Automatic Feature Generation for Setting Compilers Heuristics

Hugh Leather[1], Elad Yom-Tov[2], Mircea Namolaru[2] and Ari Freund[2]

[1] Institute for Computing Systems Architecture
School of Informatics
University of Edinburgh
Edinburgh EH9 3JZ
United Kingdom
hughleat@hotmail.com
[2] IBM Haifa Research Lab
Haifa 31905
Israel
{yomtov,mircea,arief}@il.ibm.com

1 abstract

Heuristics in compilers are often designed by manually analyzing sample programs. Recent advances have successfully applied machine learning to automatically generate heuristics. The typical format of these approaches reduces the input loops, functions or programs to a finite vector of features. A machine learning algorithm then learns a mapping from these features to the desired heuristic parameters. Choosing the right features is important and requires expert knowledge since no machine learning tool will work well with poorly chosen features.

This paper introduces a novel mechanism to generate features. Grammars describing languages of features are defined and from these grammars sentences are randomly produced. The features are then evaluated over input data and computed values are given to machine learning tools.

We propose the construction of domain specific feature languages for different purposes in different parts of the compiler. Using these feature languages, complex, machine generated features are extracted from program code. Using our observation that some functions can benefit from setting different compiler options, while others cannot, we demonstrate the use of a decision tree classifier to automatically identify the former using the automatically generated features.

We show that our method outperform human generated features on problems of loop unrolling and phase ordering, achieving a statistically significant decrease in run-time compared to programs compiled using GCC's heuristics.

2 Introduction

A recent trend in compilers has seen the application of machine learning to automatically construct heuristics[4,10,3]. Previously, heuristics needed manual tuning by compiler experts, at not inconsiderable effort. These endeavours may need to be repeated

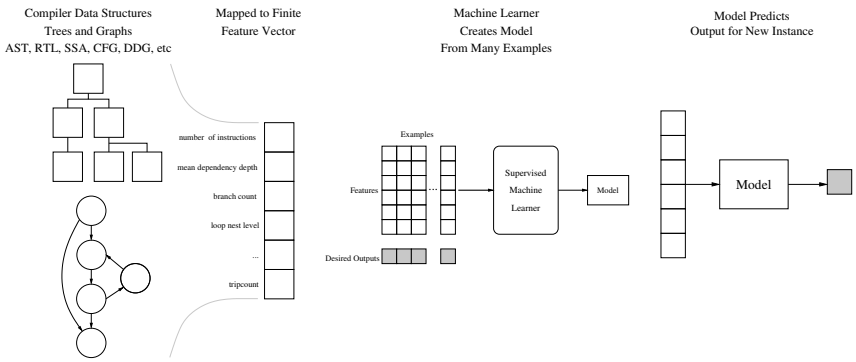


Fig. 1. Generic view of machine learning in compilers

when a new architecture is to be supported or changes elsewhere in the compiler cause degradation in the heuristic's performance or when new benchmarks are selected. In today's market of rapidly evolving architectures, the time and exertion required to achieve an acceptable optimising compiler is becoming a serious issue[12]. By contrast, the promise of machine learning approaches is to offer low cost repeatability. The ultimate goal being that a compilers heuristics can be retrained for a new environment at the press of a button. Moreover, in practice, machine learned heuristics often outperform their human created counterparts[6,11].

In the generic, simplified, view of supervised machine learning, a number of examples of a desired function are presented. Each example combines a particular input vector with the corresponding optimal output vector. The machine learning tool will then construct a model, mapping input vectors to output vectors. Frequently, the input vectors will be fixed in length and contain real numbers or discreet categories. These vectors are thought of as features of input instances.

The typical case of supervised machine learning in compilers is exemplified, and again simplified, here. The engineer identifies some heuristic to replace with a machine learned version. For example, this might be the priority function for scheduling instructions[8], in which case the output would be the relative priority for each instruction. The engineer must decide what features will be presented in the input vectors. These may include the number of instructions of different types, dependency depths, critcal path lengths and so on. Next, examples are acquired by running a number of benchmarks, collecting input features and recording the best value for the desired heuristics. Finally, some machine learning tool will be asked to construct a model over these examples and the engineer will hope that this model is better than previous work. A diagramatic representation of this process is shown in Figure 1.

While expert compiler knowledge is no longer required to develop heuristics once features are decided upon, an expert must still develop appropriate features. Obviously, every machine learning tool is bound by the performance of the input features it receives.

Programs and the derived data structures created for them inside of a compiler come from infinitely dimensioned spaces. At any point in the compilation process there will likely be a large amount of structured data describing the current state of the program. The compiler maintains list, trees and graphs of complex, interelated data. Reducing this wealth of data down to a small summary vector can be daunting and difficult. Vast amounts of information will no longer be available to the machine learner and it is hard for the compiler engineer to be sure that the best imformation is still present. The engineer must be concerned with selecting the best mapping, the best features from an infinite set of possibilities.

Suppose, for example, the engineer has some intuition that the number of addition nodes might be a crucial feature. Perhaps also, the number of addition nodes whose first child is a constant is also important; or perhaps those addition nodes followed by a multiplication would be interesting for the machine learner. As the complexity of the admissible features increases slightly the number of them explodes. The situation becomes more pronounced when we consider attribute values, graph structures and so on.

This paper shows a promising technique for automatically exploring this massive feature universe.

We target two types of optimizations, loop unrolling and phase ordering, to demonstrate our approach.

Loop unrolling is an optimization performed by practically every modern compiler. The bodies of loops are replicated to improve instruction level parallelism, assist instruction scheduling and help other optimizations. Unrolling is not always beneficial, however, as when done aggressively too much register pressure can be created, the instruction cache can become overfull and other interactions come into play, outweighing the gains. Selecting the right number of times to unroll a loop is a non trivial problem and results will likely be different for different architectures.

Decision trees are used in [2] to learn unroll factors. In that work, the authors found that their chosen features were unable to distinguish unroll factors, i.e. that after reducing loops to their finite feature vectors, loops with opposite classifications would have mapped to the same feature vector. Nearest neighbour algorithms and support vector machines have been used in [10] to predict unroll factors using human crafted features.

Phase ordering involves finding the optimal ordering of phases or passes in the compiler. Compiler writters have long known that a single, fixed sequence of optimization phases will not be optimal for all programs or even individual functions within a program[13].

Finding a good sequence of optimization phases has been solved by searching a space of iteration compilations[5]. Machine learning has been applied [1] to reduce the cost of iterative compilation.

3 Automatic Feature Extraction from Compiler Grammar

In this section we describe how we extract features from the structured data available in the compiler. We design feature languages as subsets of some broader programming languages. The feature languages describe a wide range of different feature expressions.

Listing 1.1. Simple feature grammar

```
<feature> ::= "count-nodes-matching(" <matches> ")"
<matches> ::= "is-constant"
            | "is-variable"
            | "is-any-type"
            | ( "is-plus" | "is-times" )
              ( "&&" "left-child-matches(" <matches> ")" )?
              ( "&&" "right-child-matches(" <matches> ")" )?
```

We then generate random sentences from the language, each becoming a feature. The feature is then computed over the input instances, for example, loops for loop unrolling. The computed feature values are then passed to a machine learning algorithm together with the desired outcome for each instance. The machine learner builds a model to predict the right outcome and this replaces the heuristic in the compiler.

3.1 A Simple Feature Grammar

To illustrate our approach we describe generating features for the AST of a tiny, toy language before extending to our work in GCC. The language will allow only sets of assignment statements; the left hand side of each will be a variable name; the right will be an expression containing variables, constant integers, operators '+' and '*' and parentheses. We are not concerned with how to parse this language, therefore we assume that the ambiguity in operator precedence has been suitably dealt with. We are interested only in the intermediate parse trees.

Example statements from the language are:

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

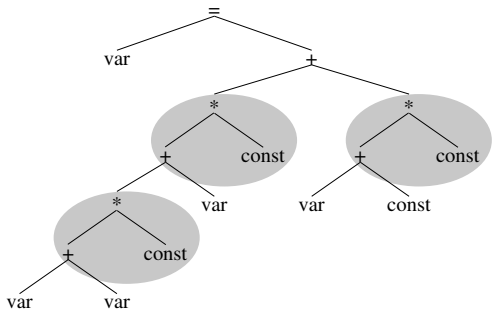
We wish to describe features over these statements. Very simple features might be to count the number of constants or the number of each different operator. There are however, an infinite number of potential features. We might have a feature counting the number of '*' nodes whose left child is a '+' and whose right child is a constant. Listing 1.1 shows a simple grammar producing an infinite set of such features in a pseudo-code style. The sentences from the grammar should be immediately understandable and it should be clear how to convert the grammar to execute on any ordinary programming language.

In order to generate a sentence from the grammar we simply expand non-terminals until only terminals remain. At each point in the expansion, where choices can be made, we select from among the choices at random.

The grammars can be arbitrarily complicated. While the grammar above creates statements counting matching substructures, more complex alternatives can easily be devised, indeed the grammars we use in practice are much more complex. Matching criteria can use disjunction as well, preventing overspecialisation. Other summaries (averages, min, max, etc.) can be used at different levels of recursion. Attributes of nodes

Fig. 2. An example feature from the grammar in Listing 1.1. In the right hand of the figure is a sample AST from the tiny language, showing the matching sub structures. The feature thus evaluates to 3.

```
count-nodes-matching(
    is-times &&
    left-child-matches(
        is-plus
    )&&
    right-child-matches(
        is-constant
    )
)
```



can be used in matching tests or combined into feature results. Graph like structures can be traversed allowing features to range over call graphs, data dependency graphs, etc.. The set of potential features is limited only by the data made available to the system, the power of the language the features are implemented in and the inventiveness of the grammar.

3.2 Problems of Recursion

Whilst ambiguities cause problems in parsing sentences from grammars, sentence production suffers from infinite recursion. Perhaps the simplest example of this is in the grammar in Figure 1.2 which obviously produces an unending string of 'a's. Attempting to generate a sentence from this grammar will lead to some discomfort. Our 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. However, in practice such grammars are unlikely to be seen.

More subtle recursion issues are caused probabilistically. Consider the grammar in Listing 1.3. The language consists of odd numbers of consecutive 'a's. However, if the two productions are chosen from uniformly at random, then we are likely to get very long strings. The probability that a string of n non-terminals, $AAA...A$, will contain fewer non-terminals after each is expanded once is given by $I_{1/2}(2n/3, n/3 + 1)$, where I is the regularized incomplete beta function. As strings contain more non terminals they become increasingly likely to grow at each expansion. Our grammar definition language allows productions to be weighted to avoid explosive sentence lengths. Deciding the appropriate weights for productions can be painful to do analytically. We have

Listing 1.2. Infinitely recursive grammar

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

Listing 1.3. Simple feature grammar

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

found, however, that trial and error produces acceptable results with very little time or effort.

3.3 Generating Features in GCC

Our experiments focus on data structures available in GCC, version 4.2. We export data - either tree-SSA or RTL - as XML trees, a convenient representation to work with offline. Our features are a subset of XQuery, a language designed to make queries about XML documents. Grammars are produced which define a large variety of XQuery expressions pertinent to the XML representations of the data structures.

Our system is in fact quite indifferent to the data language and feature processing language. Any combination could be chosen and we have demonstrated this by having our grammars produce Java source code as features and our data being Java objects instead. However, for features targeted at tree structures, the XML and XQuery pairing is particularly concise and easy to debug.

We begin by extracting data structures of interest from GCC as XML.

Extracting Data Structures For our experiment on loop unrolling, the data include various loop characteristics, each basic block in the loop and, for each of those, a structured dump of the RTL code for the statements in the basic block. A snippet of such a file is presented in Listing 1.4. Many of the attributes and values made available are removed for clarity. 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 multiplication of two others.

For our experiments on phase reordering we dump the AST of each function, in a similar manner, to XML. The AST is dumped after conversion to SSA format and contains details of each basic block in the function, predecessors and phi nodes in each basic block and the AST tree for each statement in the block.

Determining Observable Structure The hierarchical data produced in the previous step 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 foolish to create a feature like "*count the number of basic blocks which contain three loops*" since that can never be non-zero.

The grammars we construct are automatically derived from the structural rules of the data to ensure that such patently silly features are never generated. Since the rules that tree-SSA and RTL follow are not immediately derivable in a machine readable manner from the GCC source code, we take the simpler approach of examining the XML data files to find the observed structure within.

Listing 1.4. XML representation of RTL

```
<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 volatile="true" mode="SF">
            <int>84</int>
            ..
          </reg>
        </mult>
      </set>
    </insn>
```

We iterate through the XML data files and collect the number of each type of node in the XML; how many children each node type has; a histogram of child types for each child slot in each node type and a histogram of attribute values for each node type.

This structural information is then used to mechanically create a grammar by passing it through an XSL transform. The transforms, then, define families of grammars of different powers and biases. Some transforms only produce grammars similar to the one in Listing 1.1, while others will make use of the observed structure in more complex ways. The transforms we use in practice all make sure that trivially impossible features (or rather, uninteresting features which do not relate to possible structures) are never created. They also automatically set production weights to ensure grammars do not suffer from the probabilistic recursion issues described in subsection 3.2

Loop Unrolling Grammar and Features The BNFs for the grammars, being machine generated, are quite large; several tens of kilobytes. The flavour of the rules is simple, though, and is described below.

In the case of our loop unrolling experiment each feature is simply a count of the number of nodes at any depth in the tree matching certain criteria. These criteria include the name of the node, values of different attributes and may recursively impose similar criteria upon children of the node. Grammar productions are weighted to prevent explosive recursion.

The grammar, due to a biased weighting, produces many simple features testing first order structures in the data, such as a human engineer might be likely to. Therefore, all of the standard features one might expect, such as counting the number of nodes of every type will be present after only a few thousand features are generated. However, the grammar is not wholly biased towards short features; many are of medium length, such

Listing 1.5. Example RTL features

```
// Count the number of 'plus' nodes whose first child is a 'reg'
// node with mode = 'DI' and whose second child is also a reg node
count(
    descendant-or-self::*[name()="plus"]
        [*[1][name()="reg"][@mode="DI"]]
        [*[2][name()="reg"]]
)

// Count the number of 'compare' nodes with mode = 'CCZ' whose
// first child is a 'mem' node with the 'frame-related' flag set
// but not the 'return-val' flag.  The second child of that is a
// 'component_ref' node which has a null pointer as it's first child
count(
    descendant-or-self::*[name()="compare"][@mode="CCZ"]
        [*[1][name()="mem"][@frame-related="true"][not(@return-val)]
            [*[2][name()="component_ref"]
                [*[1][name()="null"]]]]
)

// Count the number of loop nodes whose average number of
// instructions is less than 54 and which has no assumptions.
// Since there is only one loop per loop, this will be one
// if the loop matches or zero if not
count(
    descendant-or-self::*[name()="loop"]
        [number(@avg-insns) lt 54]
        [@has-assumptions="false"]
)
```

as those in Listing 1.5 and there are also a good number of longer, complex features, too detailed to show here.

Phase Ordering Grammar and Features The grammar for the phase ordering is slightly more powerful. It contains several classes of feature.

The first computes summary statistics, $min, max, mean$ and sum, over the basic blocks in the function, of the number of statements matching some criterion. Statements match according to similar structural constraints as used for the RTL code, with the addition that sibling statements may be required match similar criteria. Such features allow small sequences of statements to be considered.

The next feature class computes summaries, with the same set of aggregation functions, again over each basic block. However, this time the number of substructures present at any depth in the statements is determined. Whereas the first class can yield, for example, the number of statements containing a PLUS node with a constant operand, this second class can find the number of PLUS nodes with a constant operand in any statement in the block.

The last two classes consider the phi nodes and predecessor information of each basic block. A matching criterion is created in a similar fashion to previous classes and numbers matching in each basic block are aggregated as before.

Listing 1.6 shows some example features from this grammar.

Listing 1.6. Example AST features

```
// Count the number of 'label_expr's followed by a 'cond_expr'
// whose first child was a 'ne_expr'
sum(bb/stmts/count(*[name()="label_expr"]
     [following-sibling::*[1][name()="cond_expr"]
          [*[1][name()="ne_expr"]]]
))

// Take the average of the number of 'real_cst' nodes
// in each basic block
avg(bb/stmts//count(*[name()="real_cst"]))

// Take the minimum number of predecessors across the basic
// blocks with the 'fallthru' flag set but not the 'exec'
// flag
min(bb/preds/count(*[@fallthru="true"][not(@exec)]))
```

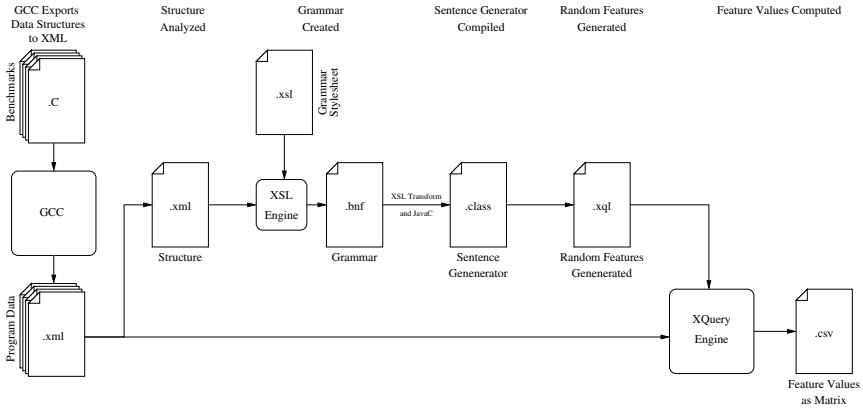


Fig. 3. Schematic of feature generation steps

Feature Generation Once a grammar has been defined, a generator is then compiled to Java code. Each call to the generator returns a random sentence or feature from the grammar. The features, as XQuery expressions, are then evaluated against the XML data from the input instances, yielding an array of feature values for each instance.

Since our search is random, not directed, a feature that has been seen before can randomly generated again. Sometimes features always evaluate to zero because no matching structures are found anywhere in the input (although we use structural information to avoid trivial examples our structural information is only an estimate of the full rule set inherent in GCC's code). More rarely two different features will produce all the same values over the inputs - to a machine learning algorithm they would be indistinguishable. We discard all features that are not unique or that do not differentiate at least one input instance.

These feature values are then used in a more conventional setting, learning a model for the best heuristic as required. We show, in Figure 3, a schematic representation of how the system creates a matrix of feature values from benchmarks.

4 Experimental setup

In order to test the effectiveness of our system we targeted two machine learning problems. The first is loop unrolling and the second is phase ordering.

4.1 Loop Unrolling

We modified GCC version 4.2 to allow the unroll factors for particular loops to be externally specified. Then we took 23 benchmarks from the UTDSP and MediaBench benchmark suites. To find the best unroll factor for each unrollable loop in the programs we took each loop, one at a time, and unrolled it by different factors, zero to sixteen. This gave a compiled program for which all but one loop has the default unroll factor as determined by GCC's default heuristic. 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 extracted 243 unrollable loops from the benchmarks.

These experiments were run on a single unloaded, headless machine; an Intel single core Pentium 4 running at 2.8 GHz with 512 Mb of RAM. All files for the benchmarks were transferred to a 32 Mb RAM disk to reduce IO variability. For each version of the benchmarks we ran one thousand executions of the program, more than insuring the error in the measurements would be small.

The best unroll factor for each loop was determined as that with the lowest mean cycle count. The average cycle counts were computed after trimming the outlying 10% of the data.

4.2 Phase Ordering

We modified GCC version 4.2 to perform feature extraction and transformation order selection for the following six transformations:

1. Constant propagation
2. Partial redundancy elimination
3. Dead code elimination
4. Forward propagation
5. Copy propagation
6. Merge phi node

Since there were 6 possible transformations, there could be 1956 permutation of every non-empty subset of the six transformations. These are known as variations.

We conducted the experiments on the 11 following SPEC 2000 integer benchmarks: `ammp`, `art`, `bzip2`, `crafty`, `equake`, `gap`, `gzip`, `mcf`, `mesa`, `parser`, `twolf`. We compiled each benchmark 1956 times, once each for every possible permutation of every non-empty subset of the six passes. During compilation we gathered the feature descriptions produced by the compiler. We then ran each compiled image and measured the amount of time it spent in each of its functions. We performed the time measurements on a SUSE 2.6 Linux PowerPC platform featuring a 4-core, 8-thread Power 5

processor running at 1.5GHz. (We only used a single thread for the executions in order to get accurate and stable readings.) We performed the time measurements using the *oprofile* tool by running each binary image five times. Thus, for each function in each of the benchmarks, for variation, we obtained a single sample point consisting of the feature descriptions provided by the compiler for this function and pass selection (and order), and the average time spent in the function during 5 consecutive executions.

We applied machine learning only to functions which had run-times longer than 0.1sec. This was done so as to leverage the learning algorithms only to functions which would have a large effect on run times. A total of 326 functions were found to have such run-times.

5 Learning compiler options

Given the limited amount of training data, we attempted to reduce the size of the output space for the learning algorithm. As described in the previous section, there are 243 loops for loop unrolling for which the output space is comprised of 17 possible settings (unrolling factors of zero through 16) and 326 functions for phase ordering with 1956 possible variation settings.

Thus, we attempted to substantially reduce the number of possible decisions. This was done by observing that the possible speedup of the functions or loops as a function of compiler options is not independent. Our observations were that certain functions and loops (hereafter 'instances') are more amenable to optimization than others.

In both experimental settings, we used the well-known k-means algorithm[7] to cluster the instances according to their speedup compared to run-time according to GCC's decision. That is, each instance was represented by the speedup factor it attained for each possible compiler option.

Figure 4 shows the average speedup obtained by each class resulting from the k-means algorithm. As can be seen, there are some instances for which no compiler setting will improve speedup over GCC (on average), indeed, for some classes it will only degrade performance. However, there are instances where, on average, almost any compiler setting other than the one chosen by GCC will improve performance.

Therefore, we trained the learning algorithm to identify, based on the features, to which class an instance belongs. We used the C4.5 decision-tree algorithm [9] as the classifier. The GCC heuristic was used for those instances which were predicted as belonging to the class of instances which would achieve the lowest run-time using the GCC heuristic. For other instances we used the compiler setting (loop unrolling factor or variation) which gave the maximal speedup on training data for that class of instances.

In all our experiments we trained the classifier using the leave-one-out method, that is, for each of the instances in the data we trained a classifier using all other data and applied it to the instance. Decision trees shown in the results section were trained over the whole data.

In summary, the steps required to decide on the correct parameter setting using our method are as follows:

1. GCC reports data structures from benchmarks as XML

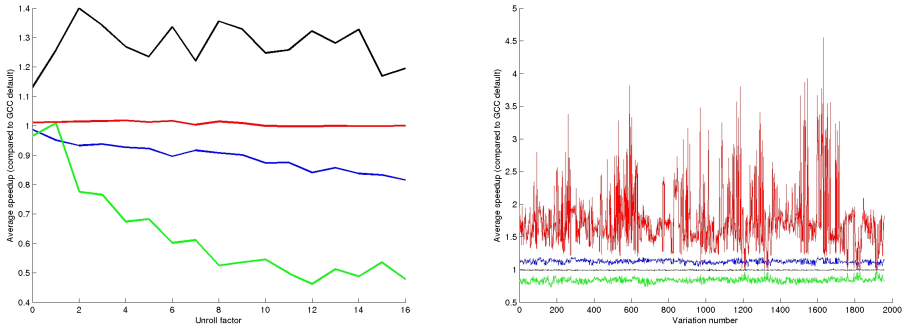


Fig. 4. Average speedup (compared to GCC) of instance, clustered according to speedup using different optimization settings. Each color represents the average speedup of the instances in the class, compared to GCC's heuristic. The left figure is for loop unrolling data and the right figure for transformation ordering selection.

2. Exhaustive iterative compilation determines best compilation strategy for training set
3. Structural relationships determined over training set
4. Grammar created from structure
5. Sentence (feature) generator compiled from the grammar
6. Features randomly generated
7. Features evaluated against XML data structures for training set and test set
8. Features values passed to machine learning tool to build model
9. Model evaluated against test set

6 Experimental Results

In order to test the effectiveness of our system we targeted two machine learning problems. The first is loop unrolling and the second is phase ordering. We show the improvements against GCC's default heuristics and give the decision trees and features found in the experiments.

6.1 Loop Unrolling

Training data was collected as described in Section 4.1. After predicting the loop unrolling factor for each function using the learning algorithm, each program was run again. We measured the total run-time of each benchmark under three settings: GCC's default heuristic, the predicted unroll factor, and using the best possible unrolling factor (as determined directly by the training data).

An average speedup (compared to GCC's heuristic) of 1.8% was obtained using the learned unrolling factors. The best speedup, obtained when using the unrolling factors that gave the shortest run-time for each loop during training, was 5.3%.

Figure 5 shows the top levels of the learned decision tree and the features on which it depends.

6.2 Phase Ordering

Training data was collected as described in Section 4.2. After predicting the variation number for each function using the learning algorithm, we computed the average speedup for each function.

An average speedup of 4.0% was obtained using the learned variation numbers. This is a statistically significant improvement (sign rank test, $p < 10^{-3}$). The best speedup, obtained when using the variation numbers that gave the shortest run-time for each function during training, was 36.4%. Alternatively, assuming that the best variation number is that which gives the minimal average run-time for all functions in the group, speedup is 4.3%, which is not very different from that using the learning algorithm. Therefore, we conclude that while the idea of selecting functions according to their group behavior gives a statistically significant speedup over the GCC heuristics, the potential is much greater to correctly identify the exact compiler setting that would be suitable for each function.

Figure 6 shows the top levels of the learned decision tree and the features on which it depends.

7 Conclusion

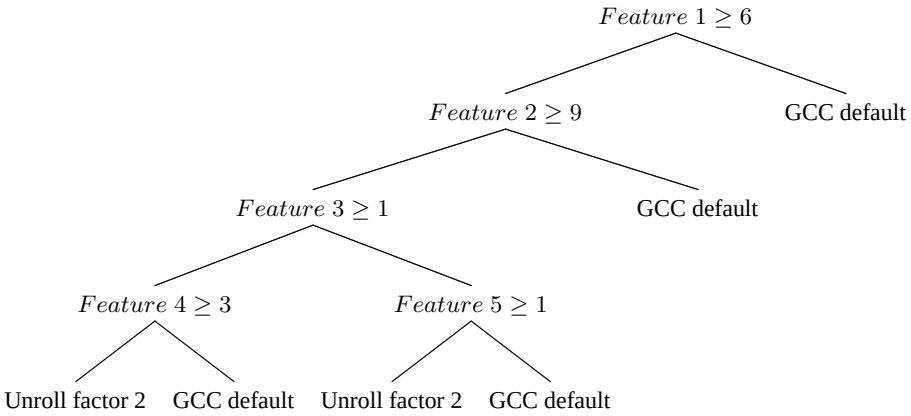
Applying machine learning techniques to compilers has heretofore required experts to generate features. At no point could the expert be sure that they had the optimal set of features to assist their machine learner.

We have demonstrated a promising system to automatically generate features for compiler allowing the exploration of an infinite variety of features. We have shown that the system outperforms existing heuristics in a modern compiler.

An obvious question is to ask why those particular features are chosen as the most important. This is difficult to answer. It is, perhaps, one of the strengths of our approach that one does not need to expert knowledge of the compiler, the architecture, and the program, in order to make full use of the system. We expect, however, that the most important features will change as we make available more of the compiler's data structures to the system.

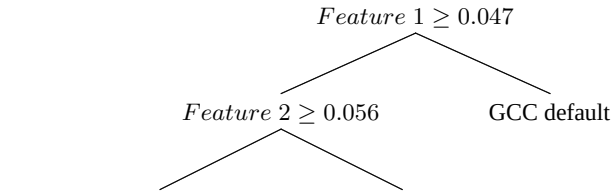
We envisage, in the future, a number of grammars being available to researchers. Each will be appropriate at different stages of the compilation process and will allow the creation of features able to make use of all the data currently in the compiler. We foresee these grammars being used to automatically determine complex features far beyond those that can be manually constructed.

Our current systems are limited by the data made available to them. For example, data dependency graphs, control flow graphs and other important data structures have not been included. We believe that significantly better results will be achieved when the majority of the compiler's data structures are exposed to the automatic feature generators. Indeed, we expect to attain superior performance over not just GCC but other state




```
Feature1 :
    // Count the number of "plus" nodes whose first operand is a "reg"
    // node with both "volatile" and "frame-related" flags set
    count(
      descendant-or-self::*[name()="plus"]
        [*[1][name()="reg"][@volatile="true"][@frame-related="true"]]
    )
Feature2 :
    // Count the number of "set" nodes whose source operand is
    // "mult" nodes for which the first operand is a "reg" node
    count(
      descendant-or-self::*[name()="set"]
        [*[2][name()="mult"]
          [*[1][name()="reg"]]
        ]
    )
Feature3 :
    // Count "compare" nodes whose first child is a "reg" node
    // which has its "frame-related" flag set and is has "mode=SI"
    count(
      descendant-or-self::*[name()="compare"]
        [*[1][name()="reg"][@frame-related="true"][@mode="SI"]]
    )
Feature4 :
    // Count instructions that set a register to the value of a
    // comparison against a register
    count(
      descendant-or-self::*[name()="insn"]
        [*[6][name()="set"]
          [*[1][name()="reg"]]
          [*[2][name()="compare"]
            [*[2][name()="reg"]]
          ]
        ]
    )
Feature5 :
    // Count the number of basic-blocks that might be hot and
    // have estimated frequency less than 5426.  They must
    // also contain a call instruction whose last field is null.
    count(
      descendant-or-self::*[name()="basic-block"]
        [@may-be-hot="true"][number(@frequency) lt 5426]
        [*[name()="call_insn"]
          [*[10][name()="null"]]
        ]
    )
```

Fig. 5. Top levels of the decision tree built for loop unrolling




```
Feature1 :
    // Count the number of "indirect_ref"s of "pointer_type" anywhere
    // in each basic block and then take the average across all basic
    // blocks in the function.
    avg(bb/stmts//count(*[name()="indirect_ref"][@type="pointer_type"]))
Feature2 :
    // Count "cond_expr"s anywhere in each basic block whose condition
    // is a greater than expressions whose second operand is a constant,
    // invariant address expression
    avg(
      bb/stmts//count(*[name()="cond_expr"]
        [*[1][name()="gt_expr"][@type="boolean_type"]
          [*[2][name()="addr_expr"][@invariant="true"][@constant="true"]]
        ]
      )
    )
```

Fig. 6. Top levels of the decision tree built for transformation ordering

of the art machine learning techniques once we have sufficient data that our features can be a strict super set of those in existing works. We will investigate this in the future.

Presently, features are randomly generated from the grammars. There are no objective functions specified to search against. We will be evaluating different objective functions, such as mutual information score and wrapper techniques together with different search techniques. The parse trees containing the choices for any particular feature expression can be mutated to allow gradient descent and simulated annealing and mated to allow genetic techniques to be used.

The grammars we have used to date have not made use of the full potential of the system. They have permitted our features to make statements about the substructures in the RTL and tree-SSA but they have not, for example, allowed features to follow the graph like pointers available from variable names, control flow, etc.. We shall be considering more powerful grammars in the future.

While the examples presented in this paper have all used XML and XQuery, the tools do not require it. Data have also been presented as Java objects and features have been constructed as Java source code. We intend to use other languages as the desired features become more complicated since while XML and XQuery are excellent choices for making statements about trees, they perform less well for more general graphs structures.

References

1. Felix Agakov, Edwin Bonilla, John Cavazos, Bjoern Franke, Grigori Fursin, Michael F.P. O'Boyle, John Thomson, Marc Toussaint, and Christopher K.I. Williams. Using machine learning to focus iterative optimization. pages 295–305, 3 2006.
2. Rene Quiniou Antoine Monsifrot, Francois Bodin. A machine learning approach to automatic production of compiler heuristics, 2002.
3. Steven J. Beaty. Genetic algorithms and instruction scheduling. In *MICRO 24: Proceedings of the 24th annual international symposium on Microarchitecture*, pages 206–211, New York, NY, USA, 1991. ACM Press.
4. John Cavazos and Michael F. P. O'Boyle. Method-specific dynamic compilation using logistic regression. *SIGPLAN Not.*, 41(10):229–240, 2006.
5. Keith D. Cooper, Philip J. Schielke, and Devika Subramanian. Optimizing for reduced code space using genetic algorithms. In *Workshop on Languages, Compilers, and Tools for Embedded Systems*, pages 1–9, 1999.
6. Christophe Dubach, John Cavazos, Björn Franke, Grigori Fursin, Michael F.P. O'Boyle, and Olivier Temam. Fast compiler optimisation evaluation using code-feature based performance prediction. In *CF '07: Proceedings of the 4th international conference on Computing frontiers*, pages 131–142, New York, NY, USA, 2007. ACM Press.
7. Richard O. Duda, Peter E. Hart, and David G. Stork. *Pattern classification*. John Wiley and Sons, Inc, New-York, USA, 2001.
8. Eliot Moss, Paul Utgoff, John Cavazos, Doina Precup, Darko Stefanović, Carla Brodley, and David Scheeff". Learning to schedule straight-line code. In *Advances in Neural Information Processing Systems*, volume 10. The MIT Press, 1998.
9. John Ross Quinlan. *C4.5: Programs for machine learning*. Morgan Kaufmann, San Francisco, CA, USA, 1993.
10. Mark Stephenson and Saman Amarasinghe. Predicting unroll factors using supervised classification. In *CGO '05: Proceedings of the international symposium on Code generation and optimization*, pages 123–134, Washington, DC, USA, 2005. IEEE Computer Society.
11. Mark Stephenson, Saman Amarasinghe, Martin Martin, and Una-May O'Reilly. Meta optimization: Improving compiler heuristics with machine learning. 6 2003.
12. Nigel Topham. *Challenges to Automatic Customization*, chapter 8, pages 186–208. Morgan Kaufmann, 2006.
13. Steven R. Vegdahl. Phase coupling and constant generation in an optimizing microcode compiler. In *MICRO 15: Proceedings of the 15th annual workshop on Microprogramming*, pages 125–133, Piscataway, NJ, USA, 1982. IEEE Press.