Automatic Feature Generation for Machine Learning Based Optimizing Compilation

Hugh Leather, Edwin Bonilla, Michael O'Boyle
School of Informatics
University of Edinburgh
Edinburgh, Scotland
hughleat@hotmail.com

Abstract

Recent work has shown that machine learning can automate and in some cases outperform hand crafted compiler optimizations. 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 paper develops a novel mechanism to automatically find those features which 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 modeling. 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, GCC's hard-coded heuristic achieves only 3% of the maximum performance available, while 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.*

I. Introduction

The use of machine learning to automate compiler optimization [1]–[3] 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 optimization and its interaction with the architecture and the rest of the compiler [4]. 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 exertion required to achieve an acceptable optimizing compiler is becoming a serious issue. Machine learned heuristics are attractive in that they not only automatically adapt for a new environment but, in practice, often outperform their human created counterparts [5].

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 [6]. 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 programs are, syntactically, unbounded tree structures and that there are an infinite number of these potential features, this is a non-trivial 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 relationship 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 [7].

Previously, researchers in machine learning for compilers have manually created lists of features they believe reasonable [8]. 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 we represent the space of features as a grammar. Each sentence from the grammar represents one feature. We search this space for good features which 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 methodology. Although genetic programming has been used before to search over the model space [2], this is the first time it has been used to generate features.

We evaluated our technique on an extensively studied problem: loop unrolling [9]. Loop unrolling is an optimization 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 [2] 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

978-0-7695-3576-0/09 $25.00 © 2009 IEEE
DOI 10.1109/CGO.2009.21

unrolling heuristic, on average, obtains just 3% of the maximum performance available. A state of the art [2] 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 paper is organized as follows. Section II shows how machine learning is presently applied for learning compiler heuristics and describes the problems with it. In section II-B we present a motivating example, showing why an approach like ours is needed. This is followed in section III by an overview of our system and section IV explains how feature grammars are used. Our experimental setup, methodology and results are presented in sections V, VI, and VII respectively. This is followed by a summary of related work and some concluding remarks.

II. Current Use of Machine Learning in Compilers

In this section we briefly describe how machine learning is deployed within a compiler and give an example showing that selecting the wrong features leads to poor performance. Consider figure 1 which shows a simplified view of the machine learning process. The compiler writer first selects the optimization, e.g. scheduling [10], to replace with a machine learned version. He examines the state of the compiler just before the heuristic is evaluated and determines what data are at hand (see figure 1(a)). These data may include structures such as the abstract syntax tree and the control flow graph depending on where the heuristic is evaluated in the compilation process.

Standard machine learning tools typically deal with fixed length vectors, so the compiler writer must summarize the data into a fixed length vector (as shown in Figure 1(b)). Elements of these vectors are called *features*. Usually, the compiler writer uses features such as 'the number of instructions' or 'the loop nest level' [2], [9]. In this process, information is inevitably lost and success will depend on how well the features are chosen.

The compiler writer must now generate training data. He starts with a number of benchmarks and calculates the features for each. The benchmarks are compiled multiple times; each time he experiments with a different trial value of the heuristic and, by running the newly compiled program, discovers which heuristic value is best for the given benchmark. The compiler writer passes all of these examples to a supervised machine learning tool and asks it to build a model. The model's job is to predict, for a new set of features, what the optimal heuristic value should be. This entire process is shown in figure 1 (c).

If this model is placed back into the compiler then he has replaced the heuristic, as desired (figure 1(d)). The great advantage of this machine learning approach is that the compiler writer has not had to develop a heuristic and the process can be easily repeated whenever there is a change in the underlying architecture. However, effort has now been transferred from tuning the heuristic by hand to creating the right features for the machine learning algorithm by hand.

A. Problems in Feature Creation by Hand

There are a number of potential problems that the compiler writer must be aware of. In this section we briefly outline some of the problems encountered in our research. *Irrelevant features:* Features such as 'the number of comments in the code' or 'the average length of identifiers' would not help a machine learning algorithm. 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. *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 as shown in [9]. *Classifier peculiarities:* A set of features that performs well for one machine learning algorithm might not be good for another [7]. 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. Constructing features that provide additional predictive power becomes increasingly hard.

B. Motivation

In this section we demonstrate that selecting the right features can have significant impact on optimization performance. Consider the loop in figure 2 selected from the `mesa` benchmark within *MediaBench*. If we apply GCC's default loop unroll heuristic (labeled GCC Default in figure 2 (b)), it determines 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 exhaustively evaluated, 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 we replace GCC's heuristic with a machine learning decision tree algorithm, whose features are 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 we, instead, use our technique to search for the best set of features and train a decision tree over those then we are able to automatically select the best unroll factor of 11, giving the maximum speedup for the loop in figure 2. The

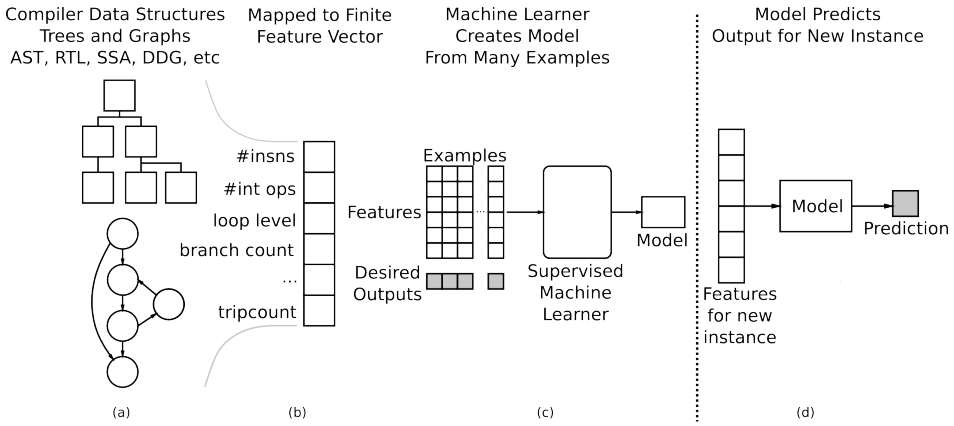

Figure 1. Generic view of machine learning in compilers. In stages (a) to (c) the model is learned from input examples; in stage (d) the model is deployed and predicts a heuristic value for a new program. In stage (a) the compiler writer investigates data structures that may be useful which are then summarized as feature vectors in stage (b). In stage (c) training examples consisting of feature vectors and the correct answer are passed to a machine learning tool. In stage (d) the learned model is inserted in the compiler.

```
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)

Figure 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 2 is selected giving a small improvement. Our technique correctly determines 11 as the best unroll factor.

path of the decision tree selecting the unroll factor of 11 is also shown in figure 4(b) and the features touched 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, we can find features better suited to the learning task at hand.

III. Overview

This section presents a high-level overview of our system, illustrated in figure 5. 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 of the program plus the optimal values for the heuristic we wish to learn. Once these data have been generated, the feature search component explores features over the compiler's intermediate representation (IR) and provides the corresponding feature values to the machine learning tool. The machine learning tool 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.

Data Generation. In a similar way to the existing machine learning techniques (see section II) 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. We also extract from the compiler the internal data structures which describe the programs. Due to the intrinsic variability of the execution times on the target architecture, we run each compiled program several times to reduce susceptibility to noise (see section V).

Feature Search. The feature search component maintains a population of feature expressions. 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 which are later used by the machine learning tool.

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)

Figure 3. The path through the learned GCC tree heuristic (b) for the example in figure 2 and the features used in that path (a).

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)

Figure 4. The path through the learned heuristic (b) for the example in figure 2 and the features from our scheme used by that path (a).

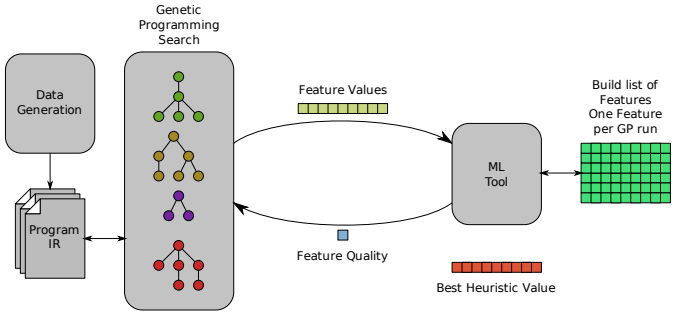

Figure 5. Overview of the system

The grammars and their use are discussed in section IV. The search component, based on genetic programming (GP [11]) and grammatical evolution (GE [12]), is briefly described in the next section.

Machine Learning. The machine learning tool is the part of the system that provides feedback to the search component about how good a feature is. As mentioned above, the system maintains a list of good base features. It repeatedly searches for the best next feature to add to the base features, iteratively building up the list of good features. The final output of the system will be the latest features list.

Figure 6 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 6(a)). We then combine this feature with the previous base features and the target best heuristic values and ask a machine learning algorithm to learn a predictive model over it (shown in figure 6(b)). Finally, we test the model's quality by determining the speedup or slowdown of its prediction against the original target heuristic and report this back to the search component (shown in figure 6(c) [1]).

Our system additionally implements parsimony. Genetic programming can quickly generate very long feature expressions. If two features have the same quality we prefer the shorter one. This selection pressure prevents expressions becoming needlessly long.

IV. Feature Grammars

In this section we explain how we use feature grammars to describe a feature space. We present a toy example to show how our process works. The grammars used in practice are more complex, but derived automatically from the IR.

1. We use cross validation to avoid over fitting to the training set. Note that this cross validation used by the search technique is separate from the cross validation we use to test the performance of our approach. A test set of programs is kept out and is never shown to any part of the machine learning tool in figure 6. After all features have been found we finally run predictions over this test set, determining how good our approach is.

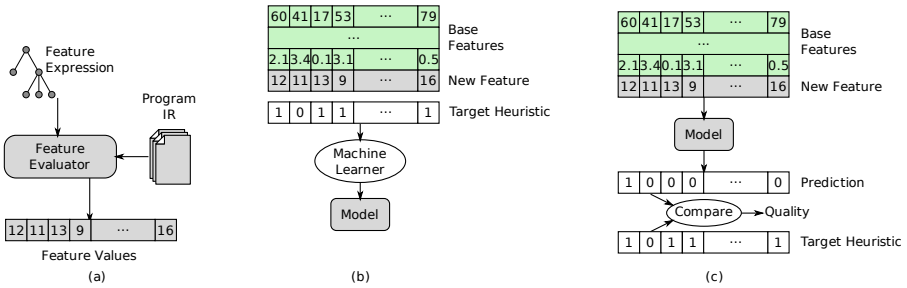

Figure 6. Machine learning used to determine the quality of a feature.

```
\scriptsize
<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> ")")?
```

Figure 7. Simple feature grammar.

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

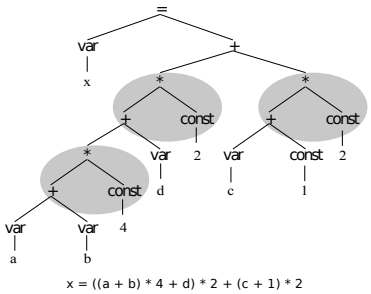

Figure 8. An example feature from the grammar in figure 7. In the right hand of the figure is a sample AST from the tiny language, showing the matching sub structures. The feature thus evaluates to three.

The toy language allows 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 assume that the ambiguity in operator precedence has been suitably dealt with. A simple example statement might be:

$$x = ((a + b) * 4 + d) * 2 + (c + 1) * 2$$

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, for instance. Figure 7 shows a simple grammar describing a set of such features in a pseudo-code style. Figure 8 shows an example feature from this grammar and an evaluation against a sample program fragment. Applying this particular feature to this piece of code yields the value three.

Now that we have our grammar describing the space of features, we can generate any number of features from it. We merely start at the root rule of the grammar (in this case, `feature`) and expand any non-terminals in it. Whenever there is a choice of production to expand, we choose randomly among them. By continuing until there are no more non-terminals left in the sentence we will have a finished feature.

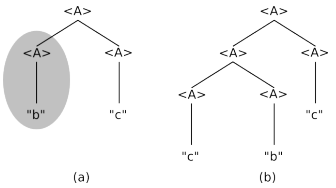

Figure 9. Mutation operator. In *(a)* a non-terminal is selected at random from an original parse tree. In *(b)* the non-terminal is replaced by a new random expansion of the same non-terminal.

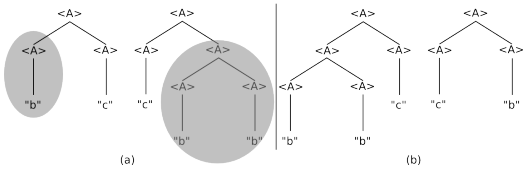

Figure 10. Crossover operator. Non-terminals of the same type from two parse trees *(a)*. Corresponding sub trees are then swapped, creating two new parse trees in *(b)*.

Searching the Feature Space. The features are represented by their parse trees in the corresponding grammar. We perform a search of these trees, keeping an initially random population which is evolved with a number of search operators. At each step or generation in the search we select a number of well performing individuals and apply these operators to them which creates a new population. Figures 9 and 10 show examples of the search operators used.

Our search technique is a hybrid between Grammatical Evolution [12] and Genetic Programming [13].

V. Experimental Setup

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

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 optimization and there exists prior work [2], [9] with which to compare. We extended the compiler to allow unroll factors to be explicitly specified for each loop in a program.

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

Platform. These experiments were run on a single unloaded, headless machine; an Intel single core Pentium 6 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.

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 fifteen. 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 compiled without inlining to increase the independence of loops. In total we gathered data for 2778 loops.

Measurement. One of the difficulties in evaluating the performance of compiler optimizations 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 one hundred times. We applied a standard statistical technique to reduce the effects of noise: applying a log transform and removing outliers outside the 1.5 x IQR (interquartile range). The best unroll factor for each loop was determined as that with the lowest average (across the 100 runs) cycle count.

VI. Experimental Methodology

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

Searching for Features. Our feature generator searches for one feature at a time. It prefers features which, in combination with the features selected by previous steps, most improve the performance of a machine learning tool. The genetic search for each feature consisted of a population

```
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
        ;
```

Figure 11. A simplified subset of the automatically generated grammar.

of one hundred individuals. Each was allowed to run until fifteen generations produced no improvement in the best feature of the population or a maximum of two hundred generations, whichever came first. Search for new features to add was stopped when either two and a half thousand total generations were reached or when we failed to find an improving feature five times.

Cross-validation and Machine Learning. We split the loops into ten groups keeping one group out for testing so that we can perform ten-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 [9], 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.

Search, Training and Deployment Cost. It took our system two 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 off activity that it is performed "at the factory" and would be easily parallelized. If we consider the amount of time it takes for a compiler writer to develop a good heuristic, this cost is in fact small.

It is theoretically possible for our system to produce extremely computationally expensive features, increasing compile time. The system forces these feature evaluations to time out, giving them at most two seconds to evaluate over all loops. 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 our system for unrolling have no significant impact on GCC's execution time.

Searching for Features for GCC. At the point at which loop unrolling occurs in GCC, the program has been lowered to the register transfer language (RTL). In RTL,

instructions are in an algebraic form with a treed, list-of-lists representation. Each node in the RTL may have some number of attributes. We extract the RTL representation of the loops, augmenting it to include the structure of the basic blocks in the loop and the RTL instructions contained within their blocks. We also export any information GCC can compute at that time such as estimated block frequencies, loop depths, and so on.

Once we have exported all loops in this way, we then examine the structure of the data. This allows us the automatic building of grammars that make features that match the structural facts observed in the RTL data. Moreover, this automation means that we have not had to hard code the grammar, making it easy to update in response to changes in the compiler.

Figure 11 shows a pared down snippet of the grammar, giving an example of some of the feature expressions that can be created. Our full grammar has many more functional capabilities and is also tuned to the structure of the RTL. We also use a slightly more terse syntax which helps when reading long features.

VII. 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 GCC's and a start-of-the-art machine learning schemes. This is followed by a brief analysis of the results.

A. 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 V we exhaustively enumerated loop unroll factors up to 15 for each loop, recording the best setting for each. We then ran each benchmark with the best unroll factors set and recorded the speedup. The bars labeled oracle in figure 12 show the maximum achievable speedups compared to no unrolling for our 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 12 labeled GCC, we see the performance of GCC across the same benchmark suite. In some case 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 compiler writers have in developing a portable optimization that delivers performance gains.

B. Our Approach

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

1) Comparison with GCC and Oracle. The bars in figure 13, labelled `Our`, 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 over 75% of the maximum speedup available. This is achieved entirely automatically and compares favorably with the 3% achieved by the hand-crafted GCC heuristic.

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 art scheme (`stateML`) based on a support-vector machine (SVM) described in [2]. We implemented this technique within GCC using the features described in [2] and shown in table 14. This model was trained and evaluated using cross-validation in exactly the same manner as ours. The bars labeled 'stateML' in figure 13 represent the performance achieved using this technique. On average it achieves 59%, outperforming GCC, which given that this was achieved automatically is significant. However, this is still short of the maximum achievable.

For the SVM method [14] we used the "one-vs-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 the remaining classes. At prediction time, when a new loop is presented, the classifiers are executed and the class (unroll factor) with the largest ouput is selected. In our experiments we have

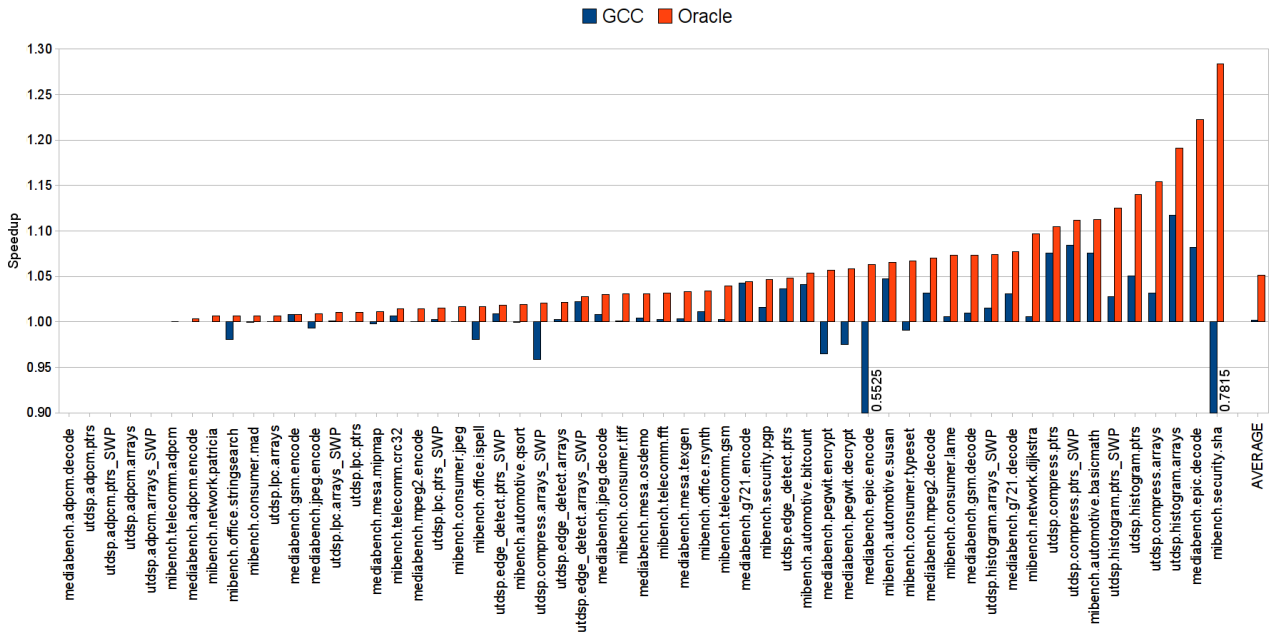

Figure 12. Speed up of the unroll factors chosen by GCC's default heuristic and the speed up of the best possible unroll factor - the oracle.

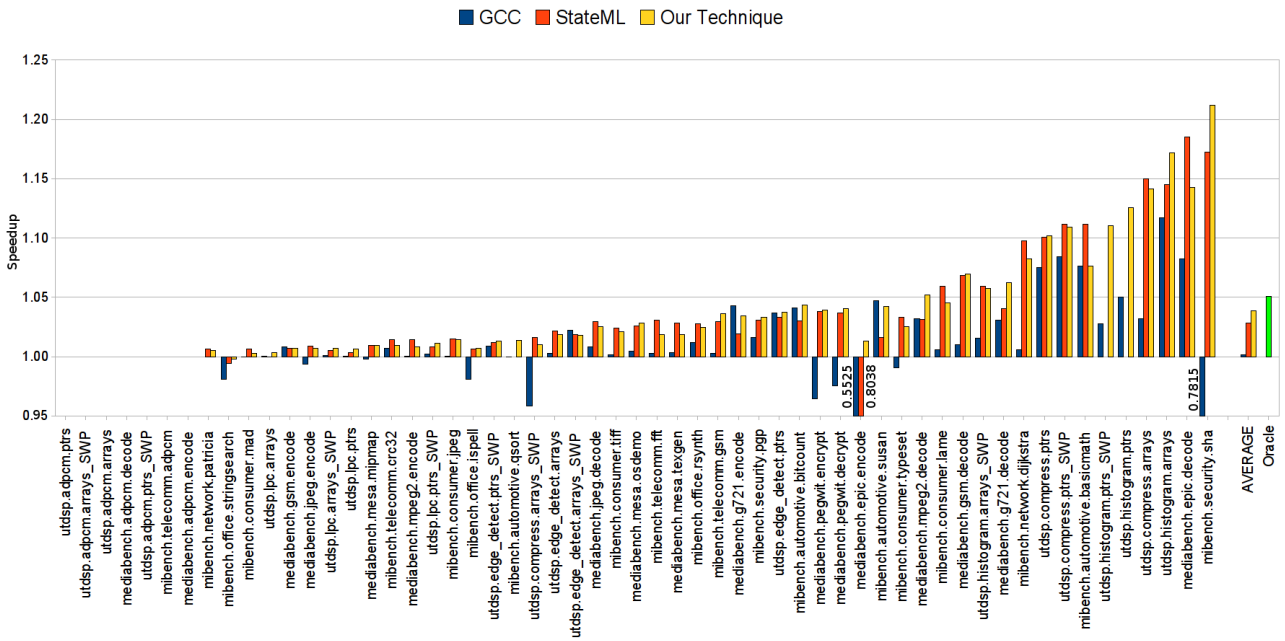

Figure 13. Comparison of speed ups between GCC's heuristic (`GCC`) , our technique (`Our`) and the state-ofthe-art ML approach (`stateML`). GCC achieves an average of 3% of the performance available, stateML achieves 59% of the performance available while our approach achieves 76% of the maximum performance available.

used the Gaussian Radial Basis Function (RBF) kernel:

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

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

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

The loop nest level	The number of operations in loop body
The number of floating point operations in loop body	The number of branches in loop body
The number of memory operations in loop body	The number of operands in loop body
The number of implicit instructions in loop body	The number of unique predicates in loop body
The estimated latency of the critical path of loop	The estimated cycle length of loop body
The language (C or FORTRAN)	The number of parallel "computations" in loop
The maximum dependence height of computations	The maximum height of memory dependencies of computations
The maximum height of control dependencies of computations	The average dependence height of computations
The number of indirect references in loop body	The minimum memory-to-memory loop-carried dependence
The number of memory-to-memory dependencies	The trip count of the loop (-1 if unknown)
The number of uses in the loop	The number of definitions in the loop.

Figure 14. The stateML features

stateML technique, it may be argued that both alternative schemes have good features, that (i) GCC just has a poorly implemented heuristics and (ii) that the stateML 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 15. Thus each of the 3 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 (labeled stateML Tree) slightly worsen from 59 to 53% of the maximum available. Combining the two sets of features, however, GCC and stateML, has no further impact on performance.

These results show that machine learning 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%.

C. Best features found

Figure 16 presents the first six out of thirty features found by our 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.

The first 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 our approach.

A few of the expression elements from the best features are explained below.

count(s) returns the number of elements in sequence, s
filter(s,m) filters sequence, s, removing any not matching expression m
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
/, //** and *[n]* are the children, descendants and particular child of the current node, respectively.

VIII. Related Work

The first work to automate compiler optimization was in the area of iterative or feedback-directed compilation which searches the program optimization space. Many smart search heuristics have been developed [4], [15]–[17]. Cooper *et al* [4], [18], [19] explore the optimization space using hill climbing and genetic algorithms. Other researchers have used analytical [20] or empirical models to explore the optimization space [21]. Agakov *et al* [21] built a model offline that is used to guide search. Haneda *et al* [22] make use of statistical inference to select good optimizations. In contrast to our work, these approaches do not make use of predictive modelling and require a recompilation of the program for each step in searching the optimization 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 optimization heuristics. One of the first researchers to incorporate machine learning into compiler for optimization were McGovern and Moss [6] who used reinforcement learning for scheduling of straight-line code. Cavazos *et al.* [8] 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.* [5] is the work most similar to ours. They used genetic programming (GP) to tune heuristic

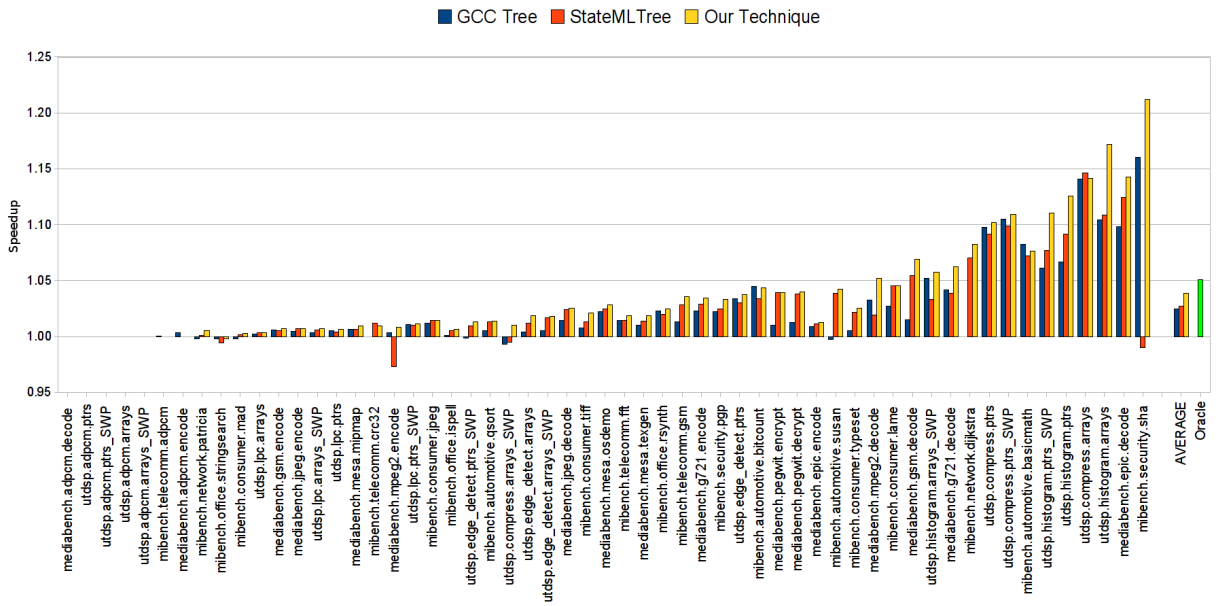

Figure 15. Speedup using a decision tree as the learning technique for GCC's, stateML compared to our technique. Keeping the machine learning algorithm identical shows the relative merits of the feature sets.

Number	Speedup	% of Max	Improvement	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))))

Figure 16. Best features found by our feature search in one fold

priority functions for three compiler optimizations. In contrast we use GP to search over *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. [9] 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 [2] went beyond Monsifort prediciting the actual unroll factor within the Open Research Compiler. Using support vector machines 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, this paper is the first to search and automatically generate features for machine learning.

IX. Conclusion and Further Work

In this paper we have developed a new technique to automatically generate good features for machine learning based optimizing compilation. By automatically deriving a feature grammar from the internal representation of the compiler, we can search a feature space using genetic programming.

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, dramatically outperforming features that were manually generated by compiler experts before.

In this paper 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 work, will investigate exploring different feature spaces for new optimizations.

Acknowledgment

This work is supported under the European project EU FP6 STREP MILEPOST IST-035307.

References

[1] J. Cavazos and M. F. P. O'Boyle, "Method-specific dynamic compilation using logistic regression," *SIGPLAN Not.*, vol. 41, no. 10, pp. 229–240, 2006.

[2] M. Stephenson and S. Amarasinghe, "Predicting unroll factors using supervised classification," in *CGO '05: Proceedings of the international symposium on Code generation and optimization*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 123–134.

[3] S. J. Beaty, "Genetic algorithms and instruction scheduling," in *MICRO 24: Proceedings of the 24th annual international symposium on Microarchitecture*. New York, NY, USA: ACM Press, 1991, pp. 206–211.

[4] K. D. Cooper, A. Grosul, T. J. Harvey, S. Reeves, D. Subramanian, L. Torczon, and T. Waterman, "Acme: adaptive compilation made efficient," *SIGPLAN Not.*, vol. 40, no. 7, pp. 69–77, 2005.

[5] M. Stephenson, S. Amarasinghe, M. Martin, and U.-M. O'Reilly, "Meta optimization: Improving compiler heuristics with machine learning," 06 2003.

[6] A. Mcgovern and E. Moss, "Scheduling straight-line code using reinforcement learning and rollouts," in *In Proceedings of Neural Information Processing Symposium*. MIT Press, 1998.

[7] R. Kohavi and G. H. John, "Wrappers for feature subset selection," *Artificial Intelligence*, vol. 97, no. 1-2, pp. 273–324, 1997.

[8] J. Cavazos and J. E. B. Moss, "Inducing heuristics to decide whether to schedule," *SIGPLAN Not.*, vol. 39, no. 6, pp. 183–194, 2004.

[9] A. Monsifrot, F. Bodin, and R. Quiniou, "A machine learning approach to automatic production of compiler heuristics," pp. 41–50, 2002.

[10] E. Moss, P. Utgoff, J. Cavazos, D. Precup, D. Stefanović, C. Brodley, and D. Scheeff", "Learning to schedule straight-line code," in *Advances in Neural Information Processing Systems*, M. I. Jordan, M. J. Kearns, and S. A. Solla, Eds., vol. 10. The MIT Press, 1998.

[11] J. R. Koza, "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*, J.-A. Meyer and S. W. Wilson, Eds. Paris, France: MIT Press, 1990, pp. 366–375.

[12] C. Ryan, J. J. Collins, and M. O Neill, "Grammatical evolution: Evolving programs for an arbitrary language," in *Proceedings of the First European Workshop on Genetic Programming*, W. Banzhaf, R. Poli, M. Schoenauer, and T. C. Fogarty, Eds., vol. 1391. Paris: Springer-Verlag, 14 1998, pp. 83–95.

[13] J. R. Koza, "The genetic programming paradigm: Genetically breeding populations of computer programs to solve problems," in *Dynamic, Genetic, and Chaotic Programming*, B. Soucek and the IRIS Group, Eds. New York: John Wiley, 06 1990, pp. 203–321.

[14] B. Schlkopf and A. J. Smola, *Learning with Kernels: Support Vector Machines Regularization, Optimization, and Beyond*. Cambridge, MA, USA: MIT Press, 2001.

[15] P. Kulkarni, S. Hines, J. Hiser, D. Whalley, J. Davidson, and D. Jones, "Fast searches for effective optimization phase sequences," *SIGPLAN Not.*, vol. 39, no. 6, pp. 171–182, 2004.

[16] Z. Pan and R. Eigenmann, "Fast and effective orchestration of compiler optimizations for automatic performance tuning," in *CGO '06: Proceedings of the International Symposium on Code Generation and Optimization*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 319–332.

[17] S. Triantafyllis, M. Vachharajani, N. Vachharajani, and D. I. August, "Compiler optimization-space exploration," in *Journal of Instruction-level Parallelism*. IEEE Computer Society, 2003, pp. 204–215.

[18] K. D. Cooper, P. J. Schielke, and D. Subramanian, "Optimizing for reduced code space using genetic algorithms," in *Workshop on Languages, Compilers, and Tools for Embedded Systems*, 1999, pp. 1–9.

[19] K. D. Cooper, D. Subramanian, and L. Torczon, "Adaptive optimizing compilers for the 21st century," *Journal of Supercomputing*, vol. 23, p. 2002, 2002.

[20] K. Yotov, X. Li, G. Ren, M. Cibulskis, G. DeJong, M. Garzaran, D. Padua, K. Pingali, P. Stodghill, and P. Wu, "A comparison of empirical and model-driven optimization," in *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*. New York, NY, USA: ACM, 2003, pp. 63–76.

[21] F. Agakov, E. Bonilla, J.Cavazos, B.Franke, G. Fursin, M. O'Boyle, J. Thomson, M. Toussaint, and C. Williams, "Using machine learning to focus iterative optimization," pp. 295–305, 03 2006.

[22] M. Haneda, P. M. W. Knijnenburg, and H. A. G. Wijshoff, "Automatic selection of compiler options using non-parametric inferential statistics," in *PACT '05: Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 123–132.