat this process, continually implementing and testing more features until either no further improvement in the machine learned model is found or he runs out of time to experiment.

The approach taken here is an automation of this labour-intensive process. Since this system will explore the space of potential features, it must know what that space is. The space of features is described by means of a feature grammar where the language

```
<feature> ::= "countNodesMatching(" <matches> ")"
<matches> ::= "isConstant" | "isVariable" | "isAnyType"
            | ("isPlus" | "isTimes")
              ("&& leftChildMatches(" <matches> ")")?
              ("&& rightChildMatches(" <matches> ")")?
```

Fig. 10. A simple feature grammar. Sentences from the grammar are expressions that can be evaluated over statements from the toy language. Each feature counts the number of subtrees in the toy language statement that match a pattern.

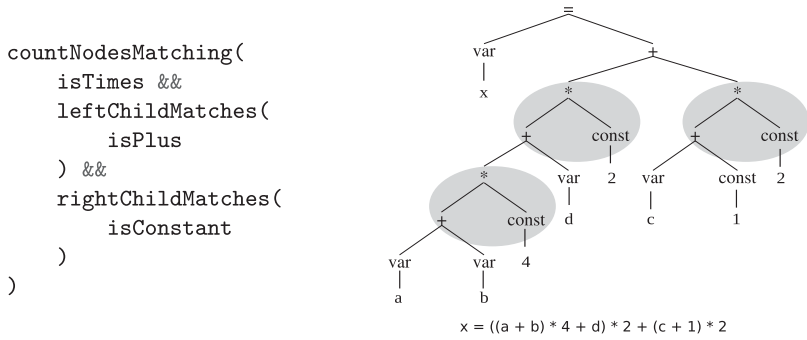



Fig. 11. An example feature from the grammar in Figure 10. In the right side of the figure is a sample AST from the tiny language, showing the matching substructures. The feature thus evaluates to three.

accepted by the grammar is some subset of an existing programming language. Each sentence[4] from the grammar will be an expression that can compute the value of a feature when run over the compiler's internal representation of the current code section. The compiler writer must choose what space of features to search over and design a feature grammar to represent that space. The design process itself is not automated; the compiler writer may make features that explore the abstract syntax tree, control flow graphs, and any other data structures available in the compiler. Feature grammars and the compiler's language grammar need not be related.

Figure 10 shows a simple grammar describing a set of such features in a pseudocode style. These example features can be computed on expressions from the toy language, and in this case, each feature counts the number of subtrees in the expression that matches a pattern. Figure 11 shows an example feature from this grammar and an evaluation against a sample program fragment, showing the matching subtrees. Applying this particular feature to this piece of code yields the value of 3.

The next section describes how a feature will be computed.

### 4.2. Feature Evaluation

The feature grammar defines a set of features, each of which is a sentence from that grammar. Once a feature has been extracted from the grammar, it will need to be evaluated against the compiler's internal data to compute feature values. A grammar is able to produce sentences that are a subset of some particular programming language. In this way, an interpreter or compiler for the features already exists in the form of whatever compilers or interpreters there are for the underlying language or if one does not exist.

[4]A sentence is a string symbols recognised by the language that the grammar represents.

```
int i1 = 123, i2 = 9, i3 = 18;
return i1 + i2 + i2 + i1;
```

Fig. 12. An example showing definitions of multiple variables followed by a use of those variables. If the code fragment were generalised to allow an unbounded number of variables to be defined and used, then this would not be representable as a CFG. CFGs do not support these semantics for sentence generation any more than they do for sentence parsing.

```
1  {int num_vars = 0;}
2  <feature> ::= "int " <defs> ";"
3                "return " <expr> ";"
4  <defs>    ::= <def>
5              | <def> "," <defs>
6  <def>     ::= "i" { print( ++num_vars ) } "=" { print( random( 0, 255 )) } ";"
7  <expr>    ::= <expr> " + " <expr> | <var>
8  <var>     ::= {print("i" + random(1 to num_vars))}
```

Fig. 13. A feature grammar with semantic actions, generalising the code fragment from Figure 12. Semantic actions are found between braces.

The underlying language in which the features are produced is typically Turing equivalent; however, using pure CFGs may make it impossible to express all types of features. The next section deals with how to handle cases where the restriction to be context free is too limiting.

### 4.3. Semantic Actions

CFGs are subject to a number of limitations that restrict the types of sentences that can be created. Consider, for example, a simple example in which a number of variables will be defined and then an expression will be created using those variables, such as is shown in Figure 12. This cannot be expressed with a CFG because a CFG cannot convey the semantics involved [Aho et al. 1986]. More realistic situations occur when building nested loops or defining functions since a symbol table is altered, which is beyond the syntactic capabilities of a CFG.

The situation has parallels in the world of program parsing. There, a CFG is used to describe the syntax of a language, and the semantics of the language are embedded as "semantic actions" ([Aho et al. 1986]) in the grammar. In parsing, these actions allow arbitrary code to be run during the parsing process; for example, symbol tables are updated and checked.

This system has a similar mechanism, allowing the grammars to produce more complicated features. Semantic actions are embedded in the grammar, and whenever a production[5] is selected, the actions are run in the appropriate place. These semantic actions are snippets of arbitrary code that can update state and print values. For example, Figure 13 shows a grammar that generalises the code fragment from Figure 12. The first action in line 1 initialises a variable counter to zero; this action is executed before any rules are expanded. The next actions are in line 6, which defines another variable and increments the count. The final action, in line 8, prints a random variable name from those available.

Semantic actions can be placed on entry to or exit from the whole grammar, rules, and productions, as well as inside productions, as shown by the example. These additional capabilities need only be used sparingly but allow extremely detailed and powerful control over the features that can be produced.

[5]A CFG consists of a number of rules, each defining the possible expansions of a single nonterminal. A production is one of the choices on the right-hand side of a rule.

```
<feature> ::= "countNodesMatching(" <matches> ")"
<matches> ::=[weight=10] "isConstant"
            |[weight=10] "isVariable"
            | "isAnyType"
            |  ("isPlus" | "isTimes")
               ("&& leftChildMatches(" <matches> ")")?
               ("&& rightChildMatches(" <matches> ")")?
```

Fig. 14. A small modification to the grammar of Figure 10, showing weighted productions. In this example, the compiler writer has decided that features testing for constants or variables should be 10 times more likely than others.

```
<digit> ::= "0" | "1" | "2" | ... | "8" | "9"
```

(a)

```
<digit> ::= "0" | <d1>
<d1>    ::= "1" | <d2>
<d2>    ::= "2" | <d3>
...
<d7>    ::= "7" | <d8>
<d8>    ::= "8" | "9"
```

(b)

Fig. 15. Two different grammars for choosing a decimal digit. Both grammars recognise exactly the same language but have significantly different preferences.

In the feature grammars used in this article, semantic actions are used to generate integer and floating point numbers with different distributions, and to provide some depth limits to some expensive functionality. Those uses of semantic actions could have been replaced by cumbersome grammar rewrites, but the resulting system would have been significantly more difficult to reason about and maintain. The grammar also uses semantic actions to define variables for various recursive graph operations, which cannot be achieved by a CFG alone. However, the evolutionary search did not select features with those operations in the final feature sets.

### 4.4. Production Weighting

There are times when the compiler writer believes that some part of the feature space is more likely to be useful than others. She would like to be able to influence the grammar to reflect her interest in different portions of the feature space. The grammar definition language allows productions to be weighted, changing the relative probabilities of productions. This makes the grammars Probabilistic ContextFree Grammars (pCFGs) and allows the compiler writer to express her interest in some features over others.

Figure 14 shows a simple example where some productions are annotated with weights. Now, when choosing between productions for the <matches> rule, the first two will be 10 times more likely than before. The compiler writer has changed the probability that some parts of the space will be explored compared to others.

Another example is shown in Figure 15. The two grammars in that figure both describe the set of decimal digits, "0" to "9." However, because of their construction, they have very different preferences for different digits. In the first, Figure 15(a), since the productions are chosen with a uniform probability, the digits will appear with equal likelihood. In Figure 15(b), however, the characters "8" and "9" are equally likely, but "7" is twice as likely as those, "6" is twice as likely as "7," and so on until "0", which will be chosen with probability 0.5, is 256 times more likely to be chosen than "8" or "9."

```
<digit> ::= "0" |[weight=9] <d1>
<d1>    ::= "1" |[weight=8] <d2>
<d2>    ::= "2" |[weight=7] <d3>
...
<d7>    ::= "7" |[weight=2] <d8>
<d8>    ::= "8" | "9"
```

Fig. 16. A weighted grammar, similar in structure to that of Figure 15(b) but that has the equal probability of choosing each digit, just as in Figure 15(a).

```
1. <feature>
2. "count-nodes-matching(" <matches> ")"
3. "count-nodes-matching( is-times &&
    left-child-matches(" <matches> ") && right-child-matches(" <matches> "))"
4. "count-nodes-matching( is-times &&
    left-child-matches(is-plus) && right-child-matches(is-constant))"
```

Fig. 17. Derivation of the example feature from Figure 11.

The grammars of Figure 15 demonstrate that the precise form of a grammar can have a dramatic effect on preferences for different parts of the space. Sometimes the most natural way of defining the grammar would lead to an unacceptable bias for some features over others. It may be quite difficult to alter the structure of the grammar to even out these issues, and instead weighting can be used to adjust the grammar back into what the compiler writer was looking for. In Figure 16, the grammar from Figure 15(b) has been weighted to have the same probabilities as the grammar in Figure 15(a).

## 5. GENERATING FEATURES FROM GRAMMARS

This section discusses how features can be created from a grammar and the design challenges that creates. The main challenge involves ensuring that sentence creation finishes, because it is quite possible to construct unbounded, recursive grammars.

How to expand a feature from the grammar is described in Section 5.1. The causes of unbounded, recursive grammars are covered in Section 5.2. The way to solve the problems of unbounded recursion is discussed in Section 5.3, and then the consequences of that solution are talked about in Section 5.4.

### 5.1. Feature Expansion

Now that there is a grammar describing the space of features, any number of features can be generated from it. One need merely start at the root rule of the grammar (which, in the case of the grammar in Figure 10, is rule <feature>) and expand any nonterminals in it. Whenever there is a choice of production to expand, they are chosen from randomly using roulette wheel selection [Back 1996], where the probability of choosing each production is proportional to its weight. By continuing until there are no more nonterminals left in the sentence, there will be a finished feature.

For the example in Figure 11, a derivation is given in Figure 17. Step one starts with the root rule of the feature grammar—placing a single nonterminal as the current sentence. In step two, the nonterminal is replaced by the only possible rule, leaving still only one nonterminal to be replaced. In step three, the <matches> nonterminal is replaced; there are five productions, and the last is randomly selected; the nonterminal is replaced with value of the production giving two nonterminals to replace. Finally, in step four, the remaining <matches> nonterminals are replaced.

```
<A> ::= <A> "a"
```

Fig. 18. Unbounded, recursive grammar. The only sentence that the grammar recognises is an unbounded sequence of as. Attempting to generate a sentence from this grammar will never finish, because there will always be a nonterminal <A> left unreplaced.

```
<A> ::= <A><A><A> | "a"
```

Fig. 19. Probabilistically recursive grammar. At any point in the expansion of the sentence, it is possible for the all nonterminals to be replaced with terminals. However, since the two productions are chosen with equal probability and the first production generates so many recursive nonterminals, there will most likely be an explosion of nonterminals and the sentence will become longer and longer.

### 5.2. Problems of Recursion

Whilst ambiguity causes problems in parsing sentences from grammars, sentence production is not vulnerable to it. On the other hand, sentence production does suffer from unbounded recursion. Perhaps the simplest example of this is in the grammar in Figure 18, which obviously produces an unending string of '*a*'s. Attempting to generate a sentence from this grammar will not succeed. The grammar definition language allows embedded actions, similar to semantic actions in parser generators, which allow such problems to be manually broken—by, for example, imposing a depth limit on the recursion. Grammars like these could be statically identified and flagged as such; however, in practice, these grammars are unlikely to be seen.

More subtle recursion issues are caused probabilistically. Consider the grammar in Figure 19. The language it recognises consists of odd numbers of consecutive '*a*'s. However, if the two productions are chosen from uniformly at random, then it is likely to produce very long strings.[6] As strings contain more nonterminals, they become increasingly likely to grow at each expansion. Production weighting solves these issues, as described next in Section 5.3.

### 5.3. Avoiding Runaway Sentence Expansion with Production Weights

Explosive sentence lengths can be avoided by changing the probabilities of different productions. If in the example from Figure 19 the weight of the first production had been less than one third of the weight of the second production, then the expected length of a sentence after replacing each nonterminal once would be less than the original; the sentence expansion would not explode.

Deciding the appropriate weights for productions is not generally possible to do analytically. This is due to the presence of semantic actions (see Section 4.3), which introduce arbitrarily complex code into the grammar expander. However, as can be seen from Figure 20, once weightings create a nonexplosive grammar, there is relatively small sensitivity to the weight values. Thus, it is quite easy to be conservative with weightings, and the grammar will not suffer much for it; trial and error produces acceptable results with very little time or effort.

### 5.4. Short Sentence Bias

As described in the previous section, the grammar must weight productions to prevent runaway, explosive sentences. A consequence of this is that the grammar system is biased toward short sentences. Figure 20 shows the sentence-length bias for the grammar in Figure 19. It can be seen that short sentences are produced, even when the

[6]The probability that a string of $n$ nonterminals, $AAA \ldots A$, will contain fewer nonterminals after each is expanded once is given by $I_{1/2}(2n/3, n/3 + 1)$, where $I$ is the regularised incomplete beta function. This quickly approaches 1 as $n$ increases.

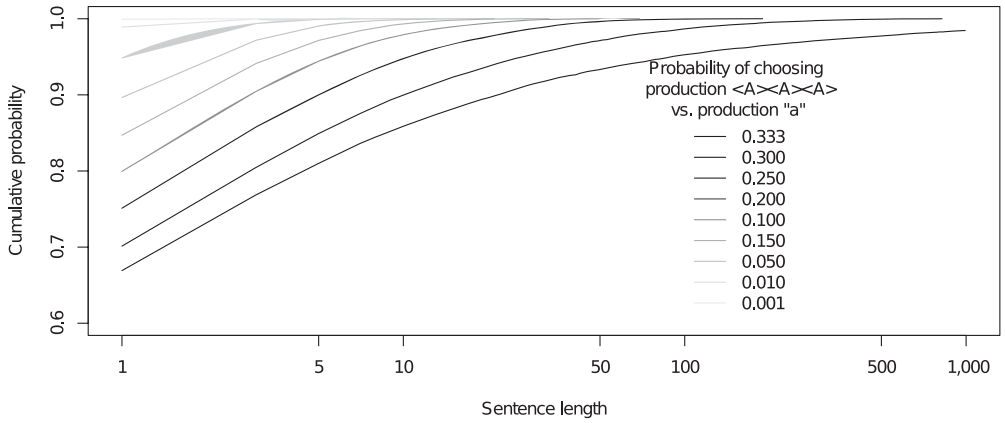



Fig. 20. Cumulative probability of getting a sentence of a given length when expanding grammar <A> ::= <A><A><A> | "a" with different weights for the two productions. Even as the weights approach the beginning of runaway sentence generation (which first happens when production <A><A><A> is one third as likely as production "a"), there is significant bias toward short sentences. The graphs also help to explain why setting weights by trial and error is trivial. Being conservative with the weights does not have a large effect on the sentence length; they will always be short for feasible weights.

productions are weighted close to the threshold at which runaway expansion begins. For this grammar, the vast majority of sentences will be shorter than 10 symbols long.

Creating grammars that give rise to mostly short sentences has some drawbacks. In essence, by avoiding the explosion, the preference for short sentences prevents the exploration of much of the grammar. Often, after generating a few thousand random sentences, the system typically recreates sentences that have already been seen; exploration effectively stops.

Overcoming the bias toward short sentences is a side effect of searching the feature space, which will be covered in Section 6.

## 6. SUPPORT FOR EVOLUTIONARY SEARCH OF THE FEATURE SPACE

The simplest way to search the feature space is to randomly generate features as described in Section 5. Thousands of features can be created in a matter of seconds that can then be evaluated. The bias toward short sentences, however (see Section 5.4), means that this approach will be practically limited to a small portion of the complete feature language accepted by the pCFG. Early experiments found that the amount of the feature space that could be reached was much smaller than expected.

One might try to use semantic actions to alter the bias of the feature space as the search begins to saturate its exploration of short features. This, however, is difficult to arrange, placing a heavy burden on the writer of the grammar to add large amounts of fragile code that are not directly concerned with describing the feature space. One could also arrange the weights of productions to favour long sentences. This quickly leads to runaway sentences in the feature generator. Even if the generator is rigged to bail out when a sentence becomes too large, the generator then spends most of its time creating features that fail or that have already been seen before.

Fortunately, the problem of short sentence bias is solved as a side effect of different search techniques than the naïve random approach. Suppose that there is some feature to start with, chosen from the biased space. Small modifications can be made to it; possibly shortening it, maybe lengthening it. The feature will no longer be bound by the imposed bias of the pCFG (the bias will now only inform where the search should start

and its direction, not limit the scope of that search). Thus, by employing evolutionary search, the short sentence bias is no longer an issue.

Section 6.1 describes the trees that are searched over. Operators that modify choice trees are discussed in Section 6.2. Search operators may leave a choice tree without sufficient information to properly produce a feature; methods to repair a choice tree after modification are covered in Section 6.3.

### 6.1. Choice Trees

The entities that are searched over in the system are the trees of choices that are made during the construction of a sentence or feature. The tree is similar to a parse tree, except that instead of containing a node for each nonterminal and terminal in the parse tree, it encodes the choices between rule productions and any other necessary data that lead to a parse tree. Since some rules may have only single productions, they do not require choices to be made, and so have no representation in the choice tree. The trees record everything needed to efficiently invoke genetic search operators on the features.

A choice tree encodes the choices that were made during the generation of a feature or sentence from a pCFG. Each subtree describes the choices made for the expansion of a single rule and its children. Each node contains the random bits that were used to select the production and any random data needed by the semantic actions of that production. A choice tree is very similar to the parse tree for a sentence, with random bits attached, except that it may also contain redundant information that is not used in the derivation of the feature. For these purposes, it is assumed that the grammar is written in BNF. Choice trees can be used in two complementary fashions; one records the choices made during the expansion of a feature, and once recorded, it can be used to replay those choices to recreate the exact same feature as before. During the recording, any random bits are remembered and subtrees delimited according to the rules and productions of the grammar. When replaying, the source of random bits is replaced by those recorded previously.

An example choice tree is given in Figure 21. A simple grammar is shown in the first block of Figure 21(a). Next, in Figure 21(b) is a possible derivation of sentence `babb` starting from `<A>`, where in each step one nonterminal is replaced with a production. Finally, in Figure 21(c), there is a choice tree that generates the preceding derivation. The first `<A>` rule has three choices, so a random number is generated, `219` (for simplicity, these are assumed to be just one byte long). This is used for roulette wheel selection, wherein the probability of selecting each production is proportionate to its weight. There are three choices for `<A>`'s productions, and since all productions have unit weights, roulette selection is equivalent to the modulo function. The random number, `219`, modulo `3` is `0`, so the first production, `<A>` → `<A><A>`, is chosen. Subsequent nodes in the tree represent the remaining replacements of nonterminals. In the cases where the `<B>` nonterminals are replaced, there is no choice; denoted by `X`.

### 6.2. Search Operators

The system offers several search operators suitable for evolutionary search, hill climbing, or other techniques. Each of these requires some method to take some number of promising trees and create one or more new, candidate trees that will be the next points that the search will consider. In hill climbing, a single tree is mutated in some fashion to create a new tree. In evolutionary search, a new tree may also inherit parts of two trees.

Operators on choice tree nodes include:

—*Deletion:* The subtree will be replaced with a new random tree during replay.

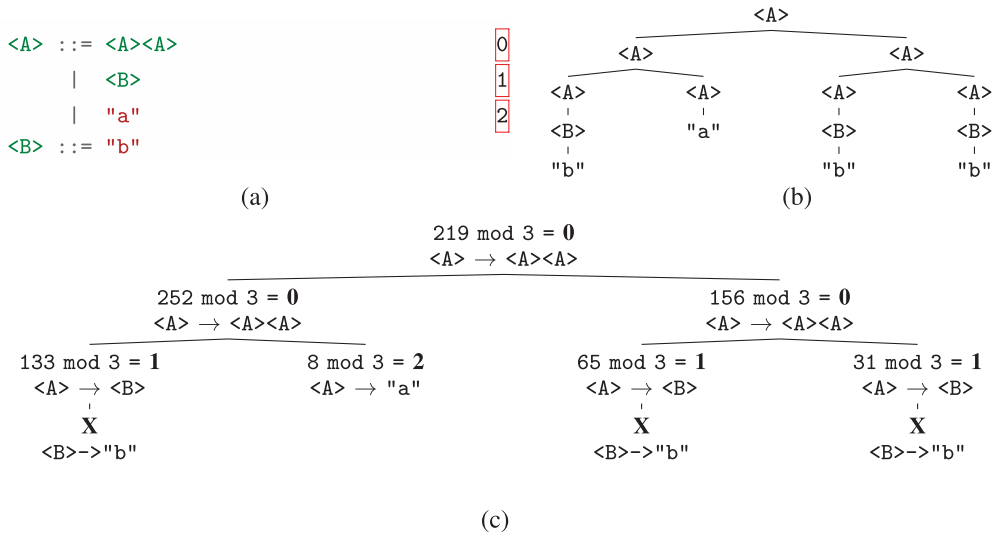



Fig. 21. An example choice tree. A simple grammar is shown in (a), where the productions of the rule for nonterminal <A> are numbered for clarity. A possible derivation of sentence babb is given in (b). In (c), there is a choice tree for the derivation in (b); when selecting a production for rule <A>, a random number is needed (in this example, from [0, 255]), and the production is then chosen by roulette wheel selection (equivalent to mod 3 in this case) to give the production number. Not all rules offer choices and so do not need random bits (marked with **X**).

—*Change random bits:* The node may now select a different production; this may cause some children to be missing or irrelevant. Often, rules have many productions with similar forms (e.g., a rule for binary expressions may have many productions with the same two nonterminals); changing random bits might simply swap one such production for another. Constants are also generated from random bits, so this type of mutation can search through different constants.
—*Shuffle children (several variants):* If the children are from unrelated nonterminals, then this might be equivalent to deleting all of the children and recreating them (although with some prespecified random bits). If children are related, such as children of many binary operators, then their orders may be changed; for example, the left and right children of a subtraction may be swapped.
—*Crossover subtrees:* Two subtrees from two choice trees are swapped over. This allows information to be shared across choice trees, as is required for GP techniques. There are two main variants of this operator: the first chooses any subtrees from the two choice trees; the second is more targeted—it first expands both trees, remembering the names of the rules to which each node in the trees relates. Then, it prefers to select two subtrees that relate to the same rule—that is, having the same nonterminal. Both of these main variants have parameters specifying likelihoods for subtrees to be chosen, based, for example, on the depth or node count of the subtree.

### 6.3. Repairing Choice Trees

The search operators in Section 6.2 perform modifications to the choice trees. They may change a choice tree so that there are insufficient random bits to complete the replaying of the tree to create a feature. For example, a mutation search operator may delete some subtree or change a simple production without children to one that requires

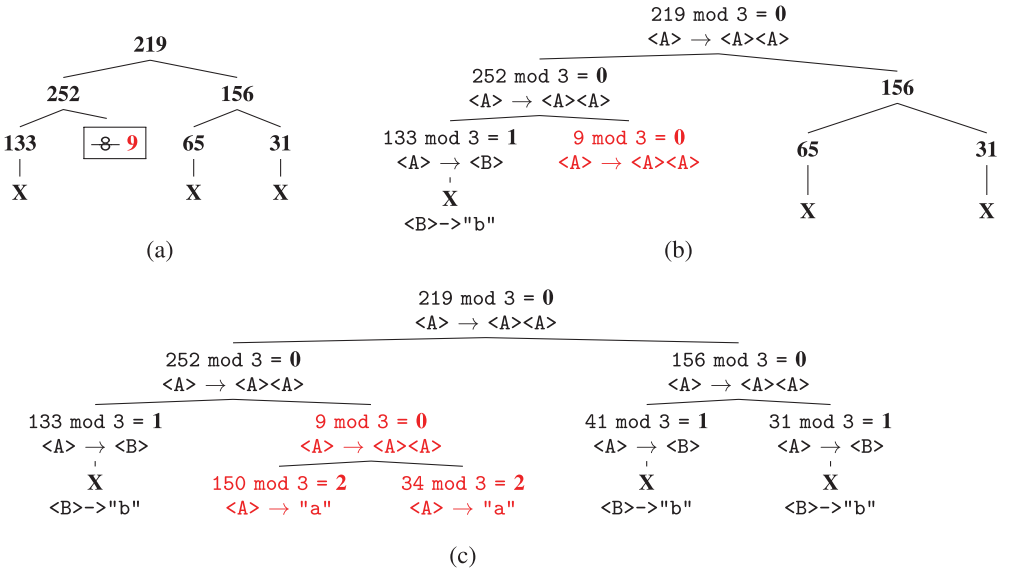



Fig. 22. Repairing a choice tree. In (a), the choice tree from Figure 21 has one node mutated (marked with a box). Replay begins reading the choice tree in (b) as normal until the mutated node is reached when a different production is chosen. The new production needs more data than the tree contains, so that is generated and recorded in the tree (c). Thereafter, the remains of the tree are unchanged.

several children. This section describes how these trees are repaired so that they are always valid; no search operation, no matter how drastic, can create an invalid tree.

The mechanism for repairing trees is that whenever additional bits are needed during playback, they are created at random and recorded back into the choice tree. During playback, whatever bits are present in the tree in the correct places are made useful. However, if at any point there should be missing information, that will cause a change from playback mode to recording mode until the required information is made up. In this way, reading a choice tree can alter the tree as a side effect, but at no point is the tree starved of information.

Figure 22(a) shows the choice tree from Figure 21 that has been mutated during a genetic search so that the random bits of one node have changed (marked with a box). The system begins to replay the tree (Figure 22(b)), expanding the root nonterminal making choices according to the data in the tree. This proceeds just as normal until the mutated node is encountered. At this point, the random bits in the selected node choose a different production from the one in the original tree; the <A> becomes <A><A>, not "a." Now, if the tree were complete, there should be two child nodes beneath the mutated node, which are missing. To solve this issue, the system begins to randomly create any nodes that it needs, building a new subtree, rooted at the mutated node (Figure 22(c)). Once returned from repairing the mutated node, the system begins replaying, just as normal, yielding a complete feature and updating the tree so that it is now complete and has remembered the new information that was used to repair the tree.

It may also happen that modifications to a choice tree increase the information in the tree, beyond what is needed. This occurs if bits that would select a node with many children are changed so that the node will only have one child, or additional bits are added to a node that are not used for the selected production. This does not damage the choice tree; the additional data simply has no effect on the feature produced from the tree.

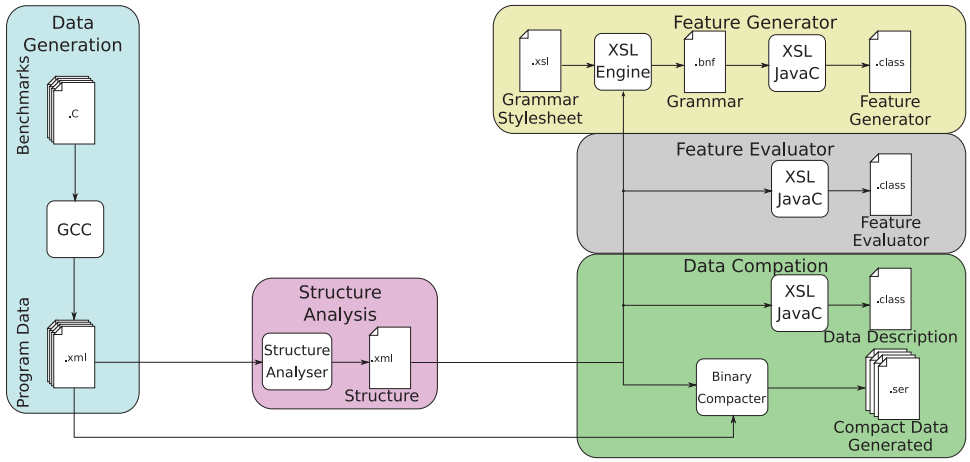



Fig. 23. Overview of the grammar creation system for RTL loops in GCC.

The system offers two modes of replaying a choice tree; in the first, any extraneous data are not removed from the tree, and in the second, extra bits and nodes are pruned from the tree. Now that it has been seen that no change to choice trees can damage them beyond repair, the next section looks at the types of choice tree search operators that the system provides.

## 7. GRAMMAR FOR LOOPS IN GCC

This section shows one of the grammars created. This is a grammar for generating features over loops at the Register Transfer Language of GCC (RTL) and is used in the experiments of this chapter. The grammar contains many tens of thousands of productions because of the large number of different data types in GCC's internal representations. The grammar is automatically created by observing the types of data that GCC uses internally; doing this is essential because of the grammar's size but also prevents the grammar from needing to be hard coded, allowing modest changes to GCC without requiring an engineer to rewrite the grammar.

An overview of the system is shown in Figure 23. The system builds three main outputs from observing GCC's internal representation of the benchmarks; the first is a suite of Java classes representing the grammar, the second is a custom feature evaluation language, and finally, the third is a compact and efficiently searchable version of the benchmark data. Subsequent sections describe the major components of the grammar generator.

### 7.1. Data Generation

The first stage of the system extracts GCC's internal data structures for each of the benchmarks into XML files.

In RTL, instructions are in an algebraic form with a list-of-lists representation. Each node in the RTL may have some number of attributes. The RTL representation of the loops is extracted, augmented to include the structure of the basic blocks in the loop and the RTL instructions contained within their blocks. Also exported is any information that GCC can compute at that time, such as estimated block frequencies, loop depths, and so on. A flavour of the data is given in Figure 24.

```
<loop name="UTDSP.fft_1024.fft.3" ... insns="28" expected-iter="11">
  <basic-block index="10" ... frequency="9100" loop-depth="3"> ...
    <insn ...> ...
      <set ...>
        <reg ... mode="SF"> <int>112</int> ... </reg>
        <mult ... mode="SF">
          <reg mode="SF"> <int>94</int> ... </reg>
          <reg mode="SF"> <int>84</int> ... </reg>
        </mult>
      </set>
    </insn>
```

Fig. 24. XML representation of RTL. Many of the attributes and values made available are removed for clarity. Attributes such as `expected-iter` and `frequency` are GCC's guesses, which it estimates in the absence of profiling information. The example shows part of the third loop from the function `fft` in UTDSP benchmark, `fft` 1024. One instruction inside one of the basic blocks is shown, which sets the value of one register to the product of two others.

### 7.2. Structure Analysis

The hierarchical data produced follow a number of relational rules. For example, loops contain basic blocks as children, and they in turn contain instructions. Those relationships are never violated. It would be wasteful to create a feature like "*count the number of basic blocks that contain three loops*" since that will always evaluate to zero.

The grammars constructed are automatically derived from the structural rules of the data to ensure that such impossible features are never generated, improving the efficiency of the search. Since the rules that GCC's internal data structures follow are not immediately derivable in a machine readable manner from the GCC source code, a simpler approach is taken of examining the XML data files to find the observed structure within.

The XML data files are iterated through, and the number of each type of node in the XML is collected; each node type's number of children is recorded; and a histogram of child types for each child slot in each node type is built, as is a histogram of attribute values for each node type. These analyses are collated into a "structure document" for use in later stages.

### 7.3. Data Compaction

To improve performance, the XML data files, produced as described in Section 7.1, are converted into a compact binary format using the structure document as a guide. The structure document declares all of the different types of nodes, their attribute names and values, and the children that each node can have.

The system first creates a Java class that represents each type of node. Each attribute maps to a field of a suitable type, and the node's children are packaged as an array. The structure document is used as source data so that the system knows what Java classes are needed. Next, the XML program data files are read, and each is converted to objects of the Java classes and serialised to files. The resulting data are much smaller than the original XML, can be loaded into memory all at once, and are much faster to compute feature values over.

### 7.4. Feature Evaluator

Although feature evaluation could be performed by an existing scripting language, generic scripting languages were found to be far too slow for searching over. Compiling

```
<numeric> ::= <numeric> ( "+" | "-" | "*" | "/" ) <numeric>
            | <value-of-an-attribute>
            | ( "sum" | "min" | "max" | "avg" )
              "(for-each-child-that(" <match> "do" <numeric> "))"
            | "count-children-matching(" <match> ")"
            | "on-child" <random> "do" <numeric>
<match>   ::= <match> ( "or" | "and" | "xor" ) <match>
            | "not(" <match> ")"
            | <numeric> ( "<" | ">" ) <numeric>
            | "is-type(" <node-type> ")"
            | <attribute> "=" <value>
```

Fig. 25. A simplified subset of the automatically generated grammar.

features into C programs and Java programs was also tried. In C, the effort of spawning GCC and then the feature program also proved too slow. For Java, the compilation could be performed in process, but the resulting classes, after being loaded into memory, tended not be garbage collected, so eventually all memory was exhausted.

Instead, a custom feature interpreter was created. There was no need to create a parser for the interpreter, since the feature grammar can produce ASTs directly, not just strings. The interpreter supports:

—Getting the types of nodes.
—Getting attribute values of nodes, including determining if attributes are missing.
—Logical, comparison, and arithmetic operations.
—Aggregators (i.e., summations, finding extrema).
—Iterating over children, descendants.
—Standard control flow operations.
—Pattern matching.

Runtime support for feature evaluation is also computed from the structure document.

The next section describes how features are created that make use of this interpreter.

### 7.5. Feature Generator

This structural information is then used to mechanically create a grammar. Because the grammar generation is automated and not hard coded, grammars are easy to update in response to changes in the compiler. The grammars, being machine generated, are quite large—several hundreds of kilobytes long. The transformations used in practice all make sure that trivially impossible features (or rather, uninteresting features that do not relate to possible structures) are never created. They also automatically set production weights to ensure that grammars do not suffer from the probabilistic recursion issues described in Subsection 5.2.

Figure 25 shows a pared-down snippet of the grammar, giving an example of some of the feature expressions that can be created. The full grammar has many more functional capabilities and is also tuned to the structure of the RTL. This bit of the grammar says how some numerical values can be computed on nodes from the data: a numeric can be a binary expression of two numerics; they may be the numerical value of an attribute; they may aggregate recursive numerical values of filtered children; they might count the number of children matching some criterion; or they might simply delegate to another numeric expression of one of its children. Matching criteria for selecting child nodes may be the logical combination of other matchers; they may be the result of comparing two numerics; they may check the type of the node; or they

may test the value of an attribute. This part of the grammar is replicated many times for each bit of structural data. There are also many other functional concepts encoded in the grammar in similar ways.

## 8. EXPERIMENTAL SETUP

In this section, we briefly describe the experimental setup and how the training data were generated as well as the steps taken to ensure accuracy of measurement.

### 8.1. Compiler Setup

To demonstrate the applicability of our approach, we have applied it to loop unrolling within GCC 4.3.1. Loop unrolling is an extensively studied optimisation, and there exists prior work [Monsifrot et al. 2002; Stephenson and Amarasinghe 2005] with which to compare. We extended the compiler to allow unroll factors to be explicitly specified for each loop in a program.

### 8.2. Benchmarks

We took 57 benchmarks from the MediaBench, MiBench, and UTDSP benchmark suites. Those benchmarks from the suites that did not compile immediately, without any modification except updating path variables, were excluded.

### 8.3. Platform

These experiments were run on a single unloaded, headless machine, an Intel single-core Pentium 6 running at 2.8GHz with 512Mb of RAM. All files for the benchmarks were transferred to a 32Mb RAM disk to reduce I/O variability.

### 8.4. Generating Training Data

In order to learn the best unroll factor, we need to generate training data where we know the best unroll factor for each of the training loops. To find this, we took each loop, one at a time, and unrolled it by different factors, zero to 15 (with zero meaning do not unroll, and 1 meaning unroll once). This gave a compiled program for which all but one loop has the default unroll factor as determined by GCC's default heuristic; that one loop was forced to take on the particular unroll factor being tested. We executed each of these versions of the program a number of times, in each case recording the number of cycles required to execute the function containing the loop that had been altered. We compiled without inlining to increase the independence of loops. All loops that GCC determined were potential unrolling candidates were used; this included both inner and outer loops. In GCC, loops can be unrolled even if the number of iterations is not known at compile time, such loops were also included. In total, we gathered data for 2,778 loops.

The range, zero to 15, was chosen because GCC's default heuristic only ever chose factors from this range over the training benchmarks. Our goal was to demonstrate the value of automatic feature generation, compared to manually constructed features.

### 8.5. Measurement

One of the difficulties in evaluating the performance of compiler optimisations is the impact of noise on the measured results. For each differently compiled variation of a benchmark, we ran that version of the program at least 100 times or until the 95% confidence interval divided by the mean was less than 0.5%. We applied a standard statistical technique to reduce the effects of noise: applying a log transform and removing outliers outside $1.5 \times$ IQR (interquartile range). The best unroll factor for each loop was determined as that with the lowest average cycle count (across the 100 runs).

## 9. EXPERIMENTAL METHODOLOGY

This section outlines the methodology used when applying the feature search technique to the problem of loop unrolling in GCC.

### 9.1. Searching for Features

The feature generator searches for one feature at a time. It prefers features that, in combination with the features selected by previous steps, most improve the performance of a machine learning tool. The evolutionary search for each feature consisted of a population of 100 individuals. Each was allowed to run until 15 generations produced no improvement in the best feature of the population or a maximum of 200 generations, whichever came first. Search for new features to add was stopped when either 2,500 total generations were reached or when we failed to find an improving feature five times.

### 9.2. Cross-Validation and Machine Learning

We split the loops into 10 groups, keeping one group out for testing so that we can perform 10-fold cross validation. Loops that are used for generating features and later learning a model are *never* used to evaluate the model. Final evaluation is always on *unseen* loops.

The machine learning algorithm used to find the quality of the features was a simple C4.5 decision tree [Monsifrot et al. 2002], selected for its speed. When a feature was evaluated, we trained a decision tree on eight of the remaining nine loop partitions, called the *training* set. We then asked the decision tree to predict the unroll factors for loops in the remaining (ninth) part, called the *internal validation* set. This was then used to determine the speedup attained by those unroll factors.

### 9.3. Search, Training, and Deployment Cost

It took the system 2 days to learn the best set of features and model for this problem. Although this is a significant amount of time, it is a one-time activity that it is performed "at the factory" and can be easily parallelised. If we consider the amount of time that it takes for a compiler writer to develop a good heuristic, this cost is in fact small.

It is theoretically possible for the system to produce extremely computationally expensive features, increasing compile time. The system forces these feature evaluations to time out, giving them at most 2 seconds to evaluate over all loops (thus, with 2,778 loops in 2 seconds, no feature can take more than 0.7ms to compute per loop on average). If a feature times out, it is discarded and cannot contribute to the gene pool. We find that the pressure for simpler features means that features rarely time out. The features selected by the system for unrolling have no significant impact on GCC's execution time.

## 10. RESULTS

This section evaluates our technique when applied to loop unrolling, demonstrating that it outperforms existing approaches. We first show the maximum benefit available from loop unrolling across the benchmark suite and to what extent GCC is able to achieve this. We then compare our approach against GCCs and a start-of-the-art machine learning scheme. This is followed by a brief analysis of the results.

### 10.1. Maximum Performance Available: Evaluating GCC's Heuristic

In order to determine how well our technique and others perform, we first conduct a limit study. As described in Section 8, we exhaustively enumerated loop unroll factors up to 15 for each loop, recording the best setting for each. This limit was chosen because it was the largest unroll factor chosen by the default heuristic for any loop. We then ran each benchmark with the best unroll factors set and recorded the speedup. The

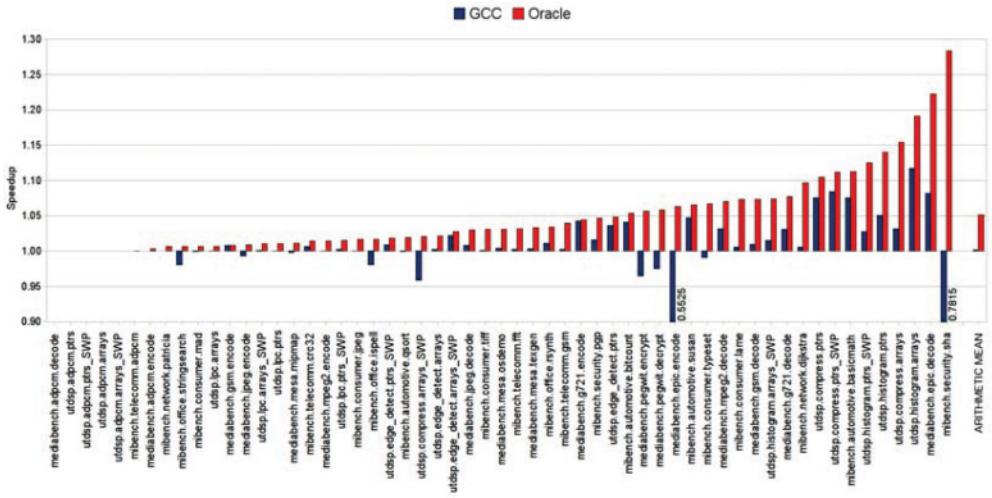



Fig. 26. Speedup of the unroll factors chosen by GCC's default heuristic and the speedup of the best possible unroll factor—the oracle.

bars labeled Oracle in Figure 26 show the maximum achievable speedups compared to no unrolling for the benchmarks.

What is immediately obvious is that the impact of loop unrolling varies dramatically across benchmarks with an average speedup of 1.05. For some benchmarks, such as `adpcm` from *MediaBench*, no unroll factor has an impact on performance. In the case of `security_sha` from *MiBench*, however, there is a potential speedup of 1.28. What we would like is a scheme that is able to exploit this potential: delivering speedups when they are available and not slowing the program down otherwise.

If we now consider the set of bars in Figure 26 labeled GCC, we see the performance of GCC across the same benchmark suite. In some cases, it is able to achieve speedup, 1.12 on `histogram.arrays` from *UTDSP*, yet in the case of `security_sha`, which has the biggest potential for performance gains, it delivers a large slowdown of 0.78. In fact, it slows down 12 of the benchmarks, the worst being `epic_encode` from *MiBench*, where the slowdown is 0.55. This demonstrates the difficulty that compiler writers have in developing a portable optimisation that delivers performance gains.

### 10.2. Our Approach

Given the potential performance available from loop unrolling and GCC's poor performance, we now demonstrate how our approach improves upon that.

*10.2.1. Comparison with GCC and Oracle.* The bars in Figure 27, labelled `Our Technique`, show the speedups of our approach across the benchmark suite. On average, we are able to achieve 76% of the maximum available. In those benchmarks where there is large potential speedup available, such as `security_sha`, we are able to achieve a speedup of 1.21 compared to GCC's 0.78. In fact, if we concentrate on the benchmarks where there is significant speedup available (>1.10 speedup), we are able to achieve 82% of the maximum. Thus, we have a technique that on average delivers more than 75% of the maximum speedup available. This is achieved entirely automatically and compares favorably with the 3% achieved by the handcrafted GCC heuristic.

*10.2.2. Comparison with a State-of-the-Art ML Technique.* Although our technique performs well, this may be due to the particular machine learning algorithm rather than carefully generating the correct features. In this section, we evaluate an alternative state-of-the

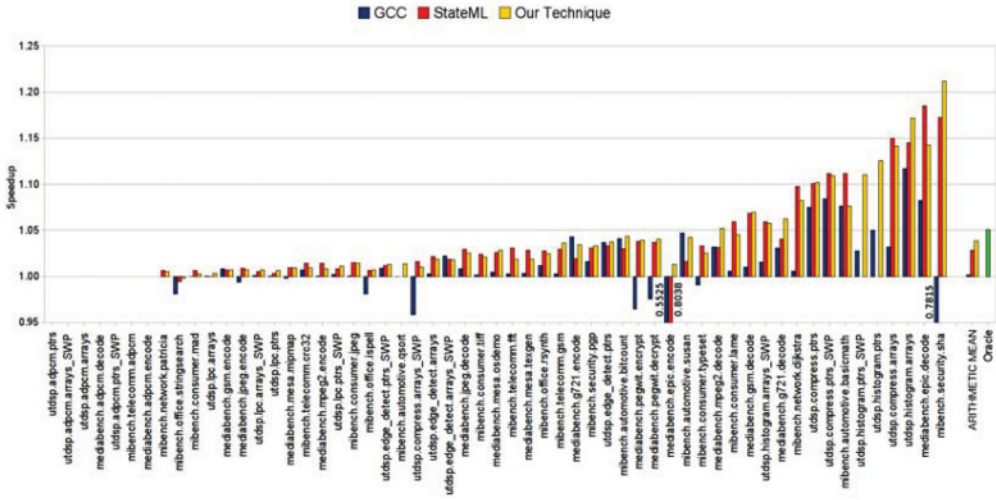



Fig. 27. Comparison of speedups between GCC's heuristic (`GCC`), our technique (`Our Technique`), and the state-of-the-art ML approach (`StateML`). GCC achieves an average of 3% of the performance available and `StateML` achieves 59% of the performance available, whereas our approach achieves 76% of the maximum performance available.

| | |
|---|---|
| Loop nest level | Num of operations in loop body |
| Num of floating point operations in loop body | Num of branches in loop body |
| Num of memory operations in loop body | Num of operands in loop body |
| Num of implicit instructions in loop body | Num of unique predicates in loop body |
| Estimated latency of the critical path of loop | Estimated cycle length of loop body |
| Language (C or FORTRAN) | Num of parallel "computations" in loop |
| Max dependence height of computations | Max height of memory dependencies |
| Max height of control dependencies | Avg dependence height of computations |
| Num of indirect references in loop body | Min memory-to-memory loop-carried dependence |
| Num of memory-to-memory dependencies | Trip count of the loop (-1 if unknown) |
| Num of uses in the loop | Num of definitions in the loop |

Fig. 28. The `StateML` features.

art scheme (`StateML`) based on an SVM described by Stephenson and Amarasinghe [2005]. We implemented this technique within GCC using the features described by Stephenson and Amarasinghe [2005] and shown in Figure 28. This model was trained and evaluated using cross validation in exactly the same manner as ours. The bars labelled `StateML` in Figure 27 represent the performance achieved using this technique. On average it achieves 59% of available speedup, outperforming GCC, which given that this was achieved automatically is significant. However, this is still short of the maximum achievable.

For the SVM method [Schölkopf and Smola 2001] we used the "one-versus-all" approach where we learn K different classifiers (one for each unroll factor), each trained to distinguish the examples in a specific class from the examples in all of the remaining classes. At prediction time, when a new loop is presented, the classifiers are executed and the class (unroll factor) with the largest output is selected. In our experiments, we have used the Gaussian Radial Basis Function (RBF) kernel:

$$k(\vec{x}, \vec{x}') = \exp\left(-\frac{|\vec{x} - \vec{x}'|^2}{2\sigma^2}\right), \tag{1}$$

with $\sigma = 1$, and we have set the upper bound parameter (C) of the SVM to 10.

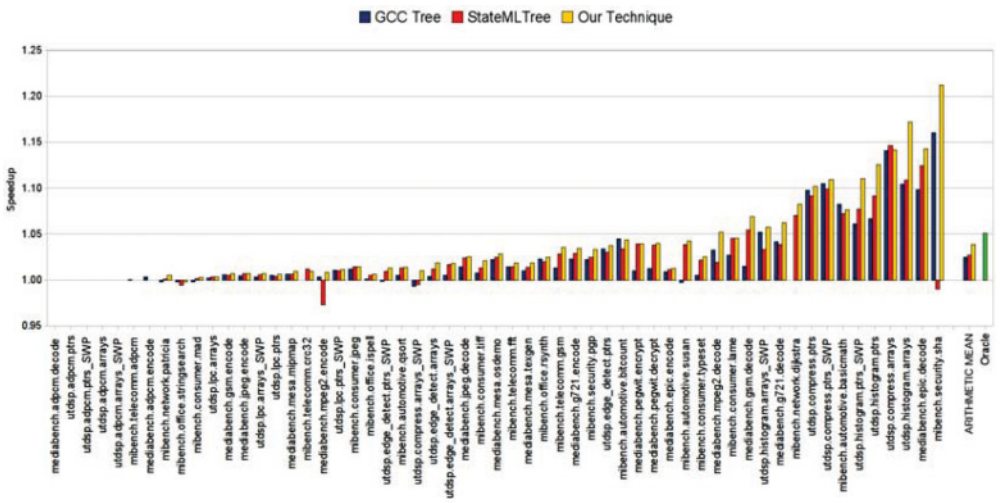



Fig. 29. Speedup using a decision tree as the learning technique for GCC's and StateML's feature sets compared to our technique. Keeping the machine learning algorithm identical shows the relative merits of the feature sets.

*10.2.3. Using Decision Trees for GCC and the* StateML *Features.* Although we have shown that our approach is superior to both the default heuristic in GCC and the StateML technique, it may be argued that both alternative schemes have good features, that (1) GCC just has poorly implemented heuristics and (2) that the StateML approach may not be best suited to loop unrolling within GCC given that is was developed within a different compiler setting.

We therefore applied the same machine learning procedure based on decision trees using both GCC' s and StateML's features, the results of which are shown in Figure 29. Thus, each of the three different approaches, GCC, StateML, and our technique, share the same machine learning model, differing in only their choice of features. Using this approach, the GCC-based features (labeled GCC Tree) are able to achieve 48% of the maximum performance available, a significant improvement over the 3% available from the default heuristic. The StateML features (labelled StateML Tree) slightly worsen from 59% to 53% of the maximum available. However, combining the two sets of features, GCC and StateML, has no further impact on performance.

These results show that machine learning with these other features does work but is limited to approximately half of the maximum performance available. By searching for the best features in tandem with learning a heuristic, we have been able to automatically improve this performance to 76%.

The average speedup of all techniques, as a percentage of the maximum, is shown in Figure 30.

### 10.3. Best Features Found

Figure 31 presents the first 6 out of 30 features found by the system in one fold. The speedup that feature attains, when combined with previous features, is given, as is the translation of that speedup into a percentage of the maximum possible. We also show how much more of the maximum speedup each consecutive feature brings.

A few of the expression elements from the best features are explained next:

—*count(s):* Returns the number of elements in sequence *s*
—*filter(s.m):* Filters sequence *s* removing any not matching expression *m*

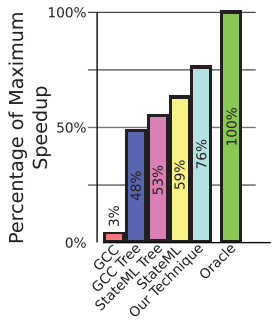



Fig. 30. Average speedup of all techniques as a percentage of the maximum available speedup.

| | Speedup | % Max | Diff | Feature |
|---|---|---|---|---|
| 1 | 1.01971 | 38.63% | 38.63% | get-attr(@num-iter) |
| 2 | 1.02665 | 52.22% | 13.59% | count(filter(//*, !(is-type(wide-int) \|\| (is-type(float_extend) && [(is-type(reg)]/count(filter(//*,is-type(int))))) \|\| is-type(union_type)))) |
| 3 | 1.03089 | 60.52% | 8.30% | count(filter(/*, (is-type(basic-block) && (!@loop-depth==2 \|\| (0.0 > ((count(filter(//*, is-type(var_decl))) - (count(filter(//*, (is-type(xor) && @mode==HI))) + sum(filter(/*, (is-type(call_insn) && has-attr(@unchanging))), count(filter(//*, is-type(real_type)))))) / count(filter(/*, is-type(code_label))))))))) |
| 4 | 1.03353 | 65.70% | 5.18% | max(filter(/*,(is-type(basic-block) && !(@loop-depth==3 && @may-be-hot==true))), count(filter(/*,(is-type(insn) && /[5][(is-type(set) && /[0][(is-type(reg) && !@mode==DF)])])))) |
| 5 | 1.03448 | 67.55% | 1.86% | count(filter(//*,is-type(array_type))) |
| 6 | 1.03503 | 68.62% | 1.07% | count(filter(//*,(is-type(le) && !has-attr(@mode)))) |

Fig. 31. Best features found by our feature search in one fold. Column 1 gives the index of the feature. Column 2 gives the speedup of the learned heuristic with all features up to that point. Column 3 shows how the model fared compared to the oracle. The next column is the percentage improvement that that feature adds, then the feature is shown.

—*sum(s,e):* Takes the sum of expression *e* applied to each member of sequence *s*
—*is-type(t):* Determines if the current node is of type *t*
—*/*, //*, [n]:* The children, descendants, and particular child of the current node, respectively

The first and most important feature computes the loop's number of iterations; clearly, there is no point unrolling a loop more times than it has iterations. The remaining features are less obvious and are unlikely to be picked by a compiler writer, demonstrating the strength of the approach. The features display elements that appeal to the intuition but are nonetheless complicated, and it is difficult to explain exactly why some elements are present. This is an artefact of the objective function, which attempts, when adding a feature, to find the most helpful feature for the machine learner but makes no effort to find features whose rationale can be understood by a human. Semantic actions were used to generate the numeric constants in these features. The meaning of the features is given in Figure 32.

## 11. RELATED WORK

The first work to automate compiler optimisation was in the area of iterative or feedback-directed compilation, which searches the program optimisation space. Many smart search heuristics have been developed [Cooper et al. 2005; Kulkarni et al. 2004; Pan and Eigenmann 2006; Triantafyllis et al. 2003]. Cooper et al. [1999, 2002, 2005] explore the optimisation space using hill climbing and genetic algorithms. Other researchers have used analytical [Yotov et al. 2003] or empirical [Agakov et al. 2006] models to explore the optimisation space.

Agakov et al. [2006] built a model offline that is used to guide iterative compilation search. Their approach yields a search technique that achieves excellent performance

*Feature 1.* Get the 'number of iterations if constant' attribute of the loop
*Feature 2.* Count the number of nodes at any depth in the loop that are
neither `wide-int` nodes
nor `float_extend` nodes for which
the first child is not a `reg` node with the count of `int` nodes beneath that `reg` node is zero
nor `union_type` nodes
*Feature 3.* Count the number of `basic-blocks` for which
the `loop-depth` was not 2 or
(number of `var_decl` nodes in the block minus
the number of `xor` nodes in the block with `mode` attribute `HI` minus
the sum, over all `call_insns` with an `unchanging` attribute of
number of `real_type` nodes
divided by the number of `code_labels` in the block)
is less than zero
*Feature 4.* For each `basic-block` that is not both of depth 3 and hot, compute
the number of instructions which have child 5 that
is a `set` and its first child is
a `reg` whose `mode` attribute is not `DF`.
Then take the maximum over all `basic-blocks`
*Feature 5.* Count the number of `array_type` nodes at any depth in the loop
*Feature 6.* Count the number of `le` nodes at any depth in the loop that do not have a `mode` attribute

Fig. 32. Meanings of features from Figure 31.

in a few dozen iterations, but it cannot compare to our approach for heuristic replacement. The first iteration of their procedure (equivalent to the single compilation of our technique) gets only 35% of the maximum available performance compared to our 76%. Their work did, however, show that machine learning features are useful across different architectures, leading to only small differences in the search efficiency. Many other works have demonstrated the effectiveness of machine learning across architectures. Wang and O'Boyle [2009] and Dubach et al. [2009] make use of statistical inference to select good optimisations. In contrast to our work, these approaches do not make use of predictive modelling and require recompilation of the program for each step in searching the optimisation space. On the other hand, our technique focuses on avoiding this search and recompilation by directly predicting the correct set of compiler settings with no profile runs.

More recently, there have been a number of papers aimed at using machine learning to tune individual optimisation heuristics. Some of the first researchers to incorporate machine learning into compiler for optimisation were McGovern and Moss [1999], who used reinforcement learning for scheduling of straight-line code. Cavazos and Moss [2004] extend this idea by learning whether or not to apply instruction scheduling. Using the induced heuristic, they were able to reduce scheduling effort but were unable to reduce the total execution time for the SPECjvm8 benchmark suite.

Stephenson et al. [2003] presented the work most similar to ours. They used GP to tune heuristic priority functions for three compiler optimisations. In contrast, we use GP to search over the *feature* space rather than the *model* space. They achieved significant improvements for hyperblock selection and data prefetching within the Trimaran's IMPACT compiler.

In the area of loop unrolling, Monsifrot et al. [2002] use a classifier based on decision tree learning to determine which loops to unroll. They looked at the performance of compiling Fortran programs from the SPEC benchmark suite using g77 for two different architectures, an UltraSPARC, and an IA64. They showed an improvement over the hand-tuned heuristic of 3% and 2.7% over g77's unrolling strategy on the IA64 and UltraSPARC, respectively. Stephenson and Amarasinghe [2005] went beyond

Monsifrot, predicting the actual unroll factor within the Open Research Compiler. Using SVMs, they show an average 5% improvement over the default heuristic.

In contrast to our work, these techniques used hand-selected features, the quality of which was not explicitly evaluated. To the best of our knowledge, we are the first to search and automatically generate features for machine learning.

The grammar system allows expressions and programs that are used for features to be searched over. Other systems allow searching over programs as well. Perhaps the most famous is GP [Koza 1990]. GP builds an expression tree where each node must have the same return type. This is very restrictive, leading to difficulties when there are different types in the system, such as booleans, floats, and integers. GP also suffers from serious problems extending to anything other than expression trees; the definitions of functions and loops, for example, have to be handled as special cases, which can quickly become unwieldy. GP does, however, offer locality for changes to the expression trees; a modification in one part of the tree affects only that part of the tree, not others. Locality is considered a very desirable quality in evolutionary search because it permits small changes to the genotype to manifest in only small changes to the phenotype.

Most similar to our approach is Grammatical Evolution (GE) [Ryan et al. 1998]. Here, a grammar is used, giving the same flexibility as we do and also avoiding the problems of GP. However, because the GE codon stream is read sequentially, changes to it that occur during search destroy locality. The ideal would be to have a change in any node affect only the subtree rooted at that node. This is not the case in GE; a change early in the codon stream can make every other node subsequently generated be completely different. This causes problems for GE searches, since the majority of mutations cause massive damage and the search degenerates into a random search (which is not the case with GP). Our system, based on choice trees, suffers no such problems. Any change to a subtree in the choice tree modifies only the corresponding subtree in the resulting parse tree. In this way, our system combines the good searchability of GP with the expressive power of GE.

## 12. CONCLUSION AND FURTHER WORK

We have presented a new technique to automatically generate good features for machine learning–based optimising compilation. By automatically deriving a feature grammar from the internal representation of the compiler, we can search a feature space using GP.

We have applied this generic technique to automatically learn good features for loop unrolling within GCC. Our technique automatically finds features able to achieve, on average, 76% of the maximum available speedup, outperforming features that previously were manually generated by compiler experts.

Our approach focusses on the RTL representation of the loop. Our system is generic, however, and is easily extended to cover different data structures within any compiler. Future works will investigate exploring different feature spaces for new optimisations.

## REFERENCES

Felix Agakov, Edwin Bonilla, John Cavazos, Bjorn Franke, Grigori Fursin, Michael F. P. O'Boyle, John Thomson, Marc Toussaint, and Christopher K. I. Williams. 2006. Using machine learning to focus iterative optimization. In *CGO'06: Proceedings of the International Symposium on Code Generation and Optimization*. IEEE Computer Society, Washington, DC, 295–305. http://dx.doi.org/10.1109/CGO.2006.37

Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. 1986. *Compilers: Principles, Techniques, and Tools*. Boston, MA: Addison-Wesley Longman.

Thomas Back. 1996. *Evolutionary Algorithms in Theory and Practice: Evolution Strategies, Evolutionary Programming, Genetic Algorithms*. Oxford University Press, Oxford, UK.

Steven J. Beaty. 1991. Genetic algorithms and instruction scheduling. In *MICRO 24: Proceedings of the 24th Annual International Symposium on Microarchitecture*. ACM Press, New York, NY, 206–211. http://dx.doi.org/10.1145/123465.123507

John Cavazos and J. Eliot B. Moss. 2004. Inducing heuristics to decide whether to schedule. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation (PLDI'04)*. ACM, New York, NY, 183–194. http://dx.doi.org/10.1145/996841.996864

John Cavazos and Michael F. P. O'Boyle. 2006. Method-specific dynamic compilation using logistic regression. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'06)*. ACM, New York, NY, 229–240. http://dx.doi.org/10.1145/1167473.1167492

Keith D. Cooper, Alexander Grosul, Timothy J. Harvey, Steven Reeves, Devika Subramanian, Linda Torczon, and Todd Waterman. 2005. ACME: Adaptive compilation made efficient. In *Proceedings of the 2005 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'05)*. ACM, New York, NY, 69–77. http://dx.doi.org/10.1145/1065910.1065921

Keith D. Cooper, Philip J. Schielke, and Devika Subramanian. 1999. Optimizing for reduced code space using genetic algorithms. In *Proceedings of the ACM SIGPLAN 1999 Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES'99)*. ACM, New York, NY, 1–9.

Keith D. Cooper, Devika Subramanian, and Linda Torczon. 2002. Adaptive optimizing compilers for the 21st century. *Journal of Supercomputing* 23, 1, 7–22.

Christophe Dubach, Timothy M. Jones, Edwin V. Bonilla, Grigori Fursin, and Michael F. P. O'Boyle. 2009. Portable compiler optimisation across embedded programs and microarchitectures using machine learning. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture (Micro-42)*. ACM, 78–88. http://dx.doi.org/10.1145/1669112.1669124

Ron Kohavi and George H. John. 1997. Wrappers for feature subset selection. *Artificial Intelligence* 97, 1–2, 273–324. citeseer.ist.psu.edu/article/kohavi97wrappers.html.

John R. Koza. 1990. Evolution and co-evolution of computer programs to control independent-acting agents. In *From Animals to Animats: Proceedings of the First International Conference on Simulation of Adaptive Behavior, 24-28, September 1990*, Jean-Arcady Meyer and Stewart W. Wilson (Eds.). MIT Press, Paris, France, 366–375. http://citeseer.ist.psu.edu/koza91evolution.html.

Prasad Kulkarni, Stephen Hines, Jason Hiser, David Whalley, Jack Davidson, and Douglas Jones. 2004. Fast searches for effective optimization phase sequences. In *PLDI'04: Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*. ACM, New York, NY, 171–182. http://dx.doi.org/10.1145/996841.996863

Hugh Leather, Edwin Bonilla, and Michael O'Boyle. 2009. Automatic feature generation for machine learning based optimizing compilation. In *CGO'09: Proceedings of the International Symposium on Code Generation and Optimization*.

Amy McGovern and Eliot Moss. 1999. Scheduling straight-line code using reinforcement learning and rollouts. In *Proceedings of the 1998 Conference on Advances in Neural Information Processing Systems II*. MIT Press, Cambridge, MA, 903–909. http://dl.acm.org/citation.cfm?id=340534.340836

Antoine Monsifrot, Francois Bodin, and Rene Quiniou. 2002. A machine learning approach to automatic production of compiler heuristics. In *Proceedings of the 10th International Conference on Artificial Intelligence: Methodology, Systems, and Applications (AIMSA'02)*. Springer-Verlag, London, UK, 41–50. http://dl.acm.org/citation.cfm?id=646053.677574

Zhelong Pan and Rudolf Eigenmann. 2006. Fast and effective orchestration of compiler optimizations for automatic performance tuning. In *CGO'06: Proceedings of the International Symposium on Code Generation and Optimization*. IEEE Computer Society, Washington, DC, 319–332. http://dx.doi.org/10.1109/CGO.2006.38

J. Ross Quinlan. 1993. *C4.5: Programs for Machine Learning*. Morgan Kaufmann, San Francisco, CA.

Conor Ryan, J. J. Collins, and Michael O Neill. 1998. Grammatical evolution: Evolving programs for an arbitrary language. In *Proceedings of the 1st European Workshop on Genetic Programming*, Wolfgang Banzhaf, Riccardo Poli, Marc Schoenauer, and Terence C. Fogarty (Eds.), Vol. 1391. Springer-Verlag, Paris, 83–95. http://citeseer.ist.psu.edu/ryan98grammatical.html.

Bernhard Schölkopf and Alexandra J. Smola. 2001. *Learning with Kernels: Support Vector Machines Regularization, Optimization, and Beyond*. MIT Press, Cambridge, MA.

Mark Stephenson and Saman Amarasinghe. 2005. Predicting unroll factors using supervised classification. In *CGO'05: Proceedings of the International Symposium on Code Generation and Optimization*. IEEE Computer Society, Washington, DC, 123–134. http://dx.doi.org/10.1109/CGO.2005.29

Mark Stephenson, Saman Amarasinghe, Martin Martin, and Una-May O'Reilly. 2003. Meta optimization: Improving compiler heuristics with machine learning. In *Proceedings of the ACM SIGPLAN 2003*

*Conference on Programming Language Design and Implementation (PLDI'03)*. ACM, New York, NY, 77–90. http://dx.doi.org/10.1145/781131.781141

Spyridon Triantafyllis, Manish Vachharajani, Neil Vachharajani, and David I. August. 2003. Compiler optimization-space exploration. - *IEEE Journal of Instruction-Level Parallelism* 7, 204–215.

Zheng Wang and Michael F. P. O'Boyle. 2009. Mapping parallelism to multi-cores: A machine learning based approach. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'09)*. ACM, New York, NY, 75–84. http://dx.doi.org/10.1145/1504176.1504189

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. In *PLDI'03: Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*. ACM, New York, NY, 63–76. http://dx.doi.org/10.1145/781131.781140

Received February 2007; revised March 2009; accepted June 2009