Using Machine Learning to Focus Iterative Optimization

F. Agakov, E. Bonilla, J.Cavazos, B.Franke, G. Fursin,
M.F.P. O'Boyle, J. Thomson, M. Toussaint, C.K.I. Williams
School of Informatics
University of Edinburgh
UK

Abstract

Iterative compiler optimization has been shown to outperform static approaches. This, however, is at the cost of large numbers of evaluations of the program. This paper develops a new methodology to reduce this number and hence speed up iterative optimization. It uses predictive modelling from the domain of machine learning to automatically focus search on those areas likely to give greatest performance. This approach is independent of search algorithm, search space or compiler infrastructure and scales gracefully with the compiler optimization space size. Off-line, a training set of programs is iteratively evaluated and the shape of the spaces and program features are modelled. These models are learnt and used to focus the iterative optimization of a new program. We evaluate two learnt models, an independent and Markov model, and evaluate their worth on two embedded platforms, the Texas Instrument C6713 and the AMD Au1500. We show that such learnt models can speed up iterative search on large spaces by an order of magnitude. This translates into an average speedup of 1.22 on the TI C6713 and 1.27 on the AMD Au1500 in just 2 evaluations.

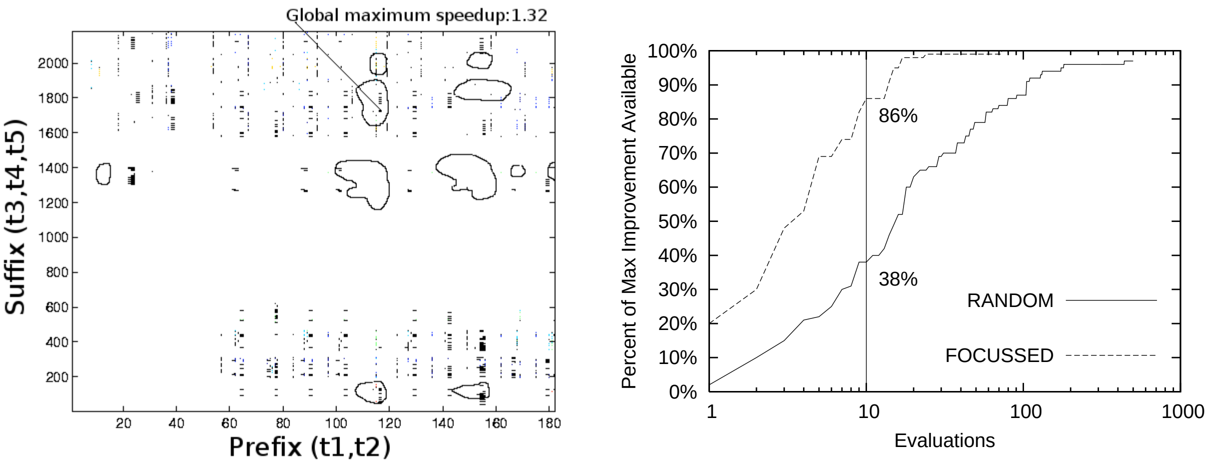
1 Introduction

Using iterative search as a basis for compiler optimization has been widely demonstrated to give superior performance over static schemes [1, 3, 11]. The main drawback of these schemes is the amount of search time needed to achieve performance improvements given that each point of the search is a recompilation and execution of the program. Although multiple recompilations/executions are acceptable for embedded code, libraries and persistent applications, these long compilation/execution cycles restrict the space of options searched. On a larger scale, they are a significant barrier to adoption in general purpose compilation.

There have been a number of papers focusing on reducing the cost of iterative optimization. As a single evaluation consists of a compilation plus execution of the program, two recent papers have investigated reducing the cost of an *individual* compilation or execution [7]. In [1, 6] a more radical approach is used to reduce the total *number* of evaluations. Cooper et al [6] examine the structure of the search space, in particular the distribution of local minima relative to the global minima and devise new search based algorithms that outperform generic search techniques. An alternative approach is developed in [20]. Here the space of compiler options is examined off-line on a per function basis and the best performing ones classified into a small tree of compiler options. When compiling a new program, the tree is searched by compiling and executing the best path in the tree. As long as the best sequences can be categorized into a small tree, this proves to be a highly effective technique.

This paper develops a new methodology to speed up iterative optimization. It automatically focuses any search on those areas likely to give the greatest performance. This methodology is based on machine learning, is independent of search algorithm, search space or compiler infrastructure and scales gracefully with the compiler optimization space size. It uses program features to correlate the program to be optimized with previous knowledge in order to focus the search. Off-line, a training set of programs is iteratively evaluated and the shape of the spaces and program features are recorded. From this data, our scheme automatically learns a model which predicts those parts of the optimization space that are likely to give good performance improvements for different classes of programs. When a new program is then encountered, an appropriate predictive model is selected, based on program features, which then biases the search to a certain area of the space. Using this technique we are able to speed up search by an up to an order of magnitude on large spaces.

This paper is structured as follows. Section 2 provides a motivating example demonstrating how learning where to search can significantly reduce the number of evaluations needed to find good performance improvements. This is



(a) (b)

Figure 1. (a) Points corresponding to those transformation sequences whose performance is within 5 per cent of the optimum for *adpcm* on the TI C6713. The contour is the predicted area for good optimizations. (b) How close to the best performance random and focused search achieve for each program evaluation. The random algorithm achieves 38 % of the maximum improvement in 10 evaluations; the focused search 86%.

followed in section 3 by a description of the experimental setup, an analysis of the optimization spaces encountered and an examination of how two standard search algorithms, random and genetic, perform. This is followed in section 4 by a description of two predictive models and a demonstration of how they can be used to speed up search. Section 5 describes how the standard machine learning techniques of principal components analysis and nearest neighbors are used to learn the predictive models. In section 6, these learnt models are then tested on (i) a medium sized exhaustively enumerated space and (ii) a very large space, where they are shown to improve search performance by up to an order of magnitude. Section 7 describes related work and is followed in section 8 with some brief conclusions.

2 Motivation & Example

This paper focuses on embedded applications where performance is critical and consequently there has been a large body of work aimed at improving the performance of optimizing compilers, e.g. [14]. Most of this work focuses on improving back-end, architecture specific compiler phases such as code generation, register allocation and scheduling. However, the investment in ever more sophisticated back-end algorithms produces diminishing returns. Iterative approaches based on back-end optimizations consequently give relatively small improvements [5]. In this paper, we consider source-level transformations [19, 8] for embedded systems. Such an approach is, by definition, highly portable from one processor to the next and provides additional benefit to the manufacturer's highly tuned compiler. However, this portability comes at cost. For example, in [8], Franke et al require 1000 evaluations to achieve reasonable performance improvements.

Search space The reason for this excessive search time is that determining the best high level sequence of transformations for a particular program is non-trivial. Consider the diagram in figure 1 (a) showing the behavior of the `adpcm` program on the Texas Instrument's C6713. This diagram is an attempt at plotting all of the good performing points (within 5% of the optimum) in the space of all transformations of length 5 selected from a set of 14 transformations. It therefore covers a space of size 14^5. It is difficult to represent a large 5 dimensional space graphically so each good performing transformation sequence $(t_1t_2t_3t_4t_5)$ is plotted at position (t_1t_2) on the x-axis, which denotes prefixes of length 2, and position $(t_3t_4t_5)$ on the y axis, which denotes suffices of length 3. The most striking feature is that minima are scattered throughout the space and finding the very best is a difficult task. Prior knowledge about where good points were likely to be, could *focus our search* allowing the minimal point to be found faster. Alternatively, given a fixed number of evaluations, we can expect improved performance if we know good areas to search within.

Focused search In this paper we develop a technique that learns off-line, ahead of time, a predictive model from iterative evaluations of other programs. This predictive model then defines good regions of the space to search. In figure 1 (a) the contour lines enclose those areas where our technique predicts there will be good points. Using this prediction we are able to reduce the number of searches to achieve the same performance - rapidly reducing the cost of iterative search. This can be seen in figure 1 (b), which compares random search (averaged over 20 trials to be statistically meaningful) with and without the predictive model focus. The x-axis denotes (logarithmic scale) the number of evaluations performed by the search. The y-axis denotes the best performance achieved so far by the search ; 0% represents the original code performance, 100% the maximum performance achievable. It is immediately apparent that the predictive model rapidly speedups up the search. For instance, after 10 evaluations, random searching achieves 38% of the potential improvement available while the focused search achieves 86%. As can be seen from figure 1 (b), such a large improvement would require over 80 evaluations using random search, justifying further investigation of predictive models.

3 Optimization Space

This paper develops machine learning techniques to improve the search performance of iterative optimization. This section briefly describes the benchmarks we use, the program transformations which make up the optimization space and the embedded platforms we evaluate. It then characterises the space and presents two standard search algorithms which are later used to show how learnt predictive models can dramatically speed up search.

3.1 Experimental setup

Benchmarks The *UTDSP* [13, 17] benchmark suite was designed "to evaluate the quality of code generated by a high-level language (such as C) compiler targeting a programmable digital signal processor (DSP)" [13]. This set of benchmarks contains small, but compute-intensive DSP kernels as well as larger applications composed of more complex algorithms. The size of programs ranges from 20-500 lines of code where the runtime is usually below 1 second. However, these programs represent compute-intensive kernels widely regarded most important by DSP programmers and are used indefinitely in stream-processing applications.

Transformations In this paper we consider source to source transformations (Many of these transformations also appear within the optimisation phases of a native compiler[1]), applicable to C programs and available within the restructuring compiler SUIF 1 [10]. For the purpose of this paper, we have selected eleven transformations described and labeled in table 1. As we (arbitrarily) consider four loop unroll factors, this increases the number of transformations considered to 14. We then exhaustively evaluated all transformations sequences of length 5 selected from these 14 options. This allows us to evaluate the relative performance of our proposed techniques. In the later evaluation section (see section 7), we also consider searching, non exhaustively, a much larger space.

Label	Transformation
1,2,3,4	Loop unrolling
f	Loop flattening
n	FOR loop normalization
t	Non-perfectly nested loop conversion
k	Break load constant instructions
s	Common subexpression elimination
d	Dead code elimination
h	Hoisting of loop invariants
i	IF hoisting
m	Move loop-invariant conditionals
c	Copy propagation

Table 1. The labeled transformations used for the exhaustive enumeration of the space. 1,2,3,4 corresponds to the loop unroll factor

Platforms Our experiments were performed on two distinct platforms to demonstrate that our technique is generic. **TI:** The Texas Instrument C6713 is a high end floating point DSP. The wide clustered VLIW processor has 256kB of internal memory. The programs were compiled using the TI's Code Composer Studio Tools Version 2.21 compiler with the highest -O3 optimization level and -ml3 flag (generates large memory model code). **AMD:** The AMD Alchemy Au1500 processor is an embedded SoC processor using a MIPS32 core (Au1), running at 500MHz. It has 16KB instruction cache and 16KB non-blocking data cache. The programs were compiled with GCC 3.2.1 with the -O3 compile flag. According to the manufacturer, this version/option gives the best performance - better than later versions of GCC - and hence was used in our experiments.

3.2 Characterizing the Space

In order to characterize the optimization space, we exhaustively enumerated all 14^5 transformation sequences on both platforms. Table 2 summarizes the performance avail-

	TI		AMD	
Prog.	**Improv.**	**Seq.**	**Improv.**	**Seq.**
fft	3.64%	{3nm}	4.49%	{4hns}
fir	45.5%	{4}	26.7%	{3}
iir	16.3%	{3h}	29.5%	{h4}
latnrm	0.34%	{nsch}	27.1%	{csh4}
lmsfir	0.39%	{1s}	30.3%	{s3}
mult	0.00%	{}	30.5%	{4}
adpcm	24.0%	{1ish}	0.75%	{ism}
compress	39.1%	{4s}	24.0%	{hs4}
edge	5.06%	{3}	23.1%	{ch4}
histogram	0.00%	{}	24.7%	{4}
lpc	10.7%	{sn2}	6.01%	{h4cnm}
spectral	7.46%	{n4}	8.53%	{sh4}
Average	15.2%	-	19.64%	-

Table 2. Summary of optimization space on the TI and AMD using exhaustive search

able; columns 2 and 3 refer to the TI while columns 4 and 5 refer to the AMD respectively.

Improved execution time The columns labeled `Improv.` (cols. 2 and 4) shows the maximum reduction in execution time obtained on the TI and AMD within this exhaustively enumerated space. Eight (out of twelve) benchmarks for Texas Instruments and eleven (out of twelve) benchmarks for AMD achieved significant improvement. The best execution time reduction was 45.5% on the TI and 30.5% on the AMD. On average, a 12.7% reduction was achieved for the TI and 13.8% for the AMD. This translates into an average speedup of 1.15 and 1.16 over the platform specific optimizing compiler.

Best performing sequences The columns labeled `Seq.`, (columns 3 and 5) in table 2 contain the best performing sequence for each benchmark on each machine. The individual letters within each entry refer to the labeled transformations in table 1, e.g. i = `if hoisting`. These entries show that the complexity and type of good transformation sequences is program dependent. While benchmarks such as *fir* and *edge_detect* for the TI and *fir*, *mult* and *histogram* for the AMD reach their best performance with single transformations, other benchmarks such as *adpcm* for the TI and *lpc* for the AMD obtain their minimum execution time with four and five-length sequences respectively. Similarly, transformations that yield good performance on some benchmarks do not appear in the best sequences of other programs. For example, on the AMD the sequence {*ism*} makes `adpcm` run at its minimum execution time; however, none of these three individual transformations is present in the best performing sequence of `compress`.

Critically, the best performing sequence on one program is never the best for another. Therefore, a technique which tries to simply apply the best sequence found on other programs is unlikely to succeed.

3.3 Search algorithms

In this section, we describe two common methods used to search the transformation spaces: a blind random search (RA) and a "smarter" genetic algorithm (GA). Random search generates a random string of transformations where each transformation is equally likely to be chosen and performs surprisingly well in our experience. We configured our GA in the same manner as "best" GA in [6] with an initial randomly selected population of 50.

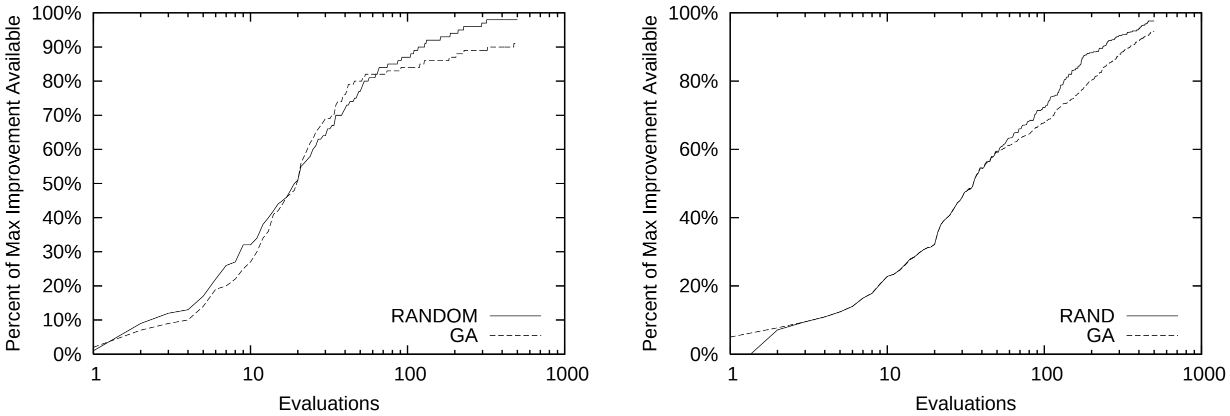
For the exhaustively enumerated space, both algorithms have similar performance as can be seen in figure 2. Here we plot the best performance achieved so far by each algorithm against how many program evaluations have been performed. This plot is averaged over all programs. Improvements by either algorithm are more easily achieved on the TI due to the much greater number of sequences giving a significant speedup.

Both algorithms have similar overall performance with the GA performing well on the AMD in the early part of the search. However, random search performs better after a large number of evaluations as the GA appears to more likely to be stuck in local minima. In both cases, however, large numbers of evaluations are needed to gain any significant performance improvements. In the next section we investigate how predictive models can speed up this search.

4 Models to focus search

In order to speed up the search algorithm we wish to focus our attention on profitable areas of the optimization space. We wish to build a model of those transformation sequences for which a program obtained good performance in the hope that this can be learnt and used on later programs. We could simply record the best sequence achieved on other programs and hope that it improves our current program. However, this has a number of flaws. Firstly, as the results in table 2 show, the best transformation on one program is never the best on others. Secondly, knowing the best sequence on another program only provides one single option and cannot guide subsequent search within a larger space.

Alternatively, we can build intricate models that characterize the performance of all transformation sequences. Here the problem is that we can easily overfit the model to the data so that it cannot be generalized to other programs.



(a) **TI** (b) **AMD**

Figure 2. Performance with respect to evaluations for the random (RAND) and genetic (GA) search algorithms on the TI (a) and AMD (b). The x-axis denotes (logarithmic scale) the number of evaluations performed by each search. The y-axis denotes the best performance achieved so far by the search; 0 % represents the original code performance, 100% the maximum performance achievable. Results averaged over all benchmarks

Furthermore, such a complex model will require extensive training data, which may be costly to gather and is unrealistic in practise. In this section we consider two different models which try to summarize the optimization space without excessive overfitting. We consider (i) a simple independent distribution model and (ii) a more complex Markov model. Both of these require relatively small amounts of training data to construct and should be easy to learn (see section 5).

4.1 Independent identically distributed (IID) model

It makes sense to start with the simplest approach first: modelling program transformations as if they were independent. We know that this assumption does not hold in general, but it might be sufficient to better focus search algorithms. Consider a set of N transformations $\mathcal{T} = \{t_1, t_2, \ldots, t_N\}$. Let $\mathbf{s} = s_1, s_2, \ldots, s_L$ be a sequence of transformations $\mathbf{s}$ of length L, where each element s_i is chosen from the transformations in $\mathcal{T}$. Under the independent model we assume that the probability of a sequence of transformations being good is simply the product of each of the individual transformations in the sequence being good, i.e.:

$$P(s_1, s_2, \ldots, s_L) = \prod_{i=1}^{L} P(s_i). \qquad (1)$$

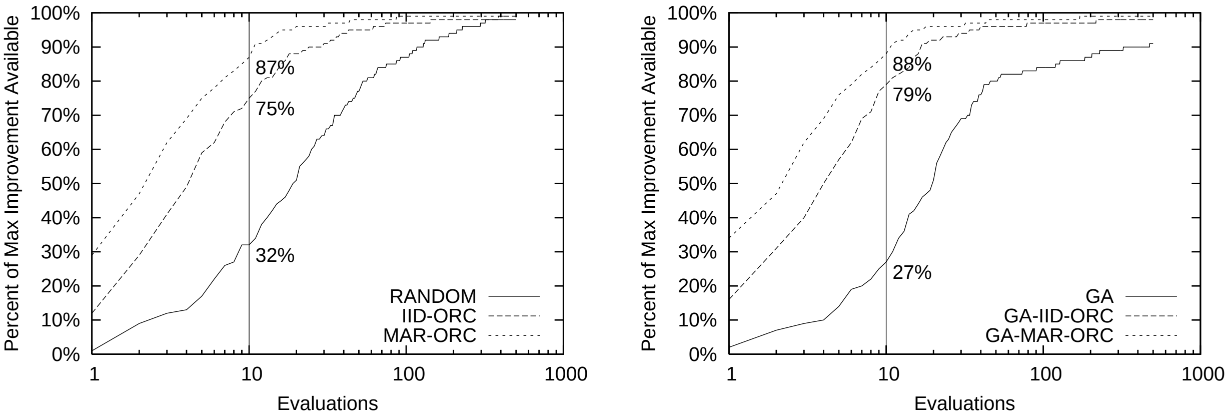
Here $P(t_j)$ is the probability that the transformation t_j occurs in good sequences. For our dataset we have chosen the set of good sequences to be those sequences that have an improvement in performance of at least 95% of the maximum possible improvement. We calculate $P(t_j)$ by simply counting the number of times t_j occurs in good sequences and normalize the distribution i.e. $\sum_{i=1}^{N} P(t_j) = 1$. We then record within a vector the probability of each of the $N = 14$ transformations.

For each benchmark we can build this probability vector or IID distribution. We refer to this as the *IID-oracle*. It is an oracle in the sense that we can only know its value once we have exhaustively enumerated the space, which in practise is unrealistic. Our goal is to be able to predict this oracle by using machine learning techniques based on a training set of programs in order to improve search. However, it is necessary to prove first that this oracle distribution does indeed lead to better search algorithms.

4.2 Markov Model

As described above, the IID probability distribution function assumes that all transformations are mutually independent neglecting the effect of interactions among transformations. This can be very restrictive, particularly when there are transformations that enable the applicability of other transformations or when some of them only yield good performance when others are applied. Therefore, including these interactions in our technique makes possible the construction of richer models that ideally will improve biased search algorithms and will obtain good performance in fewer evaluations.

In order to keep the number of samples needed for building our probability density function low while including in-



(a) **TI: Random** (b) **TI: GA**

Figure 3. TI: Random (a) and GA (b) search versus IID-oracle and Markov oracle. Results averaged over all benchmarks

teractions among transformations, we can take one step further from the IID distribution by using a Markov chain. A Markov chain for transformation sequences can be defined as follows:

$$P(\mathbf{s}) = P(s_1)\prod_{i=2}^{L} P(s_i|s_{i-1}).$$

The equation above states that the probability of a transformation applied in the sequence depends upon the transformations that have been applied before. The main assumption under this model is that these probabilities do not change along the sequence, i.e. they are the same at any position of the sequence, and therefore the model is often referred as a stationary Markov chain. This oversimplification prevents the number of parameters of the model from increasing with the length of the sequences considered.

Thus, the parameters of the model are the probability at the first position of the sequence $P(s_1)$ and the transition matrix $P(s_i|s_{i-1})$ with $i = 1, \ldots, L$, which as before can be learnt from data by counting. Once again $\sum_{j=1}^{N} P(s_1 = t_j) = 1$ and $\sum_{j=1}^{N} P(s_i = t_j|s_{i-1}) = 1$ must be satisfied.

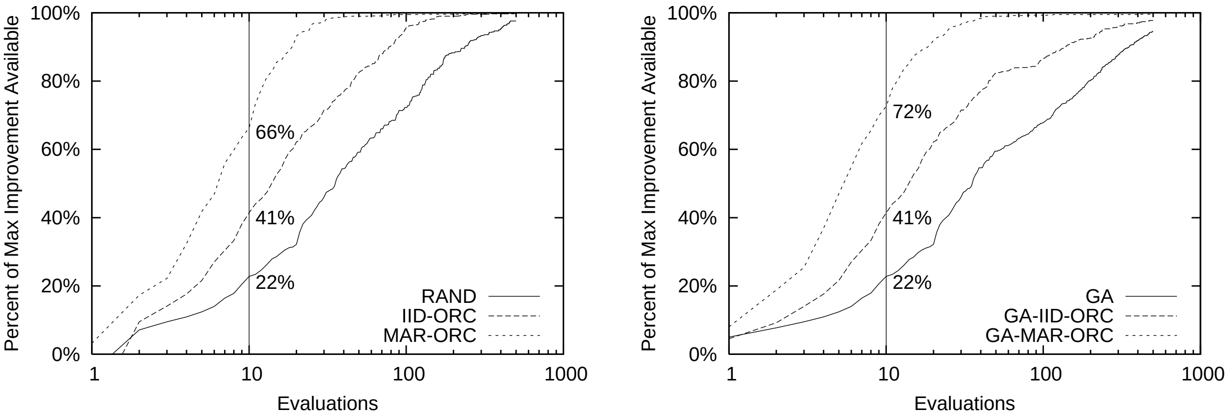
As in section 4.1 the parameters of the model have been learnt from those sequences that have an improvement in performance at least 95% of the maximum possible improvement. Using this model gives a 14 x 14 matrix.

4.3 Speeding up search: Evaluating the potential of the models

To test the potential of our scheme, we compared each baseline search algorithm against this same algorithm using each predictive model. For the random algorithm, instead of having a uniform probability of a transformation being selected, each model biases certain transformations over others. In the case of the GA, the initial population is selected based on the model's probabilities and then the GA is allowed to evolve as usual.

We construct each model using the results obtained from searching a particular program's space and then test each model-enabled search algorithm on the *same* benchmark; we call these two learnt models: *IID-oracle* and *Markov-oracle*. These "oracles" form an upper-bound on the performance we can expect to achieve when later trying to learn each model. This is to evaluate whether such models can improve the search. Clearly, if the best a model oracle can achieve is insignificant, it is not worth expending effort in trying to learn it.

Figure 4 (a) depicts the average performance, over all our benchmarks, of the baseline random algorithm against random search biased with the two oracles on the TI. Similarly, Figure 4 (b) depicts the performance of the baseline GA algorithm versus using the two oracles to generate the initial population. In both figures, we see that the oracles can significantly speed up finding a good solution. For example, at evaluation 10, `random` achieves less than 20% of the maximum available performance. In contrast, `random + IID-oracle` achieves more than 50% of the available performance and `random + Markov-oracle` achieves around 80% of the performance. Figures 5 depicts a similar picture on the AMD architecture. On the AMD architecture, our two oracles significantly improve the performance of each baseline algorithm. The baseline random search algorithm only achieves 4% of the available performance after 10 evaluations. In contrast, `random + IID-oracle` achieves about 20% of the available performance (5 times better than base) and `random + Markov-oracle` achieves 40% of the available perfor-



(a) **AMD: Random** (b) **AMD: GA**

Figure 4. AMD: Random (a) and GA (b) search versus IID-oracle and Markov-oracle. Results averaged over all benchmarks

mance, an order of magnitude better than the base random algorithm. On average, the baseline algorithm needs 100 evaluations to achieve the same performance as the baseline + the Markov oracle achieves with just 10 evaluations.

We can see from these figures, that the IID and Markov models have the potential to dramatically improve the performance of both search algorithms. In the next section we describe how we can learn these models from previous offline runs to build a predictive model.

5 Learning a model

The biggest difficulty in applying knowledge learnt offline to a novel input is considering exactly which portions of this knowledge are relevant to the new program. We show that, as is the case in many other domains, programs can be successfully represented by program features, which can then be used to gauge their similarity and thus the applicability of previously learnt off-line knowledge.

Obviously, the selection of these program features is critical to the success of this method, and so we employ a well known statistical technique, principal component analysis (PCA) [2], to assist the selection. Initially, we identified thirty-six loop-level features we thought might describe the characteristics of a program well, and use them as input for the PCA process as shown in table 3. PCA tells us that, in this instance, due to redundancy and covariance in the features' values, these thirty-six features can be combined in such a way that they can be reduced to only five features, whilst retaining 99% of the variance in the data. The output of this process is a 5-D feature vector for each benchmark, containing these five condensed feature values.

Nearest Neighbors By using a nearest neighbors classifier [2], we can select which of our previously analyzed programs our new program is most similar to. Learning using nearest neighbors is simply a matter of mapping each 5-D feature vector of our training programs (all our benchmarks) onto a 5-D feature space.

Classification When a novel program is compiled, it is first put through a feature extractor, and those features processed by PCA. The resulting 5-D feature vector is mapped onto the 5-D feature space, and the Euclidean distance between it and every other point in the space calculated. The closest point is considered to be the 'nearest neighbor' and thus the program associated with that point is the most similar to the new program.

We can apply this process to each of our twelve benchmarks by using leave-one-out cross-validation, where we disallow the use as training data of the feature vector associated with the program that is currently being evaluated, otherwise a program would always select itself as its nearest neighbor.Having selected a neighbor, a previously learnt probability distribution for that selected neighbor is then used as the model for the new program to be iteratively optimized.

5.1 Evaluating learning

It is useful to know how close our learnt distribution is to the oracle distribution for both models, IID and Markov. Averaged across all benchmarks, the learnt distribution achieves approximately 80% of the performance per evaluation of the *IID-oracle* and the *Markov-oracle* on the TI. On the AMD, we achieve a similar result - approximately 75 % of both oracles' performance.

Features
for loop is simple?
for loop is nested?
for loop is perfectly nested?
for loop has constant lower bound?
for loop has constant upper bound?
for loop has constant stride?
for loop has unit stride? number of iterations in for loop
loop step within for loop
loop nest depth
no, of array references within loop
no. of instructions in loop
no. of load instructions in loop
no. of store instructions in loop
no. of compare instructions in loop
no. of branch instructions in loop
no. of divide instructions in loop no. of noop instructions in loop
no. of call instructions in loop
no. of generic instructions in loop
no. of array instructions in loop
no. of memory copy instructions in loop
no. of other instructions in loop
no. of float variables in loop
no. of int variables in loop
both int and floats used in loop?
loop contains an if-construct?
loop contains an if statement in for-construct?
loop contains an if-construct?
loop iterator is an array index?
all loop indices are constants?
array is accessed in a non-linear manner?
loop strides on leading array dimensions only?
loop has calls?
loop has branches?
loop has regular control flow?

Table 3. Features used

As the oracles have been shown to improve performance and we are able to achieve a significant percentage of their improvement, this suggests that both learnt models should give significant performance improvement over existing schemes. This is evaluated in the next section.

6 Evaluation

This section evaluates our focused search approach on two optimization spaces. The first space is the exhaustively enumerated 14^5 space described throughout this paper. The second is a much larger space of size 82^{20} i.e. transformation sequences of length 20 with each transformation selected from one of 82 possible transformations available in SUIF 1 [10]. This was achieved using the standard leave one out cross-validation scheme i.e. learn the IID and Markov models based on the *training* data from all other programs *except* for the one about to be optimized or *tested*

6.1 Evaluation on exhaustively enumerated space

Initially, we ran both the baseline random and GA search algorithms for 500 program evaluations and recorded their speedup over time on both the TI and AMD. We then ran the same algorithms again, this time using the two learnt models: IID and Markov. This was achieved using the standard leave one out cross-validation scheme i.e. learn the IID and Markov models based on the *training* data from all other programs *except* for the one about to be optimized or *tested*.

The results for the TI and AMD are shown in figures 6 and 7 respectively. On the TI the learnt IID based models achieve approximately twice the potential performance of either baseline algorithm after 10 evaluations (60%/62% vs 32%/27%) . The learnt Markov model does even better, achieving 79% of the perfomance available after the same number of evaluations. The baseline algorithms would need over 40 evaluations to achieve this same performance improvement. On the AMD, the performance improvements are less dramatic, yet the learnt Markov based algorithms achieves more than twice the performance of the baseline algorithms after 10 evaluations.

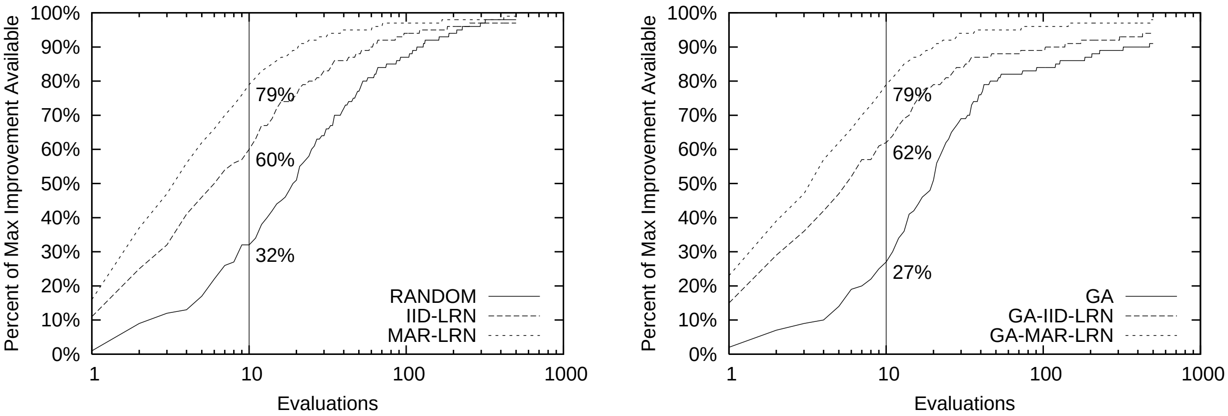
6.2 Evaluation on large space

Experiments within an exhaustively enumerated space are useful as the performance of a search algorithm can be evaluated relative to the absolute minima. However, in practise when we wish to search across a large range of transformations, it is infeasible to run exhaustive experiments. Instead we ran a random search for 1000 evaluations on each program space as off-line training data.

This time we wish to focus on the performance achieved in the early parts of iterative optimization. So, we ran the baseline random search algorithm and both learnt models for just 50 evaluations. As the genetic algorithm and random search have the same behaviour for the first 50 evaluations, the GA was not separately evaluated

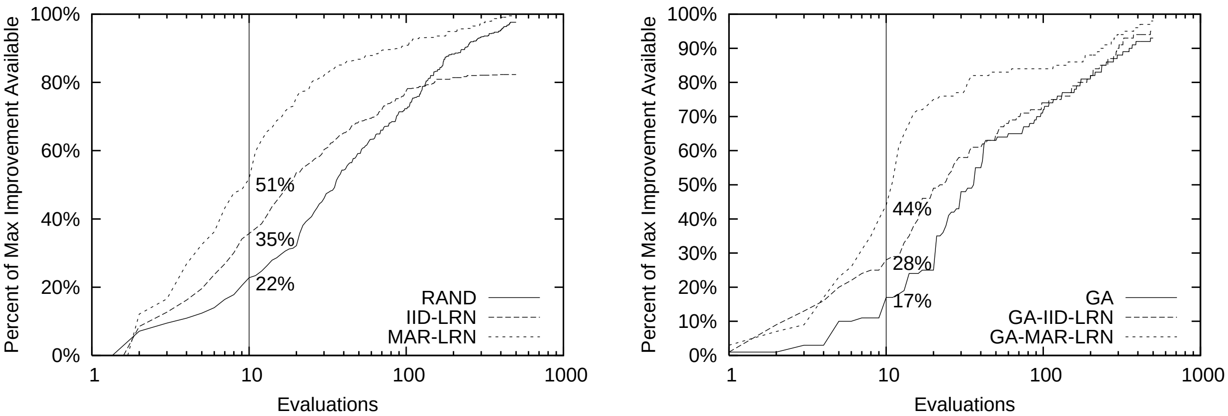
The speedups for each benchmark after 2, 5, 10 and 50 evaluations on the TI is shown in figure 7. Due to time constraints, only those benchmarks with non-negligible speedup on the exhaustively enumerated space are evaluated. The learnt models both deliver good performance and the random + IID learnt model achieves an average speedup of 1.26 after just 2 evaluations. Furthermore, the random + IID learnt model achieves a greater average performance after 5 evaluations (1.34) than the baseline random algorithm does after 50 evaluations (1.29).

Surprisingly, the IID learnt model achieves better performance than the Markov learnt model after 50 evaluations 1.41 vs 1.30 speedup in contrast to the results of the exhaustively enumerated space (see figures 6 and 7). The rea-



(a) **TI: Random** (b) **TI: GA**

Figure 5. TI: Random (a) and GA (b) search versus IID-learnt and Markov-learnt. Results averaged over all benchmarks



(a) **AMD: Random** (b) **AMD: GA**

Figure 6. AMD: Random (a) and GA (b) search versus IID-learnt and Markov-learnt. Results averaged over all benchmarks

son is that the Markov model needs a greater number of training evaluations than the IID model to model the space accurately. Here we have only 1000 evaluations to build a model.

Similarly, the speedups for the AMD are shown for each benchmark after 2, 5, 10 and 50 evaluations on in figure 8. Again both learnt models significantly outperform the baseline random algorithm. In fact the random + Markov learnt model achieves a greater average performance (1.33) after 5 evaluations than random does after 50 evaluations (1.32). It therefore achieves this level of performance an order of magnitude faster - the same is also true for the TI. Once again random + IID unexpectedly outperforms random + Markov at 50 evaluations. Thus after just 2 evaluations a speedup of 1.27 is found on average, almost three times the performance of the baseline algorithm.

Finally, the *single* sequence that gives the best performance on average on the AMD in the small space is `humc3`. This gives an average speedup of 1.11, significantly less than that achieved by random + Markov after just 2 evaluations. On the TI, there does not exist a single sequence which gives any performance improvement on average.

Discussion The Markov predictor performs less well on the exhaustive space due to the reduced amount of training data. This suggests that the IID model should initially be used on a new platform when there is relatively small amounts of training data available. Over time, once sufficient new data is accrued by iterative optimization, this can be used for a second stage of learning using the Markov

TI	2 Evaluations			5 Evaluations			10 Evaluations			50 Evaluations		
Benchmark	R	M	I	R	M	I	R	M	I	R	M	I
fft	1.00	1.00	1.00	1.01	1.01	1.34	1.00	1.01	1.65	1.34	1.21	1.81
fir	1.18	1.66	1.67	1.25	1.66	1.83	1.37	1.66	1.85	1.70	1.85	1.85
iir	1.14	1.20	1.19	1.18	1.23	1.19	1.19	1.23	1.21	1.19	1.23	1.23
adpm	1.08	1.33	1.17	1.18	1.33	1.18	1.25	1.35	1.24	1.28	1.43	1.28
edg	1.08	1.13	1.27	1.15	1.13	1.28	1.21	1.13	1.28	1.25	1.13	1.29
lpc	1.09	1.05	1.13	1.10	1.05	1.16	1.10	1.10	1.18	1.24	1.12	1.27
spe	1.01	1.10	1.15	1.03	1.17	1.16	1.05	1.17	1.16	1.07	1.17	1.18
AVG	1.08	1.21	1.22	1.12	1.22	**1.34**	1.16	1.23	1.36	**1.29**	1.30	1.41

Figure 7. Speedups up achieved by random search (R), random + Markov learnt model (M), random + IID learnt model (I) after 2, 5, 10 and 50 evaluations on each benchmark on the TI processor. Random + IID learnt model achieves greater average performance (1.34) after 5 evaluations than random does after 50 evaluations (1.29)

model.

7 Related Work

Iterative search-based optimization As well as the work of Almagor et al [1] and Triantafyllis et al [20] described in the introduction, there have been a number of related projects. A partially user-assisted approach to select optimisation sequences for embedded applications is described in [11]. This approach combines user guides and performance information with a genetic algorithm to select local and global optimisation sequences. Other authors [9, 5] have explored ways to search program- or domain-specific command line parameters to enable and disable specific options of various optimising compilers. In [8] iterative high level optimizations are applied to several embedded processors using two probabilistic algorithms. Good speedups are obtained at the expense of very large number of evaluations. Finally, in [**?**], it is shown that carefully hand generated models can approach the performance of iterative optimisation.

Machine Learning Machine learning predictive modelling has been recently used for non-search based optimisation. Here the compiler attempts to learn off-line a good optimization heuristic which is then used instead of the compiler writer's hand-tuned method.

Stephenson *et al.* [18] used genetic programming to tune heuristic priority functions for three compiler optimizations: within the Trimaran's IMPACT compiler. For two optimizations they achieved significant improvements. However, these two pre-existing heuristics were not well implemented. Turning off data prefetching completely is preferable and reduces many of their significant gains. For the third optimization, register allocation, they were only able to achieve on average a 2% increase over the manually tuned heuristic.

Cavazos *et al.* [4] describe using supervised learning to control whether or not to apply instruction scheduling. No absolute performance improvements were reported however.

Finally, Monsifrot et al. [15] 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 where there learnt scheme showed modest improvement.

8 Conclusion and Future Work

This paper develops a new methodology to speed up iterative compilation. It automatically focuses any search on those areas likely to give greatest performance. It use predictive modelling and program features to learn profitable areas of the optimization space to search. Experiments demonstrate that this approach is highly effective in speeding up iterative optimization.

Currently, we have a one-off training/learning phase to build a model which is then applied to each new program. An obvious next step is to continuously update the learnt model after each new program is iteratively optimized, similar in spirit to lifelong compilation [12]. Future work will investigate different predictive models on new spaces to further improve the performance of search based optimization.

AMD	2 Evaluations			5 Evaluations			10 Evaluations			50 Evaluations		
Benchmark	R	M	I	R	M	I	R	M	I	R	M	I
fft	1.00	1.04	1.04	1.00	1.05	1.07	1.00	1.07	1.10	1.00	1.15	1.17
fir	1.22	1.33	1.46	1.28	1.44	1.51	1.37	1.44	1.54	1.48	1.55	1.94
iir	1.13	1.29	1.10	1.20	1.32	1.13	1.27	1.37	1.18	1.32	1.39	1.32
lat	1.04	1.48	1.40	1.23	1.53	1.43	1.32	1.53	1.52	1.41	1.53	1.53
lms	1.13	1.15	1.19	1.20	1.22	1.22	1.31	1.33	1.29	1.42	1.44	1.40
mul	1.05	1.54	1.85	1.26	1.89	1.88	1.48	1.89	1.90	1.69	1.92	1.93
adpcm	1.08	1.24	1.27	1.17	1.33	1.31	1.24	1.36	1.35	1.32	1.41	1.44
com	1.11	1.34	1.50	1.22	1.59	1.63	1.27	1.62	1.69	1.60	1.70	1.74
edg	1.10	1.11	1.20	1.21	1.16	1.25	1.29	1.26	1.30	1.32	1.31	1.34
his	1.08	1.27	1.16	1.21	1.31	1.29	1.28	1.32	1.33	1.33	1.33	1.36
lpc	1.00	1.00	1.05	1.00	1.02	1.09	1.00	1.04	1.13	1.06	1.09	1.23
spe	1.00	1.04	1.01	1.00	1.09	1.01	1.00	1.10	1.01	1.00	1.12	1.04
AVG	1.08	1.24	1.27	1.17	**1.33**	1.31	1.24	1.36	1.35	**1.32**	1.41	1.44

Figure 8. Speedups up achieved by random search (R), random + Markov learnt model (M), random + IID learnt model (I) after 2, 5, 10 and 50 evaluations on each benchmark on the AMD processor. Random + Markov learnt model achieves greater average performance (1.33) after 5 evaluations than random does after 50 evaluations (1.32)

References

[1] L. Almagor, K.D. Cooper, A. Grosul, T. J. Harvey, S. W. Reeves, D. Subramanian, L. Torczon and T. Waterman: Finding effective compilation sequences In *LCTES 2004: 231-239*

[2] C. Bishop, Neural Networks for Pattern Recognition, OUP, 2005

[3] F. Bodin, T. Kisuki, P.M.W. Knijnenburg, M.F.P. O'Boyle, and E. Rohou. Iterative Compilation in a Non-Linear Optimisation Space. *Workshop on Profile Directed Feedback-Compilation*, PACT'98, October 1998.

[4] J. Cavazos and J. E.B. Moss, Inducing Heuristics to Decide Whether to Schedule, In *ACM PLDI*, May 2004.

[5] K. Chow and Y. Wu. Feedback-directed selection and characterization of compiler optimizations. In *(FDDO-4)*, December 2001.

[6] K. D. Cooper, A. Grosul, T.J. Harvey, S. Reeves, D. Subramanian, L. Torczon, and T. Waterman. Searching for compilation sequences. Rice technical report, 2005.

[7] K. D. Cooper, A. Grosul, T.J. Harvey, S. Reeves, D. Subramanian, L. Torczon, and T. Waterman. ACME: adaptive compilation made efficient. In *ACM LCTES*, Chicago, IL, 2005.

[8] B. Franke and M.F.P. O'Boyle, J. Thomson and G. Fursin. Probabilistic Source-Level Optimisation of Embedded Programs In *ACM LCTES* 2005.

[9] E.F. Granston and A. Holler. Automatic recommendation of compiler options. In *(FDDO-4)*, December 2001.

[10] M. Hall, L. Anderson, S. Amarasinghe, B. Murphy, S.W. Liao, E. Bugnion, M. and Lam. Maximizing multiprocessor performance with the SUIF compiler. *IEEE Computer*, **29**(12), 84–89, 1999

[11] P. Kulkarni, W. Zhao, H. Moon, K. Cho, D. Whalley, J. Davidson, M. Bailey, Y. Park, and K. Gallivan. Finding effective optimization phase sequences. In *ACM LCTES*, June 2003.

[12] C.Lattner and V.Adve, LLVM: a compilation framework for lifelong program analysis & transformation, In *CGO*, 2004.

[13] C. Lee. UTDSP benchmark suite. http://www.eecg.toronto.edu/˜corinna/DSP/infrastructure/UTDSP.html, 1998.

[14] S.Liao, S. Devadas, K. Keutzer, A. Tjiang and A. Wang Optimization Techniques for Embedded DSP Micro-processors In *DAC*, 1995.

[15] A.Monsifrot, F.Bodin and R.Quiniou, A machine learning approach to automatic production of compiler heuristics, In *International Conference on Artificial Intelligence: Methodology, Systems, Applications*, 2002.

[16] K. Yotov, X.Li, G.Ren, M.Cibulskis, G. DeJong, M.Garzarn, D.Padua, K.Pingali, P.Stodghill, and P. Wu. A Comparison of Empirical and Model-driven Optimization. In PLDI 2003

[17] M. Saghir, P. Chow, and C. Lee. A comparison of traditional and VLIW DSP architecture for compiled DSP applications. In *CASES '98*, Washington, DC, USA, 1998.

[18] M. Stephenson, S. Amarasinghe, M. Martin and U-M. O'Reilly Meta Optimization: Improving Compiler Heuristics with Machine Learning In *PLDI* 2003.

[19] B. Su, J. Wang, and A. Esguerra. Source-level loop optimization for DSP code generation. In *Proceedings of 1999 IEEE International Conference on Acoustic, Speech and Signal Processing (ICASSP '99)*, volume 4, pages 2155–2158, Phoenix, AZ, 1999.

[20] S. Triantafyllis, M. Vachharajani, N. Vachharajani, and D. I. August Compiler Optimization-Space Exploration In *CGO* March 2003.