Using Machine Learning to Automate Compiler Optimisation

John D. Thomson

Doctor of Philosophy

Institute of Computing Systems Architecture

School of Informatics

University of Edinburgh

2008

Abstract

Many optimisations in modern compilers have been traditionally based around using analysis to examine certain aspects of the code; the compiler heuristics then make a decision based on this information as to what to optimise, where to optimise and to what extent to optimise. The exact contents of these heuristics have been carefully tuned by experts, using their experience, as well as analytical tools, to produce solid performance.

This work proposes an alternative approach – that of using proper statistical analysis to drive these optimisation goals instead of human intuition, through the use of machine learning.

This work shows how, by using a probabilistic search of the optimisation space, we can achieve a significant speedup over the baseline compiler with the highest optimisation settings, on a number of different processor architectures.

Additionally, there follows a further methodology for speeding up this search by being able to transfer our knowledge of one program to another. This thesis shows 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 learned off-line knowledge. Employing this method, we are able to gain the same results in terms of performance, reducing the time taken by an order of magnitude.

Finally, it is demonstrated how statistical analysis of programs allows us to learn additional important optimisation information, purely by examining the features alone. By incorporating this additional information into our model, we show how good results can be achieved in just one compilation.

This work is tested on real hardware, for both the embedded and general purpose domain, showing its wide applicability.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*John D. Thomson*)

Contents

Chapter 1

Introduction

Automation has been a defining characteristic of the latter 20th century – we have readily accepted the assistance of machines and computers in our daily lives, allowing them to perform tasks that humans could not accomplish at such speed. Indeed, a traditional *compiler* is in itself an example of automation, providing a means of translating high level instructions into lower level instructions with a pace that has allowed the software industry to flourish. However, the construction of the compiler itself is a conspicuously manual task, often involving substantial effort to build an initial prototype, and even more to create the calibre of optimisation engine which generates the quality of code expected today.

This thesis investigates a means to significantly accelerate the process of creating a good optimising compiler by *automatically learning* a good optimisation strategy. In addition, it shows how such an compiler can substantially outperform a manually tuned compiler over a number of benchmark suites.

1.1 The Problem

Time-to-market

Time-to-market is now a significant driving force in processor design/manufacture, particularly in embedded systems. The stalling of the increase in clock rates due to transistor shrinkage has forced architects to explore more elaborate design strategies in order to preserve Moore's Law. As microprocessors are becoming increasingly complex [28], compilers are finding it harder to keep pace, and even more difficult to obtain good performance.

At the same time, coding in assembly is slow and labourious, and can delay an embedded systems project from design to market; thus we have increasing demand for compiled code, with the compiler having decreasing ability to exploit the processor. Further, the lack of a good optimising compiler poses a challenge for the architect during development, and can hamper the evaluation of architectures for compiled code.

There is a clear need for compilers which can provide quality code for new and emerging platforms as soon as they are released.

Compiler performance

Frequently, optimising compilers are faced with difficult decisions as to which optimisations to apply and in which order; taken together with the multitude of other extremely taxing tasks the compiler must perform such as code scheduling and register allocation (which themseves may all produce subtle interactions between each other) the task faced by an optimising compiler is indeed vast [14, 26]. Traditional compilers rely on manually written heuristics to counter this huge optimisation problem, usually with poor results[47].

There has been significant research interest in improving the performance of optimising compilers for embedded systems, e.g. [41]. Such work largely 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, and is usually specific to an architecture.

There exists an unwanted gap in performance between compiled code and hand-written code. Improving the performance of code on an embedded processor could result in a reduction in clock speed, and thus power consumption, or could lead to less expensive hardware being used.

1.2 Contributions

This thesis presents methods to increase the performance of compiled code by replacing hand-tuned or arbitrary compiler heuristics with statistically derived *machine learning* techniques. In addition, these methods do not require many hours of experts' time to tune, and can be simply and quickly regenerated for new architectures with excellent results.

Firstly, a probabilistic scheme for optimising embedded systems is presented, which takes the idea of iterative compilation and extends it, based on runtime feedback, by statistically determining which optimisations, and in which order, provide good performance for an embedded program. It uses this information as the basis for a probabilistic search of the optimisation space, concentrating the search in known good areas in order to gain further performance improvements in a capped number of iterations.

Secondly, prior experience of the effects of compiler optimisation on previously seen programs is captured and used to greatly reduce the number of evaluations necessary to gain good performance. Capturing program characteristics as *program features* (see in section 4.3) and using statistical analysis allows this scheme to achieve results an order of magnitude faster than previous work.

Finally, a third scheme is proposed which dispenses with search altogether, and provides immediate performance improvement without inconvenience. This technique uses *unsupervised learning* to train the system on a larger number of programs than was possible previously, allowing better characterisation of the program space by statistical techniques, and ultimately, better performance in one evaluation.

1.3 Structure

This thesis is structured as follows: chapter 2 describes work related to both search based compiler techniques, where multiple compilations are performed, and non-search based techniques which have only one evaluation. Work relating to library generation and choosing from a selection of heuristics is also described. Chapter 3 gives a summary of the tools and infrastructure which were used to carry out the experiments in this thesis. Chapter 4 provides an introduction to machine learning from a compiler perspective, and explains the techniques used in this thesis, and by others. Chapter 5 describes a probabilistic iterative search method for improving single program performance. In chapter 6, code features are employed to use previous experience of similar programs to prime the search, and in chapter 7, an unsupervised clustering-based approach is proposed which gives a better characterisation of the optimisation space, and allows the number of evaluations to be reduced to one, thus eliminating search entirely. Chapter 8 concludes the thesis, presenting a summary of the work achieved, an evaluation of the work, a critical analysis and a look ahead to possible directions for future work.

Chapter 2

Related Work

This chapter provides a summary of related work in the area of machine learning techniques. Section 2.1 discusses search-based compiler techniques for compilers and library generation, and section 2.2 details techniques based on modelling the optimisation space, using preditive models to predict performance and guide compilation, both using *supervised learning*. Section 2.3 details work which uses an *unsupervised* technique called *clustering* to examine benchmark suites.

Supervised Learning

Supervised learning is a term which includes a large number of learning methodologies, all of which rely on knowing the correct output for given inputs in the training data *a priori*. That is to say that a learning algorithm takes each training input pattern and produces an expected output – that expected output is then compared to the known correct output, the differences recorded and the learning system updated to try to minimise these differences. The most well-known example of a supervised learning technique is *back-propagation* in *artificial neural networks* (see section 4.5.1.3).

This section deals with supervised learning techniques. Section 2.1.1 discusses compiler-based search techniques for the purpose of gaining the best possible optimisation of code, and section 2.1.2 shows domain specific application of search in the field of library generation. Section 2.2.1 shows how predictive modelling has been used to predict the performance of a program without needing to run it, and section 2.2.2 shows its use in predicting the best way to optimise, without search.

2.1 Search-based techniques

Intelligent search-based techniques can be thought of as a specialised example of on-line supervised learning, in which the search strategy is updated during the search. They traverse an optimisation space, evaluating points in that space and attempting to find the best result. In this case, compiler transformations are evaluated. The space can be searched, and if structure can be observed, then previous results can be used to determine where in the space is most profitable to search. Simple examples of this are *hill-climbers* and *greedy algorithms*.

2.1.1 Compiler techniques

Work in this section focuses on searching the space of potential transformations at compile time, in order to produce the best performance. This is an extension of *iterative compilation*, first used by Bodin et al.[7] and Kisuki et al.[47].

Iterative compilation (as proposed) is the random searching of the compiler optimisation space for a particular program, evaluating as many points as possible within a constrained time. The evaluation consists of simply compiling the code with a given set of optimisations, executing the binary and recording the runtime. The optimisations which provide the fastest runtime are considered the best for the particular program being compiled.

Bernstein et al. (1989)

One of the earliest pieces of work in this area is Bernstein et al. [5]. They used three different heuristics, one after the other, to the problem of choosing a register to spill. By measuring the results with a cost function, they could determine which produced the best result. This is an example of a very limited, exhaustive search. Moss, Cavazos et al. [44] tackled the problem of register allocation by applying supervised learning to this naïve approach, using features of the code to determine which heuristic to employ, and thus saving the time of running and performing a cost function on all three.

Kulkarni et al. (2003)

Kulkarni et al. [36] use genetic algorithms and their VISTA optimisation framework with a compiler based on VPO (Very Portable Optimizer) to attempt to effectively search the space of possible transformations, using iterative compilation. They report on two different approaches: one which reduces the search time by 65% on average and another which reduced the number of generations by 68%. These goals seem so similar as to be almost identical.

Optimisations are performed at a low level, on a RTL (register transfer lists) representation. Optimising a single function with this approach takes around ten minutes, with applications taking hours or days. The vast majority of this time is spend compiling and linking the code rather than applying the transformations, suggesting either this low level application of transformations is very efficient, or the compiler used takes a long time to run.

Several techniques are employed to help cut the overall compile time. Firstly, a hash table of all previous runs is kept. If the genetic algorithm happens to chose a sequence which has already been tested, then there is no need to rerun that sequence. A second hash table is kept which tracks all the effective transformations, disregarding those which have no effect on the code. They use Cyclic Redundancy Checks (CRC) on the RTL representation. Data is only gathered using general purpose CPUs, and thus may not be applicable to the embedded domain.

Triantafyllis, Vachharajani and August (2003)

The authors consider the case of iterative compilation for general purpose compilers [61]. The long compilation times which might be acceptable in the world of embedded systems are not so in the general purpose world. This paper provides a system called Optimization-Space Exploration which is intended to dramatically cut the time spent compiling.

Programs are separated into three classes, with iterative compilation being used on each class to find the best sequence possible. Each of these classes is split into a further three subclasses, which are also searched for the best sequence. This happens one more time, meaning a three layer tree is built.

When a novel program is compiled, the top three elements of the tree are used to provide three transformation sequences. These are tried on the new program, and the best element selected. The child nodes of this element are then used for the same process, with again one of them being selected and the child nodes used. When this process is finished, the best sequence obtained is chosen. This is an attempt to use the knowledge built offline using iterative compilation to dramatically cut the search to just nine runs. This is likely to work well if the programs in each class are similar, but this is hard to judge. The authors contend that program features are not sufficiently informative for this process and so group programs by a more arbitrary method.

Cooper et al, (2004)

Cooper et al. [15] have performed a study into the effect of sequences of optimisations on a program. The order in which transformations are applied can make a huge difference to the quality of code produced. A certain transformation may allow another transformation to work more effectively afterwards, or instead may impede another, or indeed both. Add to that the fact that this can equally apply to groups of transformations and that the number of transformations which could be applied is unbounded in length, and the optimisation space for this problem becomes massive. In this paper, the authors explore a subsection of the space exhaustively (16 transformations of up to length 4) in order to try to characterise the full space, and employ a number of search techniques to try to find the best possible sequence.

They report that 80% of the local minima in the space are within 10% of the global optimum. Such a landscape would seem to allow an easy search of the space to get a good answer, however, this is something that has generally been found to be difficult in the past. For the cost of 200-4550 compilations, an improvement of 15-25% can be obtained over the compiler baseline. Search algorithms used are a simple hill climber, a greedy constructive algorithm and a genetic algorithm. The greedy algorithm, as one might expect, performs best over a small number of sequences, however the genetic algorithm does slight better if it is allowed to run for the maximum amount of iterations (4550).

The graphs of the exhaustive space show that the question of transformation order is a complex one indeed. The space is filled with local minima, and appears to be without any obvious structure.

Bennett et al. (2007)

This paper [4] combines a probabilistic iterative search for the best transformations to apply to a program with an automatic exploration of processor design, though the use of instruction set extensions (ISE). Bennett et al. (2007) not only show significant benefit in performing these tasks independently of one another, but also that considerable additional performance can be gained by considering the two in a combined optimisation space.

Compiler optimisations are performed at the source level using SUIF1, and a tool called lp_solve built into the compiler from CoSy. This uses data-flow-graph templates and basic block knowledge to generate a set of candidate IDE templates for each program, which can be searched through.

Using a simulator configured to a Intel XScale PXA270 processor, which has configurable extensions, and the UTDSP and STU-RT benchmark suites, an average speedup of 1.47 was obtained using the full search – this compares to only 1.09 for instruction set explorations only, and 1.35 for compiler transformations only. Thus it is clear that these two values cannot be optimised independently; however, it is not clear just how complex this problem is.

Intuitively, combining two interacting, already complex problems in an optimisation space creates an even more complex problem. Though good results are presented here, it is unknown as to whether more sophisticated machine learning approaches dealing with non-linear systems may show more utility.

2.1.2 Library Genereration

Using well-optimised libraries for often used pieces of code is a simple way of speeding up program execution. Well understood and computationally intensive filters like the fast Fourier transform and matrix multiplication are commonly implemented in libraries. Since code is libraries is executed so often, and is usually deemed to be critical for performance, significant time can be spent in optimising them.

The works in this section are application/domain specific examples of search techniques, used to optimise library code.

Whaley et al. (2001)

ATLAS [64] is a tool for automatically generating extremely efficient BLAS libraries for particular processors and applications using empirical search. Consisting of a generator search module and a multiple implementations search module, the tool attempts to search across as wide an optimisation space as possible to produce a fast library. This is at the cost of an extremely long search time – however, this can be amortised over many uses and a long period of time for very heavily used library kernels such as matrix multiplication.

The generator module consists of a code generator which receives input parameters, searches, and produces a kernel as output. The multiple implementation module then searches through hand-written codes for the particular application, and ATLAS selects the better of the final options provided by these two approaches. Although ATLAS is not a restructuring compiler, it does share many of the characteristic of the same, and is one of the first good implementations of stochastic search in this field, though it does rely on hand-tuned code for much of its speed.

SPIRAL

The SPIRAL project [52] is the result of a collaboration of a number of research groups, but it primarily based out of Carnegie Mellon University and the University of Illinois at Urbana-Champaign. The objective is to create a library of platform-tuned code for various different DSP architectures which implement most well known and commonly used signal processing algorithms.

SPIRAL uses its own language (SPL) to represent the algorithms using mathematical formulas and then uses this to generate code which implements these algorithms. Optimisations are performed over a more mathematically based than usual intermediate representation, using a feedback-driven approach with a Markov decision process combined with reinforcement learning. The SPIRAL project is similar to the FFTW project [24] (Fastest Fourier Transform in the West) in that it uses an intelligent search strategy to attempt to find the best implementation of a signal processing algorithm, however, FFTW is much more limited in terms of looking at other algorithms as their implementation is very closely tied to FFT whereas the SPIRAL approach is generalisable to any algorithm which can be represented in their SPL language.

Epshteyn et al. (2005)

This work [20] takes two elements of previous work and combines them effectively for the purpose of fast library generation. Firstly, the approach taken by the ATLAS team, involving constructing an accurate empirical model of a processor and application couplet, taking a very long time to generate useful results, and a online search based technique, similar to that employed by Cooper et al. [15].

Epshteyn et al. use *Active Learning* to achieve a much faster search than ATLAS. An initial search point is analysed and its information gathered and stored. The next search point to be evaluated is then, rather than being determined randomly or probabilistically as in [15] and [22], instead the search space is evaluated to determine which point contains the most information pertinent to the model being built, which has not already been amalgamated into the model. This process can only occur online and not *a priori*.

The results are presented for a SGI R12000 MIPS-based processor, a Sun UltraSPARC III and a Intel Pentium III processor, showing a improvement in speed of library generation of 3 to 4 times the speed of ATLAS for similarly performing libraries.

2.2 Predictive Modelling

Predictive modelling techniques use features (see section 4.3) to attempt to characterise an optimisation space. Using known correct points or explorative search, a model can be built which is a projected estimation of the real optimisation space. The model can then be used to predict the best points in the space. This approach differs from that of search because prior knowledge is used to predict results.

2.2.1 Performance Prediction

Evaluating the performance of an embedded architecture can be a lengthy process. Embedded processors generally have long runtimes compared to general purpose processors, and sometimes have incompatible toolchains and libraries that make running benchmarks difficult. Additionally, the development of architectures is limited by difficulty in evaluation. Since no hardware exists, simulators must be used, which are usually slow, and take a long time to produce.

Performance prediction attempts to address these problems by predicting the runtime of a benchmark, without actually running it.

Ipek et al. (2006)

The first work in this field came from research on hardware. Ipek et al. [34] proposed an automatic system to drastically speed up hardware design-space exploration (exploring values such as memory latency, cache size ,etc.). Although this work is in the hardware field, it is possible that the same techniques may be applicable to the realm of compiler optimisation.

Using an artificial neural network (ANN) model (see section 4.5.1.3), they are able to predict the result of variations in the hardware parameters, with excellent results. With the model considering only 5% of the design space, it can predict performance to within 2% error of the true value. In this circumstance, it seems unlikely to be easily applicable to software world as anything which causes a change in the program being run, such as a program transformation, causes the system to require retraining - very costly in time. However, it may be of interest at a conceptual level.

Cavazos et al. (2006)

Cavazos et al. [11] employ a similar approach for a compiler. They describe a system for predicting the performance of a new program on a known and previously explored architecture. The technique is shown to be effective on two different embedded architectures – a MIPS-based AMD Alchemy processor and a VLIW processor, the Texas Instruments C6713 floating point DSP. This attributes a degree of generality to the technique, that it may be applied to other processors too. Initially, a model of the processor is built, using 640 training runs from a set of 10 benchmarks. The model used is a standard feed-forward, back-propagation artificial neural network. These training runs are randomly taken from a space of possible versions of a program, post transformation. A total of 5 different transformations are considered, giving a space containing 88000 points.

In addition to this, when a new program is to be evaluated, a further 4 probing runs are made on this new program. These are used to characterise the new program, and give input to the existing model, which provides a predicted execution time as output.

The system used to characterise each program is based on ‘reactions’. This means that instead of deriving information from a features-based approach where the source code is analysed for attributes considered interesting by experts, the information is derived from comparing the performance of the program when known transformations are applied to how other programs in the training set have performed when the same transformations were applied to them.

The selection of which transformations to use to discriminate best between training programs (called canonical transformations) is produced by a formal system of information theory, designed to reduce the redundancy in the data. This approach is interesting as it removes the human element from the system – it is very difficult for a compiler expert to predict what features might be useful in building a good performance model. Indeed, it is the critical task as no amount of clever post-hoc analysis can produce a good solution when the original features used are of poor quality. It is for this reason that this paper is of particular interest, as it is the first to model a program in this way.

However, although this ‘reactions’ based approach does dispense with the human element, it does simplify the feedback available to the model to a difference in execution time. The authors argue that this is all that is necessary as it intrinsically includes the more complicated hidden information in this simple value by virtue of this information being explicitly actualised in the running of the program, yet this seems difficult to ascertain with any degree of certainty, given the variable nature of the quality of features.

If this technique performs well against features, as it is shown to in the paper, it is quite possible that the features were of poor quality, and a better set would have performed differently. Unfortunately, the quality of features is notoriously difficult to quantify. Given the extremely complicated, non-linear optimisation space which has been shown to be present in this kind of problem, it seems unlikely that such a simple approach is sufficient, although certainly helpful.

Dubach et al. (2007)

Dubach et al. [18] take an alternative approach to speeding up learning. Instead of building models of programs and predicting which optimisation would be suitable for a program, they model how programs perform on a specific processor, then generate a prediction of the runtime of new programs supplied to the system, without actually running them on real hardware.

This performance predictor can then be used in conjunction with iterative compilation techniques to improve their performance. The predictor is orders of magnitude faster than physically running the program on real hardware, and thus addresses the bottleneck of iterative compilation - the long time necessary to obtain good results; using the predictor instead of real hardware allows many more runs of a program to be made and better results to be obtained.

This work uses code features derived from source to express a description of the programs, and an artificial neural network (ANN) model. They report that using an input of 16 samples to the model, they are able to achieve a correlation coefficient of 0.65 to the actual results, which rises to 0.8 when 128 samples are allowed.

The process of building a predictor of performance is subtly different to that of a predictor of which optimisations to apply to a program, yet the models generated must be very similar. On an abstract level, the end result of each process (if one assumes iterative compilation is coupled with the performance predictor) is the same, yet the processes are obviously different in methodology. This raises the question as to what different information is being stored in these two approaches; if one approach is intrinsically better than the other, then one would expect that the better approach is that for which the a model can best express the information being stored within.

2.2.2 Predicting the best optimisation

Modelling has also been used to predict the best way to optimise a program. This involves analysing a new program, then predicting the set of optimisation options which provide the best performance.

Lagoudakis and Littman (2000)

A number of different heuristics have been proposed for the problem of register allocation, and different heuristics are known to perform better in different context. Selecting the correct heuristic for different contexts has been shown to benefit a performance by Cavazos and Moss [9].

Rather than using an intelligent machine learning technique to replace a hand-coded heuristic, it can be much easier to use machine learning simply to choose between several well-known heuristics for a particular purpose. The advantage of this, other than its simplicity, is that these heuristics are thoroughly tested, trusted and understood by compiler writers and it is therefore much easier to integrate such an approach into a production compiler than a more sophisticated technique.

Work in the area of heuristic selection, but in a different context is presented by Lagoudakis and Littman [38] at Stanford. They created a system based on features to select an algorithm for the abstract problems of order statistics selection and standard sorting. Order statistics selection is, given an unsorted array of numbers, find the nth element if the array were sorted in any given order, where n is any valid index of the array. They were able to beat the two best standard algorithms for this procedure, deterministic select and heap select, by forming a hybrid algorithm which chose between the two. They applied a similar approach to standard numerical sorting by using a hybrid of quicksort and insert sort.

Monsifrot et al. (2002)

Monsifrot et al. [43] contend with the problem of loop unrolling heuristics on both the superscalar UltraSPARC and the VLIW-esque Itanium64. Instead of a manually written heuristic, an automatically derived one is proposed, based on decision trees. Here only the question of whether to unroll or not is considered, leaving the unroll factor to the underlying compiler. Loops are gathered from unspecified programs written in FORTRAN-77, and a heuristic is generated based on features such as decision trees are used (OC1 software), which involves splitting a set of objects in a hyperspace over and over until every object in a substance belong to the same class - that is , to unroll or not to unroll.

Results are presented using the Open Research Compiler for Intel Itanium64 and Sun UltraSPARC. On average the execution time is reduced to 93.8% on IA-64 and 96% on UltraSPARC of the baseline. Interestingly, if the decision trees for the two processors are swapped, the performance benefit is reduced considerably, furthering the case that separate models are needed for different processor configurations to gain optimal performance.

Cavazos and Moss (2004)

The authors examine the problem of when it is profitable to apply an optimisation, in this case instruction scheduling, to a program in a just-in-time environment [9]. Although instruction scheduling in particular is examined, there is no reason why the technique demonstrated here could not be put to use for other optimisations. The language used is Java, and the compiler, JIKES RVM. The authors use list scheduling over basic blocks, using the critical path scheduling model, although they note that the type of scheduler used is not important.

Within a just-in-time environment, it is always necessary to weigh the cost of optimisations which may make the code run faster against the actual speedup likely to be gained from such optimisations. The scheduling optimisation can significantly improve the running time of a program, but it also an expensive optimisation to run in terms of compilation time it would therefore be useful to be able to determine which basic blocks particularly benefit from scheduling and thus apply it to only these blocks. This is the question which Cavazos and Moss attempt to answer in this paper, by employing machine learning.

The authors note that the just-in-time environment severely restrains what kind of techniques can be used to decide whether to schedule, as this process in itself adds to the compilation time. It is therefore necessary to select a method which is inexpensive both in terms of computational complexity and using a set of features which are cheap to obtain at runtime. Thus, any features based on the dependence graph of the block would be unsuitable as the DAG itself would be expensive to calculate. Instead, all possible instructions were classified into twelve categories, each which in the opinion of the authors, have similar scheduling properties. The features used therefore were simply the percentages of each type of instruction within the basic block.

Using rule set induction provided by the ‘Ripper’ tool, what are effectively decision trees are created for a binary classification problem. The learning is supervised, done using a training set to which the answer as to whether to schedule or not has been manually determined.

Results show that it is fairly rare that scheduling is effective within this just-in-time environment – the authors argue this makes it all the more necessary for a cheap heuristic to determine whether to schedule or not, however, they do not directly compare how well this system compares to simply never scheduling at all. The classification accuracy is impressive however, with over 90% of the improvement of scheduling the whole program being obtained with only 25% of the time this would take.

Stephenson and Amarasinghe (2005)

Stephenson and Amarasinghe [57] use two simple statistical techniques to try to predict the correct unroll factor for the high-level loop unrolling optimisation on a per loop basis. Loop unrolling is one of the most important high-level optimisations as it not only removes some control flow overhead, but also allows the compiler greater scope for gaining instruction level parallelism, as well as allowing further optimisations to take place on the loop. In this paper, the authors report a 5% overall improvement on the SPEC 2000 benchmark suite using these techniques, with a prediction accuracy of 65% for all loops in these benchmarks. Since these numbers are not particularly impressive, the paper concentrates on the time which could be saved by employing machine learning techniques for this and similar problems rather than employing a large number of expensive compiler writers. These experiments were performed on the Intel Itanium2 architecture and using the Open Research Compiler[46]. Unroll factors between 1 and 8 were considered.

In order to employ either of the techniques used in this paper, features must be determined and extracted. As is common, the authors first produced 38 features for consideration – far too many for these simple linear techniques to handle. A much smaller subset of features were selected from this original set by two different means: Mutual Information Scoring and a greedy selection algorithm. Mutual Information Scoring is a method of ranking how much uncertainty can be removed from the overall result (in this case the loop unroll factor) by knowing the value of a particular feature. As is pointed out, this method has the problem that interactions between features (how much of the uncertainty that is removed by feature 2 has already been removed by feature 1) are not considered. The greedy algorithm works by taking the best performing feature using a given classifier, then combining it with the second best feature, the third best feature and so on.

The five features selected using the greedy algorithm for nearest neighbours were: number of operands, live range size, critical path length, number of operations and known tripcount. For the support vector machine, they were number of floating point operations, loop nest level, number of operands, number of branches and number of memory operations. Two different classifiers are used:

Nearest neighbours is a well known, statistical technique for classifying phenomena based on available features. In this case, classifying unroll factor from some feature vector determined by the authors. Training involves mapping each input vector to a point in some n-dimensional space (where n is the number of features) whose correct classification is known *a priori*. Novel input vectors are classified by mapping them to this space, then calculating the Euclidean distance between the new point and all the training points. The shortest distance is found and the novel input is classified according to the previously determined class of the nearest training point. A confidence factor can be determined by considering the closest k neighbours and comparing their class.

It is obvious that this technique increases significantly in complexity with the number of features used and the number of training points allowed.

Secondly, Support Vector Machines (SVM) are used. A traditional SVM splits the data into two classes by constructing a maximum-margin hyperplane (the distance between the closest examples to the hyperplane is maximised) such a hyperplane is derived by solving a quadratic programming problem. This can be modified with some difficulty to accommodate multiple classes as is used in this paper.

Unsupervised Learning

Unsupervised learning does not involve evaluating any points in the optimisation space. Instead, it seeks to discover some structure in the input (feature) space or model the probability distribution of the input data. This allows us to characterise the space more quickly by using a smaller number of points, representative of the space in general.

2.3 Clustering

Clustering is an unsupervised learning technique which seeks structure in the input data, finding clusters of input points which broadly share similar features. From this, it may be possible to classify the input data into sets, according to their proximity to each cluster in feature space.

Joshi et al. (2006)

Joshi et al. examine benchmark similarity in MediaBench, MiBench and SPEC CPU2000 benchmarks. The purpose is to reduce the time needed to evaluate a system using the benchmark suites, and argues that only a subset need to be executed and profiled in order to effectively estimate the average IPC, data cache miss rate and speedup of the whole benchmark suite, when varying the system and processor.

Additionally, they evaluate the four generations of the SPEC CPU benchmark series to determine how much changes between each generation. They conclude that temporal data locality gets progressively worse through the iterations of SPEC CPU, however the inherent program characteristics stay the same.

They accomplish this by using clustering on a rich feature space derived from simulation using a custom tool called SCOPE, which is a derivative of the SimpleScalar v3.0 simulator. Features include instruction mix, control flow behaviour statistics such as basic block size, branch direction, fraction of branches taked and fraction of forward-taken branches, as well as register dependency distance, data temporal locality, data spatial locality, and instruction locality.

This work has interesting implications for anyone wanting to evaluate a new processor, but has not enough time to run the whole benchmark suite. However, the work lacks anything to compare the chosen clusters to, and therefore it is hard to say if the clustering technique is is better than a naïve selection process. Additionally, the work is intended to assist in estimating the performance of the benchmarks evaluated in the paper, and it is not possible to gain a similar benchmark subset on a different benchmark suite, without running the whole suite through a slow simulator and profiler.

2.4 Summary

This chapter has described the related work in the area of and machine learning in compiler optimisation. Firstly, techniques involving supervised learning were described, including techniques which search the optimisation space, and those which model the optimisation space to predict a good answer. This includes library generation, performance prediction, and performance maximisation. Secondly, an unsupervised approach was presented, using clustering to represent the optimisation space.

Chapter 3

Infrastructure

This chapter describes the infrastructure that facilitated the research in this work – the two primary compilers, SUIF and GCC, and the optimisation tools which were used to drive them. Section 3.1 details the benchmarks used in this thesis, where section 3.2 does the same for platforms. Finally, the tools used are discussed in section 3.3.

Using machine learning with compilers requires extremely robust infrastructure. Significant training is often required, and this demands a large quantity of data. Generating this data is only practically possible by using infrastructure capable of automatically compiling and running thousands of programs without human prompting.

Additionally, using many different transformation sequences and optimisations stresses the compiler in a way not usually tested for. This entails using options which may never have been used before in a particular combination, and thus cause even a production-level compiler to crash or do something unexpected. Therefore, it is important to have robust infrastructure which can detect and compensate for these issues.

3.1 Benchmarks

Benchmark suites are collections of programs designed to evaluate, and allow the comparison of, the performance of compilers and processors. They are designed to give a thorough appraisal of the system by employing commonly used coding techniques, algorithms, and real world examples to test how well it performs. Since this thesis focuses on compiler optimisation, the processors remain unchanged, allowing evaluation of the compiler. Two different benchmark suites are used in this thesis: UTDSP and EEMBCv2. These focus on programs most commonly found in the embedded domain, the primary target of this work.

3.1.1 UTDSP

UTDSP [29] is a benchmark suite created at the University of Toronto which targets DSPs. Written in C, the benchmarks are divided into the categories of kernels and applications. The kernels represent the main computation carried out in many embedded programs, such as fast Fourier transforms and matrix multiplication. The applications are composed of more complex algorithms and data structures. The details are shown in figure 3.1. Many of the programs are available in up to four coding styles (explicit vs pointer-based array references, plain vs source-level software pipelined).

3.1.2 EEMBCv2

EEMBC [19] is a commercial benchmark suite targetting embedded architectures. It is the most commonly used benchmark suite in commercial embedded systems comparison, and consists of some of the most important programs and kernels in this area.

They are divided into automotive, consumer, networking, office and telecom categories. EEMBCv2 takes the original v1 benchmarks, and adds more modern consumer and networking benchmarks, using up-to-date techniques. This is shown in figure 3.2.

The suite comes with its own test harness, which can be used to verify program output to check for inconsistencies, and additionally used to create a composite ‘EEMBC score’ of all the benchmarks when used in a formally specified way (this is not done in this thesis).

There are 55 benchmarks in total: 16 automotive, 5 consumer_v1, 3 networking_v1, 4 office, 6 telecoms, 13 consumer_v2 and 7 networking_v2.

Program	Description
fft_1024 **fft_256**	Radix-2, decimation-in-time Fast Fourier Transform (FFT)
fir_256_64 **fir_32_1**	Finite Impulse Response (FIR) filter
iir_4_64 **iir_1_1**	Infinite Impulse Response (IIR) filter
latnrm_32_64 **latnrm_8_1**	Normalised lattice filter
lmsfir_32_64 **lmsfir_8_1**	Least-mean-squared (LMS) adaptive FIR filter
mult_10_10 **mult_4_4**	Matrix multiplication
G721_encoder	ITU ADPCM speech transcoder
G721_decoder	ITU ADPCM speech decoder
V32.modem encoder	V.32 modem encoder
V32.modem decoder	V.32 modem encoder
compress	Image compression using Discrete Cosine Transform
edge_detect	Edge detection using 2D convolution and Sobel operators
histogram	Image enhancement using histogram equalisation

Figure 3.1: UTDSP benchmarks

Benchmark category	Description
automotive	Workload tests, Automotive algorithms, Signal processing
consumer_v1	Image compression and decompression, Colour filtering and conversion
networking_v1	Routelookup, packetflow monitoring
office	Beizer, Dithering, Text parsing
telecoms	Autocorrelation, FFT, iFFT, Viterbi decoder
consumer_v2	MPEG4 encode and decode, updated jpeg encode and decode
networking_v2	IP packet check, IP reassembly, QoS, TCP decoding

Figure 3.2: EEMBCv2 benchmark categories

3.2 Platforms

This thesis uses a number of platforms to evaluate this work. All platforms used are real hardware implementations of the architecture, and not simulated or implemented in an FPGA. Performance counters are used to give real-world performance numbers, which are not influenced by unquantified behaviour in the system libraries, as can happen when using simulators. Four different embedded processors and a general-purpose processor are targetted in this thesis. Additionally, a cut-down version of another general-purpose processor is used, which is being targetted at the embedded domain.

3.2.1 Analog Devices TigerSHARC

The TigerSHARC TS-101 is a high-performance embedded processor from Analog Devices. It has an internal floating point engine, as well as the ability to process 1, 8, 16 and 32 bit fixed-point, and process four 32-bit instructions per cycle. The manufacturers claim enough on-chip memory to cope with 64,000 point FFTs[2]. This platform does not use an OS, running in bare-metal mode.

3.2.2 Philips Trimedia

The Philips Trimedia (now made by NXP Semiconductors) is a multimedia, VLIW, embedded processor using the Harvard architecture. Philips claim [62] this processor can be efficiently programmed using only high-level languages, rather than traditional DSP assembly programming, and this makes it an interesting architecture for compiler evaluation. The version of the chip used in this thesis has 128 32-bit geneal purpose registers and 32KB instruction cache, 32KB data cache. This platform does not use an OS, running in bare-metal mode.

3.2.3 Intel Celeron

The Intel Celeron is a budget general-purpose processor manufactured by Intel [31]. The chip generally shares architectural features with the top-of-the-line Intel processors, but with less features/cache to save money and energy. The processor used in this thesis runs at 400MHz, with 128KB of L2 cache.

In recent years, Intel has also marketed the Celeron as an alternative embedded processor, stressing the low power consumption. This platform was used with the Linux OS, kernel version 2.4.20

3.2.4 AMD Alchemy Au1500

The AMD Alchemy Au1500 processor is a low power embedded SoC processor using the MIPS32 instruction set. The chip chip used in this work runs at 500MHz, has 16kB instruction cache and 16KB non-blocking data cache. This platform was used with the Linux OS, kernel version 2.4.23

3.2.5 Texas Instruments C6713

The TI C6713 is a 32/64-bit high-end floating point DSP, a wide clustered VLIW processor with a 4KB instruction cache and a 4KB data cache [59]. On chip there is also 64K-Byte L2 unified cache/mapped RAM and 192K-byte additional L2 mapped RAM. This platform does not use an OS, running in bare-metal mode.

3.2.6 Intel Core2Duo E6750

The Intel Core2Duo [32] is a general-purpose dual-core processor capable of running in 32-bit or 64-bit mode. The version used in this thesis runs at 2.66GHz, has 4MB of shared L2 cache and used 32-bit mode. This platform was used with the Linux OS, kernel version 2.6.24.

3.3 Compiler Tools

Two main tools are used in the course of this thesis: firstly the COLO Tool, which embodies the SUIF compiler [29], and a modified version of GCC called Milepost GCC. Milepost GCC differs from classic GCC in that the internal optimisations are exposed and can be externally driven. These tools are necessary to generate the optimisation space which this work evaluates.

3.3.1 The COLO Tool

The COLO (COmpilers that Learn to Optimise) Tool is an optimisation framework, developed at the University of Edinburgh, which drives source-to-source transformations in C and provides complete control of which transformations are applied and in which order. The framework is written in Java, and incorporates the SUIF compiler from Stanford [29] (discussed later in section 3.3.1.1) as its transformation engine.

C code enters the system and is translated into an intermediate representation on which all transformations operate. The Optimisation Engine is responsible for deciding which optimisations should be applied, and in which order; this stage is fully programmable and interchangeable, allowing a variety of different optimisation strategies to be used within the framework. Having selected a transformation schema, transformations are then applied by the Transformation Framework. After finishing the transformation process, the IR is translated back into C code and compiled into executable by any standard C compiler, depending on which platform is the desired target. The program is then executed and profiled in the Profiler, collecting execution time and code size, and the results fed back to the Optimisation Engine to allow it to update itself based on the success or failure of the chosen transformation schema.

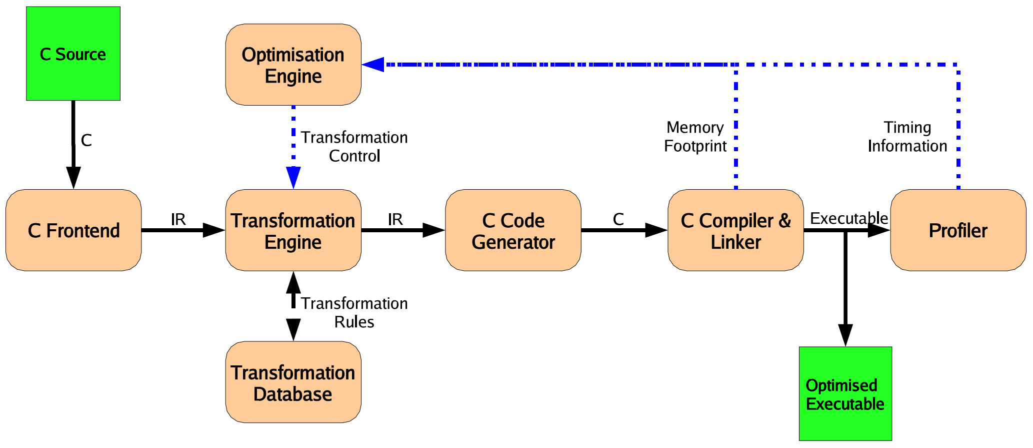


Figure 3.3: The COLO Optimisation framework Tool

Linker information such as the memory footprint of the compiled program is passed back to the optimisation engine, which decides on the further optimisation strategy based on this, and the additional timing information gathered from profiled program execution.

This process is then repeated until a set goal is reached, such as a maximum number of iterations, or desired execution time achieved. The final executable with the desired transformed code is the output of the process. This structure is depicted in figure 3.3.

3.3.1.1 The SUIF Compiler

The SUIF compiler is used as the Transformation Engine of the COLO Tool. It is a openly released research compiler developed by the SUIF group at Stanford [29]. SUIF allows independently developed compilation passes work together using specified intermediate representation called SUIFIR. It is furnished with a variety of high-level optimisations which are applied at the SUIFIR level. These transformations can be applied independently, and in any order. There are around 80 different transformations available, which are listed in Appendix A.

The compiler includes both C-89 and Fortran front-ends, but only the C front-end was used in this work. The SUIF compiler was of particular utility in this work as the IR is sufficiently high-level to allow the complete reconstruction of source code, and thus can be used as a source-to-source compiler.

The C code is first transformed into SUIFIR using the SUIF front-end, which allows the code to be transformed. Two different transformation systems are used, which can be used interchangeably: the ’porky’ SUIF stage, which allows data transformations, and a unimodular loop transformation stage, which provides classic loop transformations like unrolling and tiling. These transformations work on the SUIFIR level.

Although SUIF source-to-source transformations are powerful, using the compiler has several drawbacks. It only accepts C-89 as an input language, and thus many modern benchmarks are incompatible, or require significant time to be rewritten in C-89. Additionally, modern programs often take advantage of GNU C extensions provided by GCC, which are again incompatible with SUIF. A more robust compiler would allow more complicated programs to be evaluated, and would take less time to configure. For these reasons, a second tool is used in this thesis: Milepost GCC.

3.3.2 Milepost GCC

Milepost GCC is a modified version of the GCC compiler developed by the Milepost project [25]. This compiler does not do source-to-source optimisation, but instead is changed so that the internal optimisation phases are exposed and driveable by an external tool. The use of external tools allows sophisicated machine learning optimisation strategies to be swapped in and out easily. This is achieved using the GCC Iterative Compilation Interface, part of Milepost GCC.

3.3.2.1 GCC ICI

The GCC Interactive Compilation Interface (ICI) is an interface for controlling the internal optimisation decisions of the GCC compiler. It allows the complete substitution of default optimisation heuristics and the reordering of transformations, beyond the capabilities of command line options or pragmas. Instead, the optimisation can be

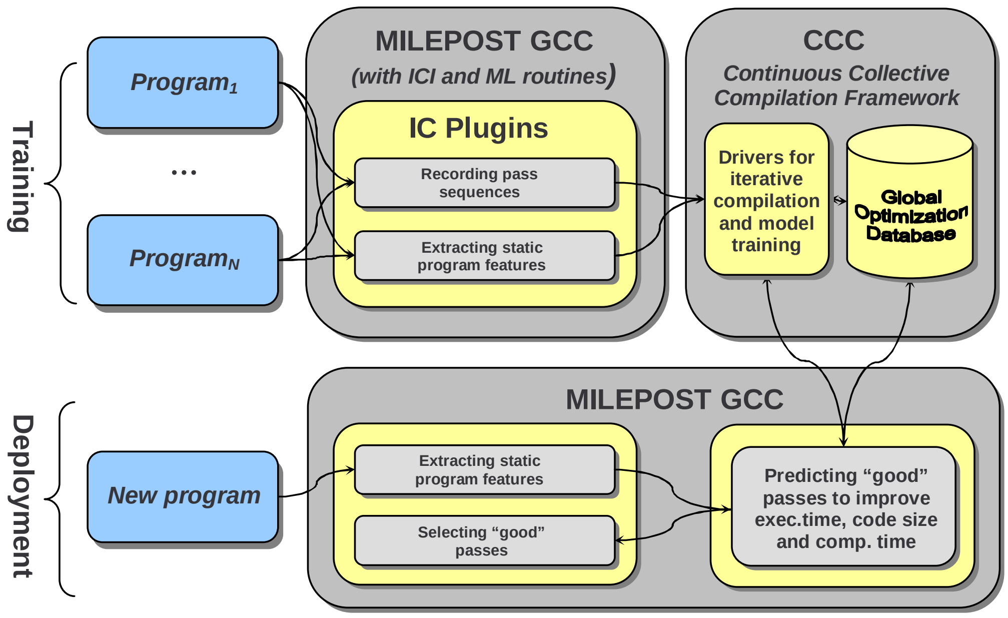

Training
Program$_1$
...
Program$_N$
MILEPOST GCC
(with ICI and ML routines)
IC Plugins
Recording pass sequences
Extracting static program features
CCC
Continuous Collective Compilation Framework
Drivers for iterative compilation and model training
Global Optimization Database
Deployment
New program
MILEPOST GCC
Extracting static program features
Selecting "good" passes
Predicting "good" passes to improve exec.time, code size and comp. time

Figure 3.4: Framework for Milepost GCC [25]

driven by shared libraries, though command line options are still available. A list of the optimisations available is found at [33].

The ICI replaces the GCC Controller (pass manager). Passes can be selected by an external plugin, choosing different optimisations than the default Controller. Additionally, the plugin can provide its own passes, implemented entirely outside GCC.

3.3.2.2 GCC CCC Optimisation Framework

The GCC Continuous Collective Compilation Framework is tool to drive compiler optimisation, particularly through the GCC ICI (interface). It contains a toolbox of techniques which allow simple interaction with internal GCC optimisations, allowing the user to automate the running of thousands of compilations with different optimisation schemas. It allows extensions so that custom optimisation selction algorithms can be implemented and used within the tool.

Figure 3.4 shows how GCC CCC and Milepost GCC interact when training and deploying a simple machine learning based compiler.

3.4 Conclusion

This chapter has detailed the infrastructure which was used in this thesis. Two embedded benchmark suites have been compiled using both SUIF, driven by the COLO Tool, and Milepost GCC. A variety of platforms have been used, including DSP-like platforms like the Analog Devices TigerSHARC, the Philips Trimedia (a VLIW processor) and a general purpose processor, the Intel Core2Duo.

Chapter 4

Introduction to Machine Learning

This chapter provides an introduction to machine learning from a compiler perspective, explains some of the general concepts in machine learning and discusses the techniques used in this thesis, and in other work. Section 4.1 discusses why machine learning is useful for compilers, section 4.2 outlines the idea of machine learning, section 4.3 explains the concept of features, section 4.4 shows how machine learning is affected by complexity, section 4.5 details some machine learning techniques, section 4.6 shows how machine learning techniques can be applied in the compiler field and section 4.7 warns of some of the problems one may encounter when employing machine learning.

4.1 Why use Machine Learning?

During the phases of optimisation and code generation, a compiler makes hundreds of decisions which impact the quality of the outputted code. Indeed, given the same input C code, two properly implemented but different compilers are extremely unlikely to produce the same output code, although the functionality of that code would be the same. This is because many of the decisions needed to be taken by the compiler are dependent upon extremely complex scenarios, where it is very hard to tell which answer would give better code, and which worse. In addition, these decisions have interacting effects, meaning that one optimisation decision which initially gives better code, may in the end result in worse code being produced.

Traditional compilers tackle these problems by using *heuristics*. These are effectively vastly simplified, hand-generated models of the system, which allow the compiler to make an optimisation decision in a very small space of time. The main reason why code outputted is often different between compilers is that different heuristics are often employed, both with different estimates as to how the spaces should be modelled. It logically follows that if different heuristics are used in different compilers, either the compiler with the best set of heuristics should be best for all code generated (which is demonstrably not the case) or that different sets of heuristics (and thus different compilers) have more beneficial effects on some pieces of code over others. This shows that heuristics are not a good solution, simply the best which has been used so far.

This is the nub of the matter – that these heuristics are often little better than educated guesses made by experienced compiler writers, whose performance can vary wildly over different code types, and have no statistical evidence in their provenance.

In some cases, such as code scheduling, very accurate heuristics have been developed which provide near perfect performance – their mapping of inputs to outputs is very near that of the oracle. For register allocation too, graph colouring heuristics (amongst others) have been used to great effect, however, firstly, these solutions took a great deal of effort to arrive at, and secondly, there exist even more complex problems which a compiler must deal with to obtain optimal results, which a human could not hope to tackle fully.

Therefore there is a need for systems which can quickly and accurately provide an answer to difficult, non-linear (see section 4.4) problems within a compiler, which can be both quickly and reliably generated, and also are based on real statistical analysis of the problem, rather than a human ad hoc view. Machine learning allows us to generate more complicated models, which more accurately represent the complexity of the problems at hand, and whose generation is based on a true scientific approach.

4.2 What is Machine Learning?

The general paradigm of computation is classically:

INPUT → PROCESS → OUTPUT

Machine Learning (ML) fits very easily into this conceptualisation. ML can be thought of as computational process used to map a set of inputs to a set of outputs, much like a mathematical function. ML is useful to us when it is not intuitively obvious what that function is, such as is the case with many decisions an optimising compiler makes. In a traditional compiler, these hard choices are either not considered at all – being mapped to a fixed number for all inputs – or else are made by manually written and tuned heuristics. ML can be used to replace these ad hoc heuristics with proper statistical analysis and modeling, which much better express the true nature of the problem.

A compiler may wish to know which transformation to apply next – this problem can be addressed using ML by representing the program as a vector of *code features* (see section 4.3) which describe the program's important characteristics and comparing this to a pre-prepared model of the system. By comparing the new program to a model which represents the learned past experience of the ML tool, the correct transformation can be selected. e.g.:

FEATURES→MODEL→TRANSFORMATIONS

Section 4.6.2 deals with how these models are constructed and section 4.5 describes how they function.

4.2.1 Supervised and Unsupervised Learning

Machine learning techniques can usually be classified into supervised or unsupervised techniques(a third general category, reinforcement learning also exists, but is not discussed in this thesis).

Supervised Learning

Supervised learning techniques rely on having *labelled data* in the training stage – that is input data, to which the correct answer is already known by some other means. A learning algorithm takes each training input pattern from the labelled data, and produces an expected output. That output is compared to the known correct output from the labelled data, and the difference calculated. The learning algorithm then attempts to change the variables within the algorithm to compensate for the error, to learn from past mistakes. The most well-known example of a supervised learning technique is *back-propagation* in *artificial neural networks* (see section 4.5.1.3). Search techniques can be thought of as a specialised form of online supervised learning, in which the search strategy is updated during the search.

Unsupervised Learning

Conversely, unsupervised learning does not use labelled data at all. Instead, it seeks to discover some structure in the input (feature) space or model the probability distribution of the input data. There is no feedback loop to allow learning from mistakes, and so incorrect classifactions do not affect these algorithms.

4.3 Program Features

One of the most vital elements in any machine learning environment is defining how the modelling technique perceives the input – in our case, how models can differentiate between programs or sections of code, and how they can gauge their similarity. We need to be able to represent our inputs (programs) in a way which is intelligible to our models – to this end, we employ *code features*.

A set of code features pertaining to a program consists of a vector of real or binary values which we hope accurately depicts the crucial characteristics of that program. The selection of good features is important to any machine learning-based technique, as without accurate and relevant inputs, a model cannot hope to produce pertinent outputs.

Initial selection of features is a matter for expert opinion, but statistical techniques can be used to help assess those features as to their relevancy and redundancy (see section 4.5.5). Some examples of features which are used in this work are the total number of adds used in a program, the proportion of multiplications or shifts, and features related to memory usage such as a count of loads and stores. A more detailed discussion of the features used in the experiments reported in this thesis is given in the relevant chapters.

4.4 Complexity in the Optimisation Space

Code optimisation within a compiler was always considered to be a difficult problem to solve, but no real evidence or analysis was produced to show quite how difficult. This is an important question as the complexity of the problem informs the type of model to be used to confront it. The use of too simple a model will lead errors due to an inability to accurately represent the true system, and too complex a model may lead to *overfitting* (see section 4.7), which results in a more complicated but inaccurate model being imposed on a simple system.

4.4.1 Linear Problems

A significant proportion of problems do not in themselves produce complex interactions. A classification problem which can be solved by a simple straight line is considered *linear*. The output of an OR logic gate given two inputs is a classical *linearly separable* problem.

It is obvious that a straight line is enough to separate both output states. In any attempt to solve the OR problem, or indeed any other linear problem, a linear modelling technique is best used, such as logistic regression or a single-layered neural network (see section 4.5.1). It should be noted that some compiler problems can be solved well with linear modelling, as is shown by Cavazos and O'Boyle[12], however simply because a linear model works well on an issue is not proof in itself that the problem is intrinsically linear – rather it may be a linearisation of a more complicated system.

4.4.2 Non-linear Problems

When a straight line is insufficient to separate classes of output, the problem can be considered *non-linear*. The XOR logic gate provides a classical non-linear problem in figure 4.1

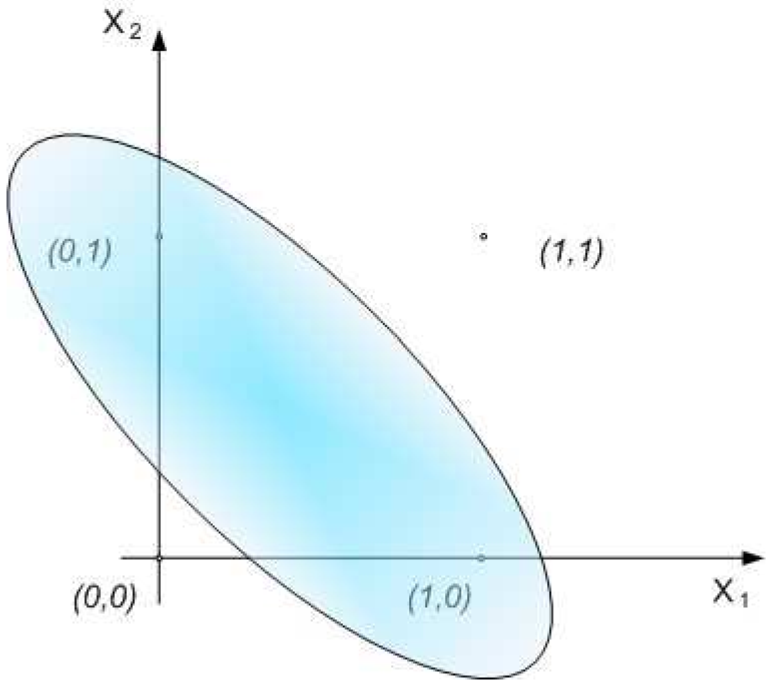


Figure 4.1: XOR diagram – inside the oval area signal on output is '1'. Outside of this area, output signal equals '0'. It is not possible to divide it by one line. [65]

Plainly, a single straight line alone cannot separate the two classes – a more complicated model is needed. Many compiler optimisation decisions fall into this category of non-linearly separable problems.

Such a system can only be accurately modelled using a non-linear technique, such as non-linear regression, Hidden Markov Models (HMM), a kernelised Support Vector Machine or Multi-Layer Perceptron, amongst others (see section 4.5.1).

4.5 Machine Learning Techniques

The established field of machine learning has produced thousands of techniques with varying complexity for a multitude of tasks. Since the use of machine learning in compilers is in its relative infancy, simple techniques are likely to produce the most success at this point. This section presents five commonly used machine learning techniques which have been used in compilers.

4.5.1 Artificial Neural Network (ANN)

An Artificial Neural Network is a model and methodology of reasoning, loosely based on a biological brain. It is comprised of a number of simple and highly connected computational units, connected by weighted links which affect the communication between one neuron and another. It is by means of these weights that an ANN is able to store information, and by changing and updating the weights in a systematic way, that it is able to 'learn'.

4.5.1.1 Computation in a neuron

A neuron receives multiple signals as input, which it makes a computation upon and then sends an output through a different channel. This output is then relayed as input to other neurons.

The task a neuron performs is determined by its *activation function*. It computes a weighed sum of inputs, and compares it to a *threshold* value, θ. If the computed input is greater than the threshold, the neuron is considered to have 'fired' and a value of 1 is outputted. If less than θ, the neuron does not fire and -1 is outputted.

Thus,

$$Y = sign \left[\sum_{i=1}^{n} x_i w_i - \theta \right]$$

This is known as a *sign function*, and is a *hard limit function*. Other activation functions may also be used, such as a *sigmoid function* ($Y = \frac{1}{1+e^{-x}}$), which is used in a standard back-propagation network.

4.5.1.2 Single layered network

A single layered neural network can be used to solve linearly separable problems. For this to occur, the neurons in the network must not only be able to store information, but also be able to update that information to converge with the data supplied to it – to ‘learn’. The model represents a *hyperplane* in an n-dimensional space which divides that space into two sections. The type of simple network described here is called a *perceptron*.

The weights in the network are first initialised to random values. The output of the neuron is then calculated for the pth iteration (in the first case, $p = 1$)We use the *sign activation function* described above:

$$Y(p) = sign\left[\sum_{i=1}^{n} x_i(p)w_i(p) - \theta\right]$$

The weighted inputs must then be updated as follows, using the *delta rule*:

$$w_i(p+1) = w_i(p) + \alpha x_i(p)e(p)$$

where $e(p)$ is the error for iteration p, defined as:

$$e(p) = Y_{desired}(p) - Y_{actual}(P)$$

and α is the learning rate. The *learning rate* specifies how quickly a network is updated in relation to the calculated error. A slower rate of learning is often advantageous as it stops the network from oscillating between points of noise in the data.

This process is then repeated for $p+1$ until the model has converged, or some other stopping criterion has been met (see section 4.7.1). The threshold θ can be changed to move the decision boundary if that is desirable.

4.5.1.3 Multi-Layer Perceptron

The *Multi-Layer Perceptron* (MLP) is a slightly more sophisticated form of neural network. By employing an additional *hidden layer* of neurons, this system can overcome the limitation of a single-layered network of only being able to accurately differentiate in a linear space, and is capable of the representation of a non-linear space (see section 4.4). The network consists of three fully-connected layers, the *input layer*, the *hidden layer* and the *output layer*, with each connection having a weight attached. Even further hidden layers may be used in some circumstances, such as when modelling a discontinuous function.

MLP uses neurons in the same way as the single-layered model, and so error can be calculated in the same way, using the difference between desired and actual output of the network, and the delta rule for updating the weights. However, the appearance of a hidden layer of neurons raises the further difficulty of how to assign ‘blame’ for the error in a network between the different nodes and weights. The traditional methodology used is *back-propagation* of error.

Instead of a sign activation function, a *sigmoid activation function* ($Y = \frac{1}{1+e^{-x}}$) is used. This ensures that the output is between 0 and 1, and that the derivative is easy to calculate. This is important in calculating the *error gradient*.

The weights are first initialised to random values, uniformly distributed.

Where the nodes at the input layer, hidden layer and output layer are respectively referred to as i, j, k respectively, the output at the hidden and output neurons can be calculated as follows:

$$Y_i(p) = sigmoid\left[\sum_{i=1}^{n} x_{ij}(p) w_{ij}(p) - \theta_j\right]$$

$$Y_j(p) = sigmoid\left[\sum_{j=1}^{m} x_{jk}(p) w_{jk}(p) - \theta_k\right]$$

where n is the number of inputs to the hidden layer and m is the number of inputs to the output layer.

The error gradient for the output layer (δ_k) is then calculated:

$$\delta_k(p) = y_k(p)\left[1 - y_k(p)\right] e_k(p)$$

where $e_k(p)$ is the error for the iteration (p) at the output layer:

$$e_k(p) = y_{desired,k}(p) - y_{actual,k}(p)$$

The weights connecting to the output neurons can then be updated for the next iteration:

$$w_{jk}(p+1) = w_{jk}(p) + \alpha y_j(p)\delta_k(p)$$

where α is the learning rate.

Similarly, the error gradient for the output later (δ_j) is then calculated:

$$\delta_j(p) = y_j(p)\left[1 - y_j(p)\right]\sum_{k=1}^{l}\delta_k(p)w_{jk}(p)$$

where l is the number of output neurons.

The weights connecting to the hidden neurons can then be updated for the next iteration:

$$w_{ij}(p+1) = w_{ij}(p) + \alpha y_i(p)\delta_j(p)$$

This process is then repeated for $p+1$ until the model has converged, or some other stopping criterion has been met (see section 4.7.1). Again, the threshold θ can be changed to move the decision boundary if that is desirable

4.5.2 Independent and Identically Distributed Model

An *Independent and Identically Distributed model* considers each element of that model (for our purposes, each transformation) to be independent of each other with respect to the effect it has in the optimisation space. We know that this is untrue in many circumstances for the case of program transformations, but it is still useful for providing a simple model of the system. It is a normalised distribution of probability values which represents the usefulness of each transformation individually, without reference to any possible interaction with others.

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.:

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

Here $P(s_j)$ is the probability that the transformation s_j occurs in good sequences.

4.5.3 Markov Model

The IID model (above) is incapable of representing any interaction between transformations. This is unlikely to be a good model for representing a transformation space, as we are aware that many transformations are *enabling transformations* – that is, by their actions they allow an another transformation to optimise further where before no optimisation would be possible: i.e. loop unrolling may expose an opportunity for common subexpression elimination between the head and the tail of a loop that did not exist before.

Similarly, we know that there exist *inhibiting transformations*, that while perhaps providing some optimisation themselves, may disable the effect of future transformations which might eventually produce better optimisation overall: i.e. the conversion of a for-loop to a while-loop which breaks a perfectly nested for structure and prevents loop tiling from occurring safely.

For these reasons, it useful to use a model which represents the interactions between transformations – a *Markov chain* provides such ability.

A Markov chain for transformation sequences can be defined as follows:

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

where $\mathbf{s}$ is a sequence of length L and s_i with $i = 1,\ldots,L$ is each position of the sequence with possible states taken from $\mathcal{T} = \{t_1,t_2,\ldots,t_N\}$. 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, 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.

4.5.4 Nearest Neighbours

Nearest neighbours is a very simple statistical technique, usually used for solving *classification problems*. Features (see section 4.3) from labelled data are extracted, and that data projected onto a n-dimensional space, where n is the length of the feature vector.

When a new unlabelled feature vector is presented, it is classified by attributing it to the class of its closest neighbour in the feature space, usually using Euclidian distance. The main advantages of this scheme are the simplicity of the technique and the lack of any time required for training the model, whereas potential disadvantages include significant computational complexity when a large amount of labelled data is present due to the need to calculate the distance between the new input and every existing labelling point in the space.

4.5.5 Principal Components Analysis

In general, any reduction in the dimensionality of a space will inevitably result in some loss of information. A good dimensionality reduction technique will preserve as much of the information as possible that can be used to differentiate between different classes.

Principal Components Analysis (PCA) is an unsupervised learning technique (see section 4.2.1) which helps in feature selection by removing features which do not vary across the feature space, whilst retaining those that do.

PCA is a linear transformation which produces a subspace of some bigger space that possesses the greatest variance over that space - that is to say it eliminates redundant information and tries to encapsulate as much information from the original space in a smaller number of principal components (which are a combination of the original features), now used as the new features. It does this with no reference to the output classification space, using only the input space, and can therefore be classed as a type of unsupervised learning. More formally, it is a rotation of the coordinate system of the original space of vectors of dimensionality m to a new set of coordinates on a space with dimensionality r, where $r < m$, such that the greatest variance by any projection of the data set comes to lie on the first axis, the second greatest on the second axis and so on. These axes are the principal components, ordered by variance, and so the top five principal components capture the most variance possible and can be used as features for our nearest neighbours classifier.

Step 1.

Construct a $m \times n$ matrix M, where m is the number of programs and n is the number of features. Each row is the original transposed feature vector for one program. Calculate the empirical mean along each column and subtract that mean for each element in the column to create a new matrix N which has a zero empirical mean.

Step 2.

Calculate a $n \times n$ covariance matrix P:

$$P^{n \times n} = (p_{i,j}, p_{i,j} = cov(Dim_i, Dim_j))$$

where Dim_x is the xth dimension of an Independent and Identically Distributed Model. cov is the standard covariance function:

$$cov(X,Y) = \frac{\sum_{i=1}^{n}(X_i - \overline{X})(Y_i - \overline{Y})}{(n-1)}$$

Step 3

Calculate the unit eigenvectors and eigenvalues for square matrix P. Create a new $n \times n$ feature matrix, Q, by reordering the eigenvalues by greatest first, then entering the corresponding eigenvectors into Q so that the matrix contains these eigenvectors in its columns. The vectors with the largest corresponding eigenvalues represent the vectors which exhibit the greatest variance.

Step 4

Select the first r ordered eigenvectors or components from Q, where r is the number of principal components wanted for use as features as the output of PCA, and enter them into an $m \times r$ matrix S.

Step 5

Calculate the final $m \times r$ feature vectors T:

$$T = S^T N^T$$

4.6 Using Machine Learning with a Compiler

This section describes how machine learning can be used in the compiler field. Firstly, a search strategy is briefly detailed, followed by a method for performance prediction, and finally, a means to predict the effects of code transformation.

4.6.1 Searching an Optimisation Space

Given the huge optimisation space produced by attempts to optimise a single program, it makes sense that simply choosing one point in the space is not sufficient to obtain the best results. It is therefore worthwhile to make multiple attempts to compile a program, randomly selecting optimisations and test them one by one to see which is best. This is known as iterative compilation [26].

However, it is clear that this optimisation space is not randomly distributed, rather it has structure. This raises the question of how we can exploit this structure to better select which optimisations to test; a search strategy is needed. There are numerous search strategies which could be employed such as using a genetic algorithm or a probabilistic search. Chapter 5 will consider a simple probabilistic approach.

Probabilistic Search

For the case of testing ten transformations, which could be applied in any order only once, the potential optimisation space is around 10^{10} – obviously not exhaustively searchable. The simplest way of searching is just random sampling, however we can improve on this by building a probabilistic model online.

We can construct a probability vector, of which each element corresponds to a single transformation. Each element contains a probability P, where $0 \leq P \leq 1$. Elements may be initialised to 0.5, or randomly initialised. A length of transformation sequence must be selected. This can either be determined randomly for each run, within specified bounds, or fixed for all runs.

Once the length of sequence has been determined for the run, the transformation vector must be populated. This is done by selecting the requisite number of transformations with respect to their associated probability. The run can now proceed with the selected transformations used, and the result recorded.

The probability vector must then be updated by using the result of the run. An example of how this might be done is to equally distribute the responsibility for the speedup obtained among the transformations used, so that their corresponding elements in the probability vector are multiplied by the speedup obtained, then normalised to 1. In this way, transformations which contribute to a net speedup of the program are gradually more and more likely to be selected for each test run, and those which cause a reduction in speed are less likely to be selected. In this way, the search can focus on the more profitable transformations, and combinations thereof, and select less useful transformations less often.

This process can be repeated over and over until the required performance is achieved or a set amount of time has passed. The probability vector is reset for each new program on the basis that different programs benefit in different way from transformation

4.6.2 Predicting Execution Time

Predicting the performance of a new piece of code is helpful when wanting to reduce the time spent running the new code, either on real hardware, or on a simulator. Using machine learning, we can make a prediction of the execution time of a new program many orders of magnitude more quickly than a cycle-accurate simulator. This section gives a basic overview of how such a system might be constructed.

Programs are needed to both train and evaluate the system. The programs are then partitioned into training and testing sets. The number of programs needed for training varies with both the type of the model employed and its complexity. If there is a shortage of training data, then cross-validation (see section 4.7.2) may be used.

Code features (see section 4.3) are determined and extracted for each program in both the training and testing sets. There is a single feature vector associated with each program. Each of the training programs are then executed and their execution times recorded. These training runs will provide the experience necessary to build the model.

Models may be constructed in a number of different ways depending on which model is chosen (see section 4.5), for example the MLP model (see section 4.5.1.3). The model is constructed by feeding each feature vector into the model, one at a time, and supplying the corresponding execution time so that the error may be corrected. The model should be trained on all inputs, over and over until the model stabilises, while also watching out for overfitting (see section 4.7.1). The model's values are then locked to the learned static values.

FEATURES→MODEL→EXECUTION TIME GRAPH

Having trained the model, it can then be supplied with a new, unseen feature vector from the testing set. The model will then output the predicted execution time, which can be compared with the actual execution time for the real hardware or simulator, in order to evaluate the results.

4.6.3 Predicting Effects of Transformation

Evaluating the effectiveness of a particular transformation can be time consuming, particularly if the program must be run on a simulator. If performance tuning is being attempted, it is beneficial to be able to try as many different transformations as possible, and the limiting factor is likely to be the time taken to run the programs. Machine learning can assist in this problem by facilitating the construction of a model which automatically predicts the speedup of a modified program. Using code features, and a model which models performance independently of transformations, it is possible to predict the impact not only of transformations used to construct the model, but also new transformations not seen before.

This is similar to the more general case of predicting execution time (in the previous section) but differs in that here the runtime of the baseline is already known, and just the difference a transformation would make needs to be predicted. This approach is taken by Cavazos et al.[11].

FEATURES→MODEL→CHANGE IN EXECUTION TIME

Program features must be selected, as described in section 4.3. Extra features which describe the relative differences between both the original and transformed program are also used. Additionally, a set of *canonical transformations* must be selected. These are a set of transformations which is felt best characterise the transformation space - that is giving the best coverage of the kinds of transformation available. The number of transformations used is constrained by the time and resources available, but has been shown to work with as little as 4 [18].

The program to be modelled is then transformed using each of these canonical transformations in turn, and executed and its performance recorded, along with the baseline.

A model can then be built using feature-speedup pairs as inputs – the features of the transformed program and the speedup obtained relative to baseline. A number of regression models may be used (see section 4.5).

When a new transformation is to be evaluated, the baseline code is transformed, and the code features extracted from that transformed program. These code features are then used as input to the model, with the output being the expected speedup over baseline of this transformed program.

4.7 The Pitfalls of Using Machine Learning

In this thesis, we show how machine learning can significantly outperform manually derived heuristics and methodologies. Indeed, large speedups are available using these techniques, but it is important to remember that machine learning is not a panacea. It is not simply a matter of removing a heuristic from a compiler, and socketing in a model in its place. There are significant difficulties to overcome, both in the selection and utilisation of features, and in the training process of building a model. If the inputs to the model are not of sufficient quality, no modelling technique, sophisticated or otherwise, has any hope of providing good results. Similarly, if a model is not constructed and evaluated properly, it may not represent the true optimisation space correctly, instead oversimplifying it, or attributing complexity where none in fact exists. This section discusses some of these programs and some potential solutions.

4.7.1 Overfitting

In order to provide the best results, a model must *generalise* the space which it represents, allowing new, unseen, data points to be assigned with accuracy. A significant danger to good generalisation is *overfitting*. Overfitting is the attribution by a model of a more complicated optimisation space than the underlying data warrants. This might mean mapping the noise in the data, or result from a lack of data points which suggest complexity, where in fact there is none when more points are revealed. An overfitted model will produce poor predictions as any new data given to the model is unlikely to follow the complicated specifics of the training data.

An extreme case of overfitting is that of total training data memorisation. Given a model capable of representing a sufficiently complex system (such as a MLP with many hidden nodes, see section 4.5.1.3), and given enough time to train, the model may simply represent a memorisation of the training data, and not the general case. This results in perfect or near perfect prediction of the training data, but not a true representation of the space being modelled.

Thus, preventing overfitting is imperative when training a model. Many techniques have been suggested to assist avoiding overfitting, such as the early-stop method, and the most simple of which are good validation, the early stop method and Bayesian priors.

The Early-stop Method

The early-stop method is a very simple, though not properly mathematically analysable, technique for preventing overfitting, and ensure generalisation in models which employ iterative learning schemes such as gradient descent. The model is trained on the training set data as normal, causing the model to adjust to fit the data, and the error rate to gradually lower. However, instead of ceasing this training after the model and the data have converged and the error rate is static, the learning is stopped early.

Choosing when to stop learning can be determined by constantly referring to a separate validation data set. After a fixed number of iterations, learning is temporarily halted, and the model evaluated on the validation set. Learning then proceeds for another fixed number of iterations, and the same validation is done, and this is repeated. When the falling error rate on the validation set has reached its lowest point, and the error rate begins to rise, then the learning is halted and the model fixed. It is important to note that the error rate on the training set may still be falling at this stage, but it is necessary to stop learning to enable generalisation.

If enough data points are available, it is advisable to use a third labelled data set called a testing set for further validation when using this method – this is to ensure that there is no overfitting toward the validation set. In reality, the availability of labelled data is often low, and so *cross-validation* is used.

4.7.2 Cross-validation

Cross-validation is a technique for ensuring the generality of a model when labelled data is scarce and must be used in the training of the model. It is advantageous as it allows almost all of the data to but used in training, whilst still ensuring proper validation of data, and thus good practice.

The data is partitioned into P segments, and the model trained using $P-1$ segments. The model is then evaluated using the remaining data segment. The model is then rebuilt P times, each time omitting a different segment and evaluating on it. In an extreme example, the data points can be partitioned individually, meaning as many models as the number of data points need to be constructed. This is known as *leave-one-out-cross-validation*. The main disadvantage of this scheme is the time and effort involved in building a large number of models. When a model is very complex, it may require a long time to complete training, even on modern machines, and this must be taken into account.

4.7.3 Underfitting

Underfitting occurs when a model is used which is not capable of representing the complexity of the underlying system for which it is being used to represent, such as a straight line to solve the XOR problem (see figure 4.1). Good testing and validation techniques can ensure that underfitting has not occurred, as well as using existing knowledge of the complexity of the space and choosing models accordingly.

4.8 Summary

This chapter has described the basics of machine learning, and how it can fit in a compiler context. Sections 4.1 and 4.2 dealt with what machine learning is and why it is useful. Representing programs by means of features was described in section 4.3, and the complexity of the problems has been discussed in section 4.4, which directly relates to which implementation of machine learning is likely to be most successful. Models capable of expressing different orders of complexity were presented in section 4.5. Examples have been given of how machine learning can solve specific compiler problems in section 4.6 , and a short guide to what can go wrong when using machine learning techniques given in section 4.7.

Chapter 5

Evolving Iterative Compilation

There exists an unwanted gap in performance between compiled code and hand-written code. Iterative compilation [7, 13, 26] narrowed this gap by attempting a large number of different optimisation strategies, and choosing the best. The implication of this work is that built-in compiler heuristics which select optimisation strategies are not doing as good a job as is possible.

This chapter proposes a new approach to selecting compiler transformations – namely probabilistic optimisation. It details how stochastic methods can be used to select the high-level transformations, directed by execution time feedback, where optimisation space coverage is traded off against searching in known good regions. Using such an approach we achieve significant performance improvements – on average over 1.71 across three different architectures. This approach can easily be transfered to other, or even yet to be invented, processors and extract high levels of performance unachievable by traditional techniques with no additional native compiler effort.

Section 5.1 outlines the main problem and the motivation behind this chapter; section 5.2 describes the probabilistic search approach; section 5.3 gives details of our experimental set up; section 5.4 presents our results and our analysis thereof and section 5.5 gives some brief conclusions.

5.1 Motivation

Embedded systems designers are presented with a dilemma: to hand-code their programs using assembly code, or to use a compiler. The former has much to offer – it produces very fast, clean and small code, but comes at the price of the significant time required to code the program. In addition there is the cost of hiring an experienced assembly programmer to carry out the work and the maintainability issues such assembly code generates.

On the other hand, if a compiler could be used then these problems are reduced, but it entails sacrificing execution speed and code size [67]. Indeed, with the increasing speed of embedded processors, chips are increasingly programmed using high-level languages as the benefits begin to outweigh the cost of assembly code.

Time-to-market is now a significant driving force in embedded systems, and with chips becoming more complex, compilers are finding it harder to keep pace, and even more difficult to obtain good performance on these chips [50]. At the same time, coding in assembly could delay this fast-moving field from design to market; thus we have increasing demand for compiled code, with the compiler having decreasing ability to exploit the processor.

It is clear that a solution is necessary to this problem, therefore, there has been significant research interest in improving the performance of optimising compilers for embedded systems, e.g. [41]. Such work largely 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. This chapter proposes a solution to help counter this problem.

Given that an embedded system typically runs just one program in its lifetime, we can afford much longer compilation times (e.g. in the order of several hours) than in general-purpose computing. In particular, feedback directed or iterative approaches where multiple compiler optimisations are tried and the best selected, has been an area of interest [51]. However, these techniques still give relatively small improvements as they effectively restrict themselves to trying different back-end optimisations.

In this chapter, an entirely distinct approach is considered, namely using source-level transformations for embedded systems. Such an approach is by definition highly portable from one processor to another and is entirely complementary to the efforts of the manufacturers back-end optimisations.

While high-level approaches can deliver good performance, it is extremely difficult to predict what the best transformation should be. It depends both on the underlying processor architecture and the native compiler. Small changes in the program – a new release of the native compiler or the next generation processor – will all impact on the transformation selection. Typically, high level restructures have a static simplified model with which to guide transformation selection. It has been shown [14, 26], however, that the optimisation space is highly non-linear and that such approaches are unlikely to prove good solutions.

5.1.1 Motivating Example

High-level transformations are a portable, yet highly effective way to improve performance by assisting the back-end compiler to produce efficient code. Deriving efficient program transformation sequences, however, is a complex task. For all but the most basic programs, the interaction between the source-to-source transformation, the back-end compiler and its built-in optimisations and the underlying target architecture cannot be easily analysed and exploited. Furthermore, programmers frequently apply their own program transformation to the program they wish to improve based on their expert knowledge and experience with a specific processor and its compiler. However, with each new generation of the processor, or even the release of a new compiler version, their knowledge becomes outdated. Furthermore, new processors and their frequently immature compilers are a challenge for each program developer aiming at high performance.

As an example, consider the program excerpt in figure 5.1(a). The `lmsfir` function is part of the UTDSP [40] (see section 3.1.1) `LMSFIR` benchmark. It computes a single point of an N-tap adaptive finite impulse response (FIR) filter applied to a set of input samples. The first of the two *for* loops iterates over the input and coefficient vectors and performs repeated multiply-accumulate (MAC) operations. The second loop updates the filter coefficient for the next run of this filter function.

In figure 5.1(b) the main differences due to transformations in an optimised Analog Devices TigerSHARC (see section 3.2.1) implementation are listed. While the routine

(a) Original implementation	*(b) TS-101 implementation*	*(c) TriMedia implementation*
`void lmsfir(float input[], float output[],`		
`float expected[], float coefficient[], float`		
`gain)`		
`{`		
`int i;`		
`float sum,term1,error,adapted,old_adapted;`		← New temps. introduced
`sum = 0.0;`		
`for (i = 0; i < NTAPS; ++i) {`	← Loop totally unrolled	← Lowered to DO-WHILE loop*
`sum += input[i] * coefficient[i];`	← Array references dismantled	← Pseudo 3-address code
`}`		← Linear pointer-based
`output[0] = sum;`		array traversal
`error = (expected[0] - sum) * gain;`		
`for (i = 0; i < NTAPS-1; ++i) {`	← Loop totally unrolled	← Loop totally unrolled
`coefficient[i] += input[i] * error;`	← Array references dismantled	← Pseudo 3-address code
`}`		← Linear pointer-based
`coefficient[NTAPS-1] = coefficient[NTAPS-2] +`		array traversal
`input[NTAPS-1] * error;`		
`}`		

* See figure 5.2 for the specific example of this loop.

Figure 5.1: Differences between the original *lmsfir* implementation (a), and implementations for the TigerSHARC (b) and TriMedia (c) processors

has not changed semantically, it outperforms the routine in figure 5.1(a) by a factor of 1.75 on the TigerSHARC TS-101 processor. In this transformed version of the program, both loops have been flattened and the array references dismantled into explicit base address plus offset computations.

On the Philips TriMedia (see section 3.2.2), however, different transformations produce the best performing *lmsfir* implementation (see figure 5.1(c)). Here the speedup of 1.2 is achieved by converting the first *for* loop into a *do-while* loop and flattening the second. All array references have been converted to pointers and an almost 3-address code produces the best result. The first loop of example 5.1(a) in its optimised form

```
data = input; coef = coefficient; sum = 0.0F;
i = 0;
do
   {
      {
         float *suif_tmp, *suif_tmp0;
         suif_tmp = data;
         data = data + 1;
         term1 = *suif_tmp;
         suif_tmp0 = coef;
         coef = coef + 1;
         term2 = *suif_tmp0;
         sum = sum + term1 * term2;
      }
      i = i + 1;
    } while (!(8 <= i));
```

Figure 5.2: First loop of example 5.1(a) optimised for the TriMedia processor

for the TriMedia is shown in figure 5.2.

This short example demonstrates how difficult it is to predict the best high-level transformation for a new platform. Iterative compilers interleave transformation and profiled execution stages to actively search for good transformation sequences. Portable, optimising compilers, however, must be able to search a potentially huge transformation space in order to find a successful sequence of transformations for a particular program and a particular architecture. This chapter proposes a probabilistic search algorithm that is able to examine a small fraction of the optimisation space and still find significant performance improvements.

5.2 Probabilistic Search

This section describes a scheme to intelligently search the vast optimisation space presented to a compiler. Using a combination of probabilistic and random search, the compiler can focus in on the profitable optimisation sequences, but still gain good enough coverage of the space to avoid becoming stuck in local minima, and limiting the scope of the search.

Selecting the best overall high-level transformation normally consists of selecting a sequence of smaller transformations which are applied to part or all of the program. Given that certain transformations may be parameterised (for example, loop unrolling is parameterised by the unroll factor), and that different combinations may be considered, selecting the best transformation is effectively an optimisation problem over the space of all possible transformations.

This approach to program optimisation makes use of an iterative transformation framework called the COLO Tool (see section 3.3.1) that alternates phases in which individual points of the optimisation space are sampled and their fitness is evaluated. In particular, transformation sequences are constructed and applied to the input program and the resulting program is then executed to determine its performance. This *dynamic* program optimisation approach does not rely on model-based static analysis, but guides the search on actual performance.

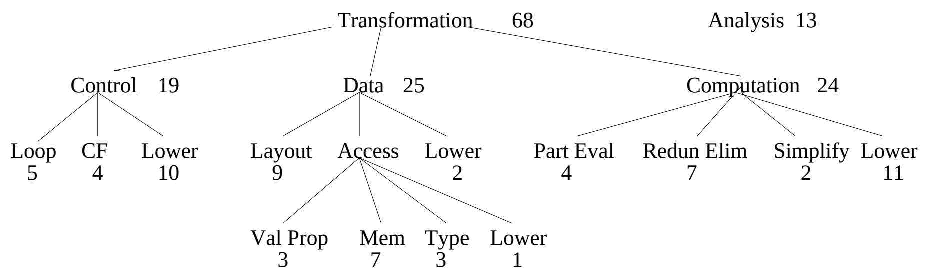


Figure 5.3: Transformation categorisation

5.2.1 Optimisation Space

In this chapter we consider 81 high level transformations (provided in appendix A), applicable to C programs and available within the SUIF [29] based compiler framework (see section 3.3.1.1). For convenience we have classified them as shown in figure 5.3. 13 are in effect analysis phases that mark the IR enabling later transformations which actually modify the source. These transformations can be classified into three broad groups; those aimed at modifying the program's control-flow, those that modify the actual computation performed and those focused on data which is further subdivided into actual layout and access. These broad categories are further refined as shown in figure 5.3.

All categories contain lowering transformations which translate a complex structure into a smaller one, i.e. unpacking a structure into its sub-components.

The control-flow transformations are aimed either at loop transformations or more general control-flow changes. The data access transformations include value propagation, modifying memory references and data type conversion. Finally, the computation based transformations include partial evaluations, redundancy elimination and code simplification. This is by no means a definitive transformation taxonomy, but provides an overview of the options available.

5.2.2 Optimisation Algorithm

Central to the success of this technique is the optimisation algorithm hosted within the optimisation engine of the COLO Tool framework. The huge size of the optimisation space and its complexity make it necessary to find a balanced trade-off between *space exploration* and *focused search*. For the benchmarks considered here, the size of the space is approximately 81^{10} (81 transformations in any order, up to length 10). To find good points, whilst keeping the number of sample points (and thus the number of program runs) within reasonable limits, a probabilistic algorithm is employed.

Although the random search of the optimisation space leads to significant performance improvement [26], it is, by definition, unable to direct efforts and search for an optimal point. If a transformation or sub-sequence is found to consistently perform well or poorly, or indeed have no effect, we would like to use this information to guide the search. However, there is a natural tension between avoiding hardwiring of biased heuristics and cost-effective search. What is needed is a technique that combines an unbiased sampling of the transformation space with feedback-focused attention on good areas.

In order to overcome this dilemma of space exploration vs. focused search, two simple, yet powerful algorithms are combined, representing each of the two domains. These two algorithms compete with each other and within a final *merge* stage the best of the two individual solutions is chosen. To facilitate a broad and non-biased space coverage we have chosen a simple *random search* as our space exploration algorithm. The focused search is represented by a *search* algorithm inspired by a modified *Population-based Incremental Learning (PBIL)* [3] approach. Both algorithms can be considered as two extreme cases of a continuum where the learning rate is $LR = 1$ for the PBIL inspired technique and $LR = 0$ for random search. In particular, in a competitive learning network the activations of the output units are computed and the weights adjusted according to the rules given by the following two equations [3]:

$$output_i = \sum_j w_{ij} \times input_j \tag{5.1}$$

$$\Delta w_{ij} = LR \times (input_j - w_{ij}) \tag{5.2}$$

A learning rate $LR = 0$ leads to constant weights which are not adjusted during the search. On the other hand, a learning rate $LR = 1$ enforces strong adjustment to the individual weights over changing input. These two algorithms are discussed in the following two sections.

5.2.2.1 Space Exploration

Random search assigns a constant uniform probability distribution to the set of transformations and chooses the next transformation solely based on a value generated by a pseudo-random number generator. In the case of parameterised transformations, we equally divide the assigned probability across all enumerated versions. For example if each transformation has a 0.1 probability of being selected but there are 50 loop unrolling options, then each of them is assigned a probability of 0.002.

The learning rate, LR, is 0 for random search as no information is carried across iterations of the algorithm and from equation 5.2 it follows that $\Delta w_{ij} = 0$.

Both the transformation and the length of the transformation sequence (up to some upper limit) are determined by a random process. The random search algorithm does not use the effectiveness of any transformations to direct its search.

5.2.2.2 Focused Search

PBIL is a stochastic search technique which aims to integrate genetic algorithms and competitive learning. It increases the probability of an option being selected whenever a positive instance using that option is encountered.

In our stochastic optimisation algorithm, transformations have an associated selection probability, but unlike the space exploring random search algorithm, probabilities can change over time and their distribution does not need to be uniform, i.e. $LR \neq 0$. In fact, we have chosen $LR = 1$ to emphasise its fast convergence on encountered performance enhancing transformations. The original PBIL algorithm considers binary encodings of parameters and generates a population of solutions based on a fixed-length probability vector, which had to be modified for this purpose.

Starting with a uniform probability distribution, sample points (i.e. transformation sequences) are chosen and evaluated by executing the corresponding program. The selection probabilities of the individual transformations are updated based on the success (i.e. execution time) of the sequence as a whole. Transformations contributing to better performance are rewarded while those resulting in performance losses are penalised. Thus, future sample points will include previously successful transformations more frequently, and search their neighbourhood more intensively.

Standard PBIL allows for random mutation within the probability vector, but we discard this as we do not wish to incur the overhead. Finally we do not generate a population based on a probability vector, but just one candidate. Depending on its success we update the probability vector accordingly.

The high learning rate, lack of mutation and a single candidate per generation means that the search is strongly focused on the result of feedback.

5.3 Experimental Setup

5.3.1 Processors and Compilers

The adaptive transformation scheme is evaluated against three different processors representing different aspects of the embedded computing domain. Among the three embedded processors are a high-performance floating-point digital signal processor, the Analog Devices TigerSHARC TS-101 (see section 3.2.1), a multimedia processor, the Philips TriMedia TM-1100 (see section 3.2.2), and an embedded processor derived from a popular general-purpose processor architecture, the Intel Celeron 400 (see section 3.2.3).

As back-end compilers we used Analog Devices' VisualDSP++ 3.5 for the TigerSHARC v7.0.1.5, Philips' TriMedia v1.1y Software Development Environment (SDE v5.3.4) for the TriMedia, and both Intel's ICC 8.0 and the GNU GCC 3.3.3 for the Celeron. The highest optimisation settings were used on the native compilers and execution times were measured using hardware cycle counters.

5.3.2 Benchmarks

The technique is evaluated on the *UTDSP* [40, 54] benchmark suite. Details are given in section 3.1.1. This set of benchmarks contains compute-intensive DSP kernels as well as applications composed of more complex algorithms and data structures. Many of the programs are available in up to four coding styles (explicit vs pointer-based array references, plain vs source-level software pipelined). Some of the benchmarks are excluded from this study, due to the incompatibility between the differing interpretations of acceptable C syntax/semantic between SUIF and the back-end compilers. The

TigerSHARC in particular is much stricter than SUIF in terms of the C accepted. Additionally, some benchmarks are focused on bit manipulation which causes problems due to conflicting endianness.

5.3.3 Program versioning

Transforming or rewriting a program at source level may have an impact on performance. To illustrate this, consider each of the UTDSP benchmarks (see section 3.1.1) which are supplied in up to four distinct versions. Firstly each is written using arrays or pointers. These in turn may also be rewritten as source level software pipelined versions. Although these versions are four independent sources, each version can be readily derived from the other by pointer conversion/recovery [42, 23] or source-level software-pipelining [58].

Figure 5.4 shows the average execution time of each version across the benchmarks on each processor. On the TigerSHARC, the clean array version gives the best average performance while the TriMedia prefers the pointer based version.

We consider two compilers, GCC and ICC, for the Celeron. Both compilers marginally prefer the array based code over pointer based versions. In most cases, with the notable exception of the TriMedia, the software pipelined versions of the program perform poorly.

From this set of data, we can conclude that source-level transformations will affect performance and that this will depend on the processor, program and possibly the underlying compiler.

Due to the variation in performance of the four different versions, all speedups in this chapter are with respect to the best performing original code. In the case of the TigerSHARC this is normally the array based original code while on the TriMedia it is usually the pointer version.

5.3.4 Encoding Transformations

One of the main difficulties in selecting the best transformation sequence is that many transformations are position dependent, i.e. only applied to a part of the program. Unlike global optimisations, we have to specify the location of the transformation.

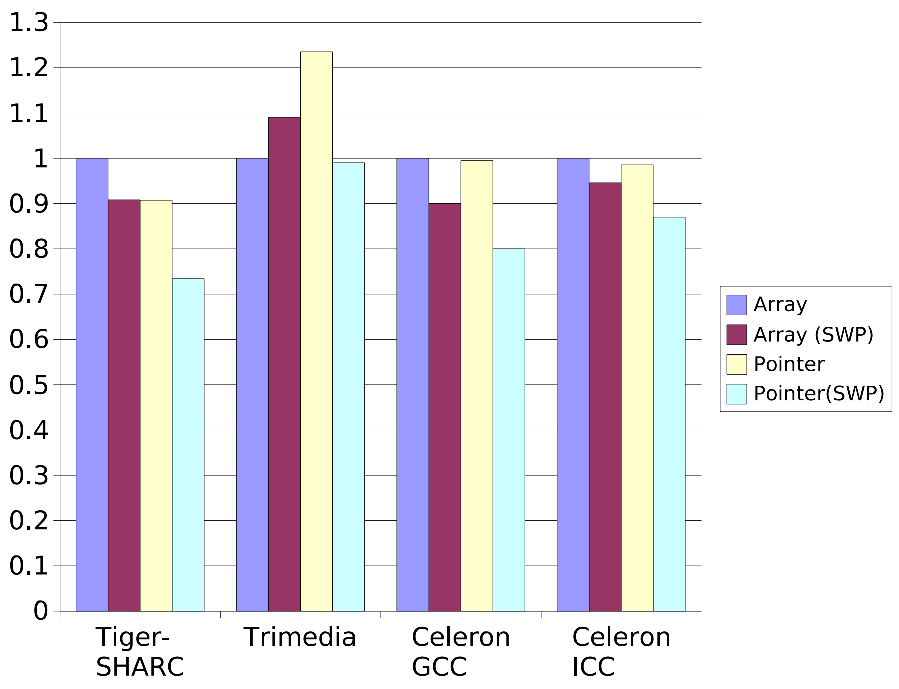


Figure 5.4: Relative speedup or slowdown for different coding styles per processor. The data is normalised to the performance of the baseline array code

Furthermore, these transformations may be parameterised. This leads to two problems : firstly, the optimisation space now increases in size and, secondly, it becomes asymmetric in description. This means search cannot proceed in a uniform manner.

To overcome this, the system employs a simple method to make the treatment of parameterised location specific transformations indistinguishable from the yes/no binary decision of global optimisations such as constant propagation. This is achieved by simply enumerating all possible parameters and all locations.

5.3.5 The COLO Transformation Framework

The chapter uses the iterative transformation framework called the COLO Tool (see section 3.3.1 for more details) to carry out all experiments. This section briefly describes it, and the different optimisation algorithms used to balance potentially conflicting search strategies.

Benchmark C code enters the COLO Tool and is translated into an intermediate representation on which all transformations operate. After finishing the transformation process, the IR is translated back into C code and compiled into an executable by the particular machine's back-end C compiler making use of its most aggressive optimisation setting.

The COLO Tool makes extensive use of the Stanford SUIF compiler [29] (see section 3.3.1.1) to provide a C front-end, a code generator and a rich selection of already implemented transformations.

5.4 Results and Evaluation

This section presents, discusses and analyses the empirical results that were gained using our iterative transformation tool on a number of processors. All results are found after running the search algorithm for 500 evaluations.

5.4.1 Results

As stated in section 5.3.3, all speedups are with respect to the best performing original program, giving a true evaluation of our approach. Thus, the best original execution time of the four possible versions of each program is selected for speedup comparison with the highest optimisation level selected on the native compiler.

5.4.1.1 Platform Based Evaluation

Figures 5.5, 5.6, 5.7 and 5.8 show the performance improvements achieved by our approach across processors and benchmarks. All the platforms benefited from iterative search. The TigerSHARC had an average speedup of 1.73, the TriMedia 1.43, the Celeron with GCC 1.54 and with ICC 2.14 with an overall average of 1.71. This overall figure demonstrates the importance of high-level optimisation. Using a platform independent approach we are able to reduce execution time on average by 41%, outperforming any other approach.

Examining the TigerSHARC results (see figure 5.5) more closely we see there is much variation. Surprisingly, the matrix multiplication routines can be improved by almost a factor of 7 by completely flattening the code. As this is such a well known routine, one would have thought that the baseline compiler would do well here, but it appears that the heuristic controlling the loop unroller in the backend compiler is unwilling to

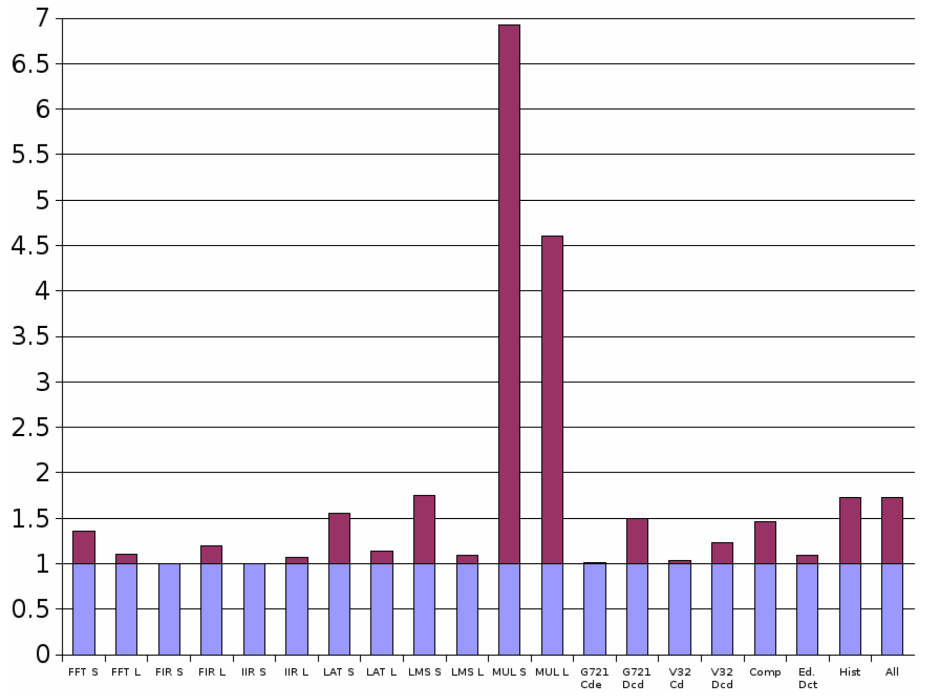


Figure 5.5: Speedup due to high-level transformation over the most aggressive back-end compiler optimisation alone for TigerSHARC

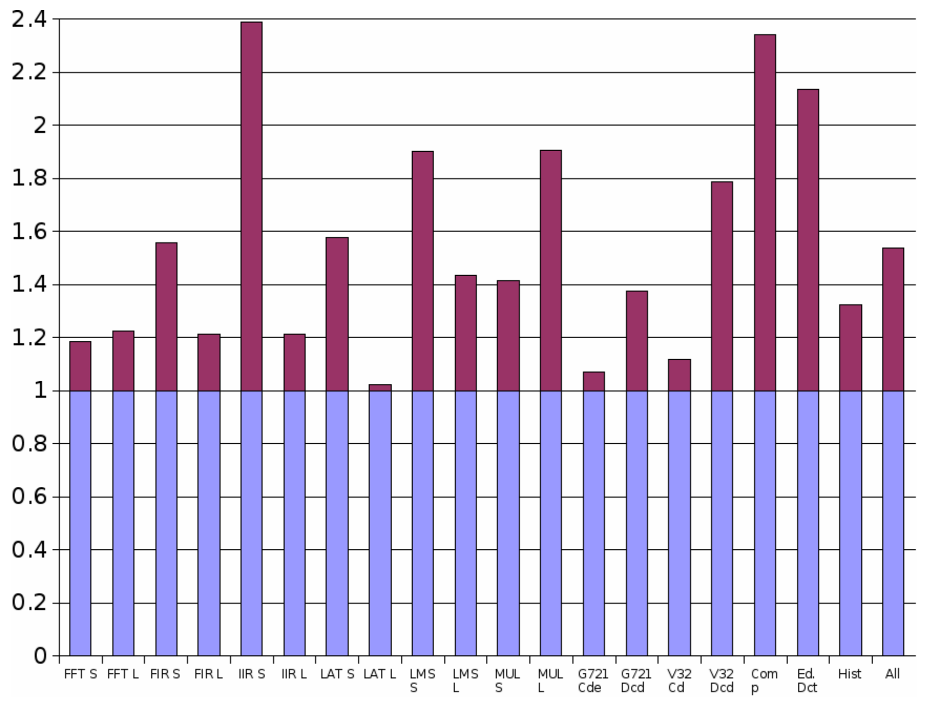


Figure 5.6: Speedup due to high-level transformation over most aggressive back-end compiler optimisation alone for Celeron/GCC

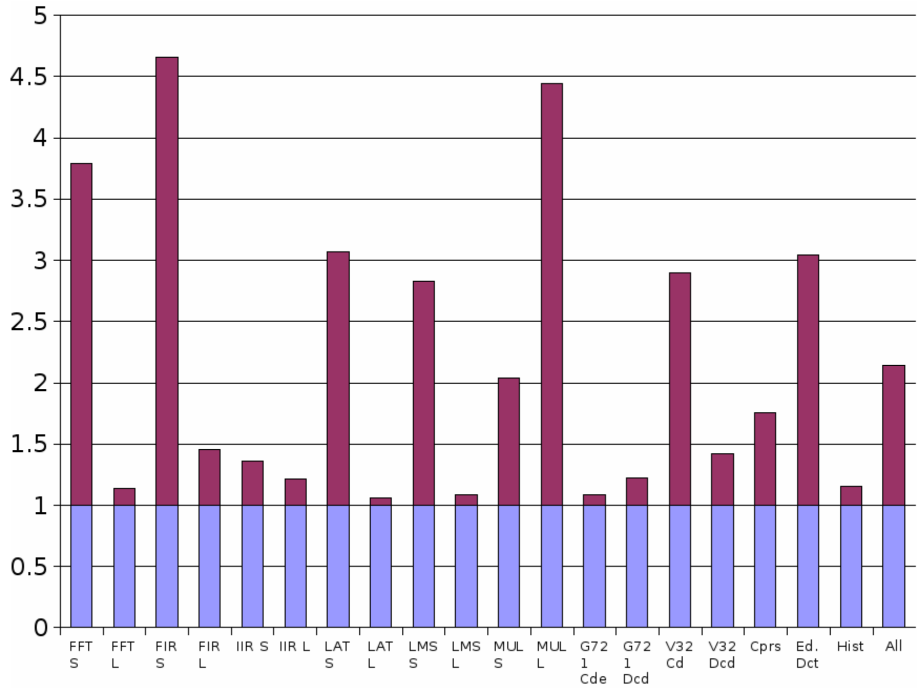


Figure 5.7: Speedup due to high-level transformation over the most aggressive back-end compiler optimisation alone for Celeron/ICC

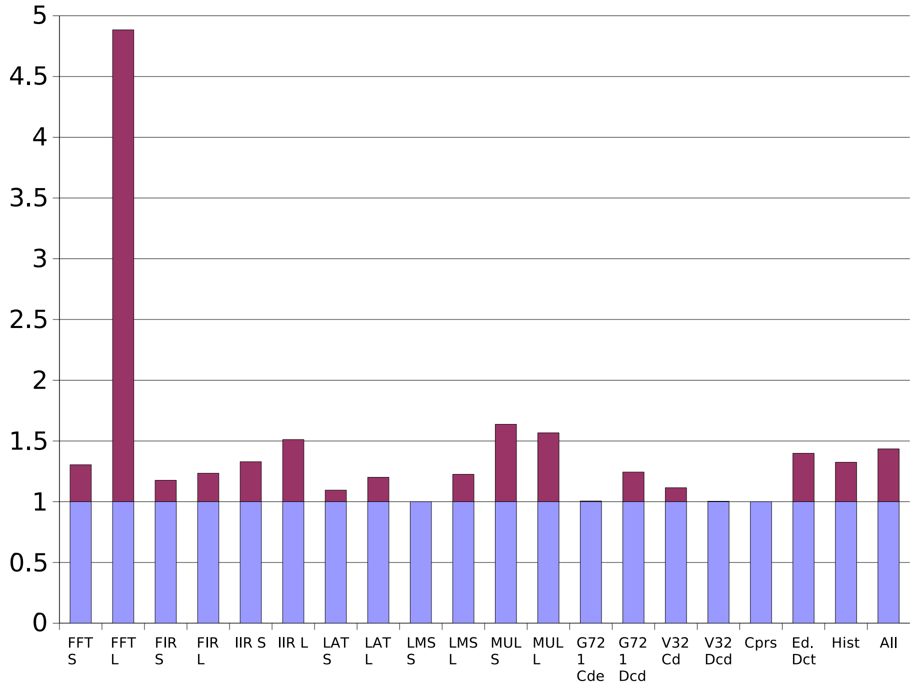


Figure 5.8: Speedup due to high-level transformation over the most aggressive back-end compiler optimisation alone for TriMedia

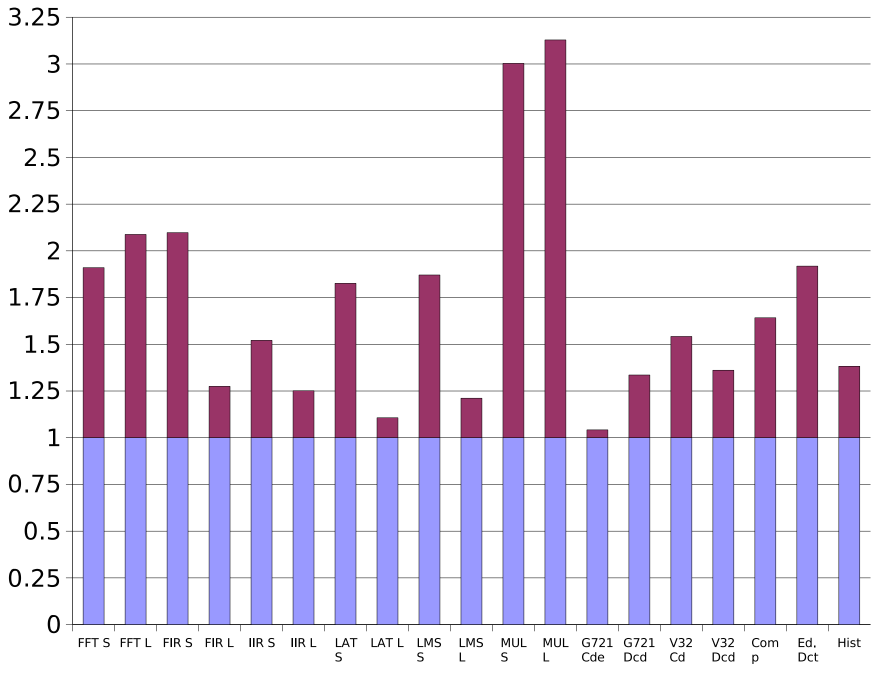


Figure 5.9: Program speedup averaged across all platforms

be aggressive enough here to derive the necessary performance. The compiler for the TigerSHARC is well respected in industry, and further supports the view that feedback directed compilation outperforms static heuristics, especially in extreme scenarios.

The iterative scheme performs less well on the very small data sizes of FIR and IIR, unlike the other processors. It also is unable to improve the performance of the G721 encoder – a problem shared by all of the processors. This is likely due to the large number of conditional branches present in these codecs, which makes them difficult to optimise using high-level transformation.

A different picture emerges when considering the Celeron processor with GCC (see figure 5.6) where the speedups are less variable. In direct contrast to the TigerSHARC, large performance gains are achieved on the small data sized IIR program. Good results are also found for the compression and edge detection applications. Like the TigerSHARC, little performance was gained on the G721 encoder.

The largest performance gains were achieved with the ICC compiler on the Celeron. This in itself is a surprising result given that it is the most mature compiler here and therefore should have proved difficult to improve upon. Like the TigerSHARC it performs well on the large matrix multiplication and the small FFT and poorly on the

G721 encoder. However, it performs well on the small IIR, as with GCC, and shares similar performance gains on edge detection and V.32 encoder. We will compare the two compilers GCC and ICC for the Celeron in more detail below (see section 5.4.3).

The TriMedia has the lowest average speedup of 1.43 and like the TigerSHARC has an uneven distribution of results with the large FFT achieving a speedup of almost 5. Once again it performs poorly on the G721 encoder, but unlike other platforms it performs poorly on the V.32 decoder and compress benchmarks.

5.4.2 Benchmark Orientated Evaluation

Across all the benchmarks, only three of the benchmarks fail to achieve the average performance improvement of 1.25. *LATNRM* benefits from loop unrolling, however, due to cross-iteration dependencies the native compilers instruction scheduler cannot take full advantage of the enlarged loop body. *LMSFIR* suffers from a coding style that introduces frequent conditional branches to the innermost loop. Similarly, *G721* is limited in its transformation potential by many conditional branches between tiny basic blocks.

Surprisingly, in four out of six cases high-level iterative search is able to speed up programs to a greater extent for small rather than large data sizes. This is counter-intuitive as many of the restructuring transformations only have any noticeable effect when dealing large amounts of data and computation. Examining the output code, it seems that in several cases the iterative search has completely unrolled or flattened certain sections of code, turning loops into large basic blocks and act as an enabler of baseline compiler optimisation. The large speedup of matrix multiplication on the TigerSHARC is also due to this reason when applied to the inner loop.

5.4.3 GCC vs ICC

Using two compilers on one platform gives an insight into their effect on performance. As expected, overall the ICC compiler outperforms the GCC and is approximately 1.22 times faster on average. However, after applying high level transformations on top of GCC, we see an improvement on average of 1.54, outperforming ICC on its own. This means that an automatic platform-independent approach could use a less mature compiler as a baseline, and still outperform hand-crafted optimisers based on many

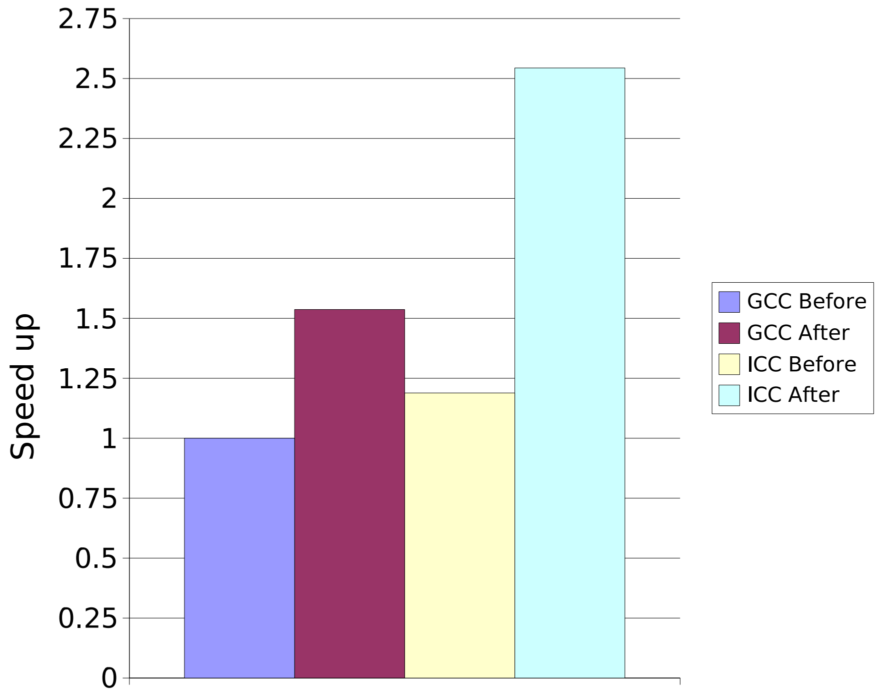


Figure 5.10: Comparison of the two compilers for the Celeron. Results are normalised to GCC performance before transformation.

years of work. Furthermore, it allows vendors to put less effort into their compiler, reducing the time to market of their product, while giving higher performance.

The diagram also shows that applying transformations to ICC gives a speedup of more than 2.5 relative to GCC alone. This also shows that a platform-independent approach can also port and scale with improved baseline improvements and is a complementary approach to vendor improvements. This additional speedup is likely because of superior low-level transformation within ICC. High-level transformation of code often exposes significant opportunity for optimisation at a lower level, and it seems this is better exploited by ICC.

5.4.4 Evaluating transformations

Overall, loop transformations have been identified as the most beneficial class of transformations in our framework. This category (cf. figure 5.3) is followed by the classes of value propagation transformations and partial evaluation. The differences between the remaining classes are too small to derive any significance from them.

Across all platforms and benchmarks, the focused search phase of the optimisation algorithm finds the best sequence 65% of the time with an average effective transformation sequence length of 4.1. For example, in *compress* on the TriMedia the best sequence of transformations was hoist loop invariants, optimise function parameter passing, globalise constants, scalarisation and flatten the main loop.

The remaining 35% of the time, the best transformation sequence was found by the random phase where the absolute length was on average 40.1. This result looks surprising at first. No high-level restructuring compiler research has suggested that such sequence lengths are beneficial and they obviously contrast with the focused search results. However, as we are randomly selecting sequences between 1 and 80, then an average around 40 is to be expected.

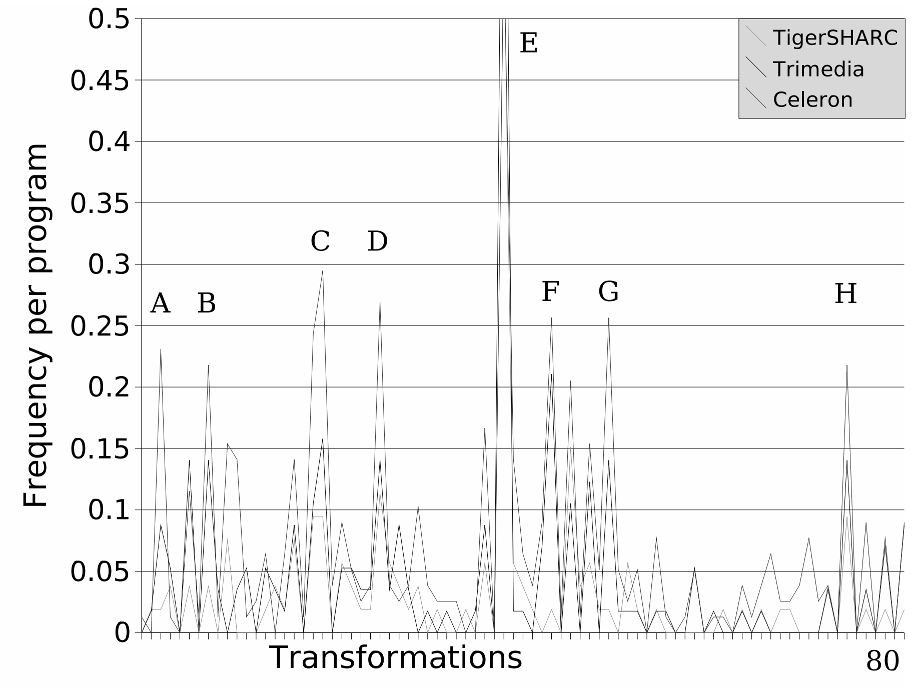
Furthermore, on examination it can be seen that there are many transformations included which do have any impact on the code. These junk transformation sub-sequences frequently contain repeated transformations or ones which have no effect on that particular program. Hence, the effective transformations sequence length is much shorter.

As PBIL only selects transformations that are guaranteed to have improved the program in the past, then redundant sub-sequences are eliminated, and this gives much shorter sequence lengths. This means that while long sequences may be beneficial, it is sufficient for future work to consider short but effective sequences, less than 10 in length.

It is interesting to note that while the focused search finds the best optimisation 65% of the time, it achieves an average performance gain of 1.57. Space exploration finds the best solution less often, but achieves an average speedup of 2.00 in these cases, justifying the choice of using two approaches to searching the space.

5.4.5 Distribution

Examining the probability distribution of the useful application of a transformation, there are eight transformations or peaks labelled A-H in figure 5.11. There is much commonality at first glance across the processors. Loop unrolling is by far the most successful transformation. Although it is well known to improve performance, it is surprising that it is so successful here as each of the native compilers applies unrolling internally. This means that the heuristic employed by the native compiler is not capable of extracting high performance from these benchmarks. Propagating known values and

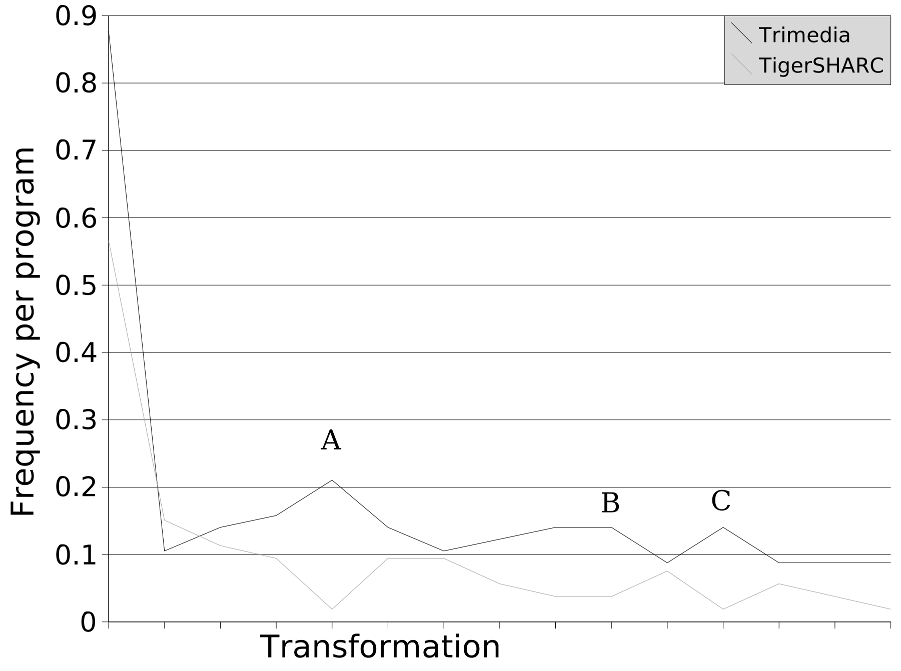


A - break up large expression trees, B - value propagation, C - hoisting of loop invariants, D - loop normalisation, E - Loop unrolling, F - mark constant variables, G - dismantle array instructions, H - function parameter passing optimisation.

Figure 5.11: Probability of transformation being successful

loop hoisting are also useful transformations, again surprising as a back-end compiler should perform this. Less obviously, breaking up expression trees (A) so that they can be effectively handled by the code generator proved useful. Finally changing arrays into pointer traversal (G) is useful for machines with separate address generation units while eliminating copies (H) reduces memory bandwidth.

If we focus now just on the TriMedia and TigerSHARC whose speedup profiles are similar, then we see that there are also differences among the processors. Figure 5.12 shows the transformations ordered by overall effectiveness. At three points A, B and C we see marked differences in the usefulness of transformations. This shows how transformations can have different effects on different architectures.



A - data layout analysis, B - control flow simplification, C - dismantle array references.

Figure 5.12: Highlighted differences in overall effectiveness of transformations

5.5 Conclusion

This chapter has described a probabilistic search algorithm for finding good source-level transformation sequences for typical embedded programs written in C. Source-to-source transformations have been shown to be not only highly portable, but also provide substantial scope for performance improvements. Two competing search strategies provide a good balance between optimisation space exploration and focused search in the neighbourhood of already identified good candidates. The work integrates both parameter-less global and parameterised local transformations in a unified optimisation framework that can efficiently operate on a huge optimisation space spanned by more than 80 transformations.

The empirical evaluation of this optimisation toolkit, based on three real embedded architectures and kernels and applications from the UTDSP benchmark suite, has successfully demonstrated that the approach is able to outperform any other existing approach and gives an average speedup of 1.71 across platforms.

Nevertheless, there is a significant drawback to this technique – the substantial amount of compile and evaluation time required to achieve the results. This has to be balanced against the often very long runtimes of embedded programs, which can afford such long compilation times, yet this is clearly an undesirable aspect of this technique.

The main reason for this very long compile time is that the optimisation of each program is carried out individually, starting afresh each time. However, we know from experience, as well as intuition, that similar code is often susceptible to similar optimisation. If there was a way to automatically gauge the similarity between programs, we should be able to prime our technique with previously acquired information – to learn from experience – which could dramatically speed up search, and improve the results. This *learning compiler* approach is investigated in the next chapter.

Chapter 6

Knowledge Acquisition and Transference

Iterative compilation has raised the bar for what can be considered well-optimised, machine-generated code [26, 47], by illustrating the significant performance gains still available to compilers by purely automated techniques. In addition, it has demonstrated that accessing these gains is an extremely difficult task due to the complicated and highly non-linear optimisation space.

Searching the optimisation space using iterative compilation can be an extremely time-consuming task [7, 13]. As has been shown in chapter 5, probabilistic methods can be used to help speed up this technique, and build up some knowledge of which optimisations are profitable to apply on a single program, but this knowledge is simply discarded at the end of compilation, and the process must start over from scratch on a new program.

In section 6.2, the scale of the problem facing compilers, and the wastefulness of previous techniques is discussed; in section 6.3 the experimental set up is outlined; section 6.4 describes how the optimisation space is characterised, and the interesting elements of the space; section 6.5 describes how models can help solve the problem, and how to train the models; in section 6.6 the features are selected and described; section 6.7 illustrates how the nearest neighbours technique can be used to achieve knowledge transference; in section 6.8 the results of the experiments are presented, and section 6.9 draws some conclusions from the data.

6.1 Introduction

In this chapter, we describe a statistical technique to replace hand-written compiler heuristics, which is capable of considering a highly non-linear optimisation space, with many dimensions. We show how knowledge of a program can be gathered, modelled and then applied to a completely new program, reducing the number of compilation iterations needed to achieve an equivalent performance increase by an order of magnitude, compared to previous techniques.

This is achieved by building statistical models of each of our training programs (see section 6.5), and employing *code features* and a simple statistical technique called *nearest neighbours* (see section 6.7) to determine which of our models a novel program is most similar to, and thus which model to apply. This work is primarily aimed at embedded platforms, and thus two embedded processors are used for evaluation: the Texas Instruments C6713 and the AMD Alchemy Au1500 MIPS32 based processor (see sections 3.2.4 and 3.2.5).

In order to do this, we must be able to both represent the characteristics of a program in a fashion amenable to machine learning techniques (so that we might know when our learned knowledge is applicable), and employ a methodology for representing and updating our understanding of the optimisation space, based on experience. The former is tackled by means of using code features in section 6.6, and the latter by building a mathematical model of the optimisation space using statistical techniques, as described in section 6.5. Critically, learned experience must additionally be allowed to be *transferred* from programs used to train the system onto new programs never seen before. This is discussed in section 6.7. Using these techniques achieves a substantial reduction in the number of iterations required to produce good performance.

In this chapter, source-level transformations [22, 58] for embedded systems are considered, as in the previous chapter (see section 5.2.1). Such an approach is, by definition, highly portable from one processor to another and provides additional benefit to the manufacturer's highly tuned compiler.

6.2 Motivation

Many optimisations in modern compilers have been traditionally based around using analysis to examine certain aspects of the code; the compiler heuristics then make a decision based on this information as to what to optimise, where to optimise and to what extent to optimise. The exact contents of these heuristics have been carefully tuned by experts, using their experience, as well as analytical tools, to produce solid performance.

It is easy to deduce from this that characteristics of code are important in deciding what and how to optimise. However, given the highly non-linear nature of optimisation interactions [14, 26] and the limited scope of these heuristics – normally limited to a simple linear calculation based only on local evidence – it is likely that a much better method of guiding optimisation can be produced if a larger scope, both in terms of code characteristics analysed and the assumed complexity of the output space, is utilised.

This chapter focuses primarily on embedded applications where performance is critical and, consequently, there has been a large body of work aimed at improving the performance of optimising compilers, e.g. [41]. 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 optimisations consequently give relatively small improvements [13].

Solving this problem presents several major challenges: the difficulty of producing a complex non-linear algorithm by hand and the difficulty of understanding which of the many hundreds of program characteristics are important in deciding this. In this chapter, we propose the answer to these challenges is to use machine learning techniques to automatically derive a better optimisation methodology, built around experience of what has gone before, and based on empirical evidence rather than an expert's opinion.

In chapter 5, we saw how feedback-directed search can offer a solution to this problem, but this technique alone takes a long time to reach a satisfactory result. A speedier technique would allow better performance to be gained in a fixed time, or the same performance to be obtained in a shorter time.

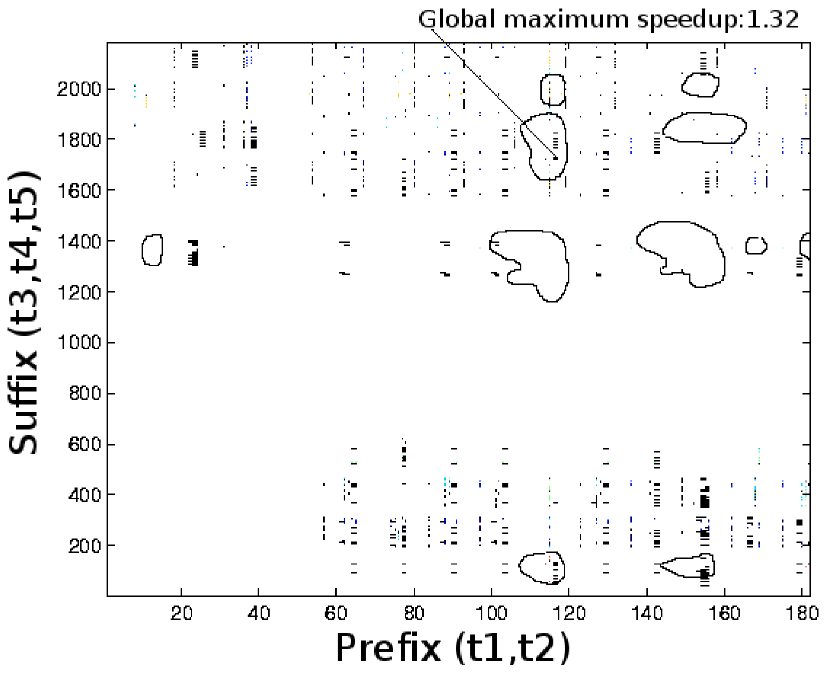


Figure 6.1: Points corresponding to those transformation sequences whose performance is within 5 % of the optimum for *adpcm* on the TI C6713. The contour is the predicted area for good optimisations.

6.2.1 Search space

The reason for long search search times in iterative compilation [7, 26], and to a lesser extent in chapter 5, is that determining the best high level sequence of transformations for a particular program is non-trivial. Consider the diagram in figure 6.1 showing the behaviour 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 suffixes 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 are likely to be could *focus our search*, allowing the minimal point to be found more quickly. Alternatively, given a fixed number of evaluations, we can expect improved performance if we know good areas to search within.

6.2.2 Focused search

This chapter demonstrates a technique that learns off-line, ahead of time, a predictive model to guide optimisation of a new program, based on learning from iterative evaluation of other programs – this predictive model suggests potentially good regions of the space to search. In figure 6.1 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, thereby rapidly reducing the cost of iterative search. This can be seen in figure 6.2, 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 speeds 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 6.2, such a large improvement would require over 80 evaluations using random search, justifying further investigation of predictive models.

6.3 Experimental setup

This section describes the experimental setup used in this work, including the processors and benchmark suite used for evaluation, and the transformations considered for an exhaustive study.

The experiments were driven by the COLO Transformation Framework Tool (see section 3.3.1) which allows complete control of source-to-source transformation selection and ordering.

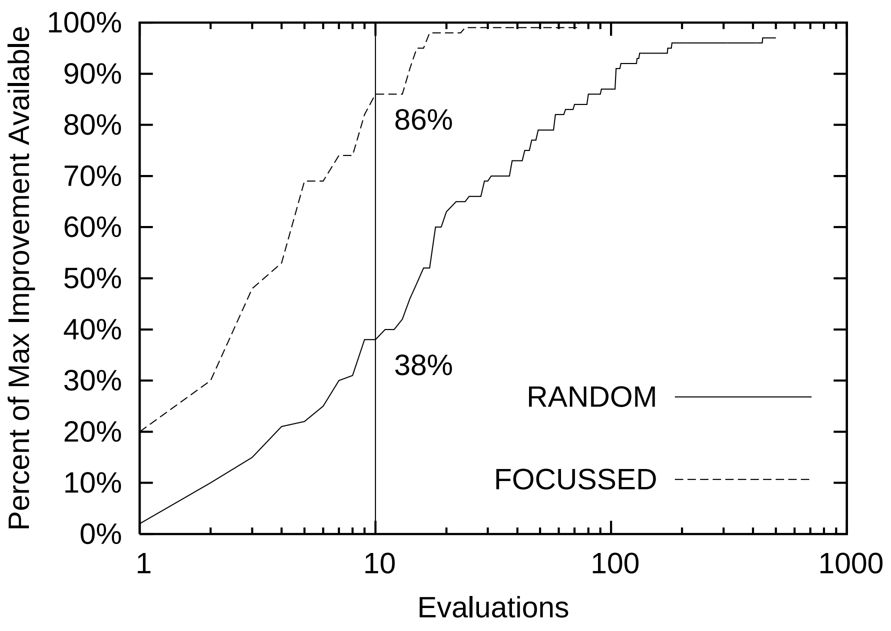


Figure 6.2: How close to the best performance random and the new focused search achieve on the *adpcm* benchmark on the TI platform. The random algorithm achieves 38 % of the maximum improvement in 10 evaluations; the focused search 86%.

6.3.1 Platforms

The experiments were performed on two distinct platforms to demonstrate that our technique is not specific to a particular processor – the Texas Instruments C6713 and the AMD Alchemy Au1500 (see sections 3.2.1 and 3.2.4).

The TI C6713 is a high end floating point DSP, a wide clustered VLIW processor with 256kB of internal memory. The programs were compiled using the TI's Code Composer Studio Tools Version 2.21 compiler with the highest -O3 optimisation level and -ml3 flag (generates large memory model code).

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.

6.3.2 Benchmarks

The *UTDSP* [40, 54] 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)” [40]. 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 as most important by DSP programmers and are used indefinitely in stream-processing applications. This is the same benchmark suite as was used in the previous chapter, and is described in section 3.1.1.

6.3.3 Compiler transformations

In this chapter, as in the previous chapter, source-to-source transformations are considered (many of these transformations also appear within the optimisation phases of a native compiler[1]). These are applicable to C programs and available within the restructuring compiler SUIF (see section 3.3.1.1) [29]. Further details of the framework are given in section 3.3.1.

For the purpose of this work, we have selected eleven transformations described and labelled in table 6.1. As four loop unroll factors are considered (arbitrarily), this increases the number of transformations considered to 14. All transformation sequences of length 5 are then exhaustively evaluated, selected from these 14 options. This allows the evaluation of the relative performance of our proposed techniques. In the later evaluation section (see section 6.8), we also consider searching, non-exhaustively, in a much larger space.

6.4 Characterising the space

Employing an exhaustive enumeration of all transformation options is the best, though time-consuming, method to evaluate the optimisation space. This allows us to make definitive statements about the space in terms of best available transformation, and to evaluate optimisation selection techniques with reference to a fully known space.

Label	**Transformation**
1,2,3,4	Loop unrolling
f	Loop flattening
n	FOR loop normalisation
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 6.1: The labelled transformations used for the exhaustive enumeration of the space. 1,2,3,4 corresponds to the loop unroll factor.

In order to characterise the optimisation space, all 14^5 transformation sequences are exhaustively enumerated on both platforms. Table 6.2 summarises the performance available; columns 2 and 3 refer to the TI while columns 4 and 5 refer to the AMD respectively.

The columns labelled `Improv.` (cols. 2 and 4) show 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 15.2% reduction was achieved for the TI and 19.6% for the AMD. This translates into an average speedup of 1.15 and 1.16 over the platform specific optimising compiler.

6.4.1 Best performing sequences

The columns labelled `Seq.`, (columns 3 and 5) in table 6.2 contain the best performing sequence for each benchmark on each machine. The individual letters within each entry refer to the labelled transformations in table 6.1, e.g. *i* = `if hoisting`. These entries show that the complexity and type of good transformation sequences is pro-

	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%	-	19.6%	-

Table 6.2: Summary of optimisation space on the TI and AMD using exhaustive search.

gram 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 `edge_detect`. This variance shows that different transformation sequences are needed for each different program. Two kinds of model are evaluated to represent our programs in the following section.

6.5 Representing Experience - Using Models

6.5.1 The case for models

As we have seen in chapter 5, it is clearly the case that certain optimisations, and indeed sequences of optimisations, are particularly suited to a particular program – however, it takes a long time to find these optimisations, or areas of the optimisation space. In order to speed up our search algorithms, we wish to focus our attention on the most profitable areas of the optimisation space. To this end, a model is built for each of our training programs, reflecting those transformation sequences for which the program obtained good performance, in the hope that this knowledge can be effectively transferred to new programs.

It is possible simply to record the best sequence achieved on other programs and hope that it improves the current program, however, this technique has drawbacks. Firstly, as the results in table 6.2 show, the best transformation on one program is never the best on others. Since our goal is to achieve the best speedup possible, we can afford to invest time to try several different optimisation sequences, and afford to be wrong some of the time. Knowing the best sequence on another program only provides one single option and cannot guide subsequent search within a larger space.

The alternative is to build intricate models that characterise the performance of all transformation sequences. Here the problem is that the model can be easily overfitted to the data, so that it cannot be generalised to other programs. Furthermore, such a complex model would require extensive training data, which may be costly to gather and is unrealistic in practice. In this section we consider two different models which try to summarise the optimisation space without excessive overfitting.

6.5.2 Building the model

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 from our training data.

Independent identically distributed (IID) model

The IID model is a very simple approach to modelling. It assumes that all transformations are independent (i.e., there are no interactions between transformations). Even though we know this is not the case, it still makes sense to start with this model as it is one of the simplest, and is easier to learn with a small number of datapoints than more complex models. A simple approach may provide a good ’priming’ so that search can uncover further speedup.

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.:

$$P(s_1, s_2, \ldots, s_L) = \prod_{i=1}^{L} P(s_i). \qquad (6.1)$$

Here $P(t_j)$ is the probability that the transformation t_j occurs in good sequences. For our data set 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. This allows us to capture information about sequences which are not quite the best, but still do very well, expecting that they might be the best for similar programs. We calculate $P(t_j)$ by simply counting the number of times t_j occurs in good sequences and normalise 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 IID distribution, and 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 practice 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.

Markov Model

Using the IID model, there is no way to represent interactions between transformations, and thus any such information present is discarded. This is particularly restrictive in cases where there are transformations that enable the applicability of other transformations or when they only yield good performance after others are applied. It is therefore useful to try to model these interactions between transformations, and to do this, we use a Markov chain based model.

A Markov chain for transformation sequences can be defined as follows:

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

Under this scenario, the probability of a transformation occurring is dependent on the transformations proceeding it. This model assumes that the probability does not change along the sequence – i.e., it is 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 learned 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 learned 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.

6.5.3 Speeding up search: Evaluating the potential of the models

Baseline search

Two common methods used to search the transformation spaces are compared against: a blind random search (RAND) and a slightly more sophisticated genetic algorithm (GA). Random search generates a random string of transformations where each transformation is equally likely to be chosen.

The genetic algorithm was configured in the same manner as ”best” GA in [16] with an initial randomly selected population of 50. This follows the standard GA format, and uses a two-point randomised crossover, and scaled fitness values as weights in making reproductive choice. In addition, the algorithm employs a kind of ‘hash checking’

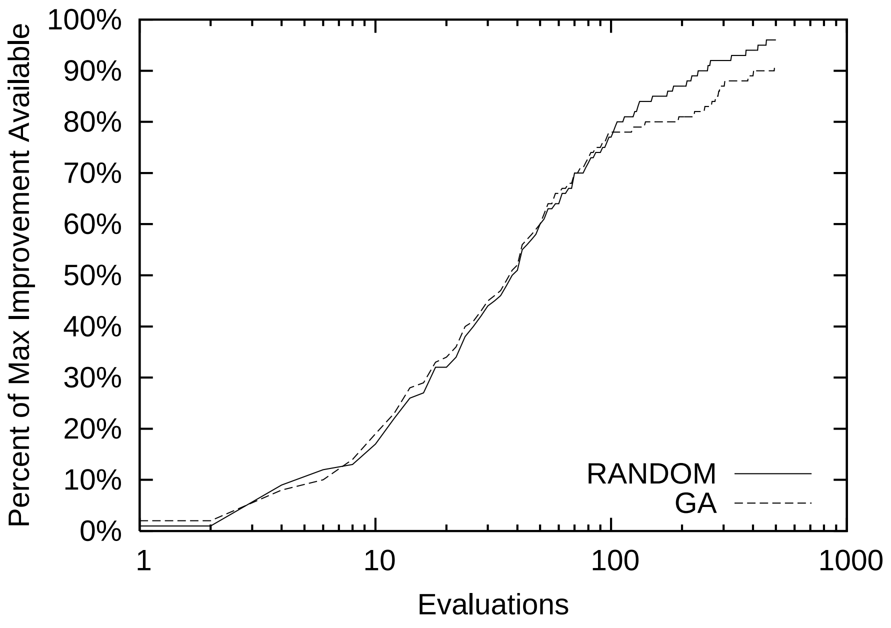


Figure 6.3: Performance with respect to evaluations for the random (RAND) and genetic (GA) search algorithms on the TI board. The x-axis denotes (logarithmic scale) the number of evaluations performed by each search. The y-axis denotes the best performance achieved so far by the search; 0 % represents the original code performance, 100% the maximum performance achievable. Results averaged over all benchmarks

system, where all new sequences are hashed and that hash stored. When new sequences are generated, they are checked against previous hashes so that there is no duplication of previously evaluated sequences. If a duplication is detected, the sequence mutates until it becomes unique.

For the exhaustively enumerated space, both algorithms have similar performance as can be seen in figures 6.3 and 6.4. Here, the best performance achieved so far by each algorithm is plotted 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.

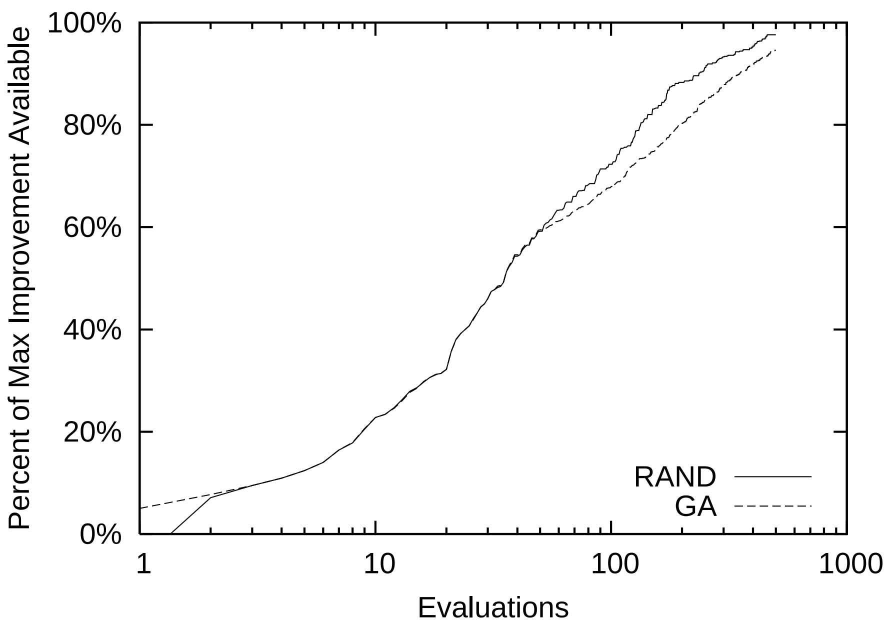


Figure 6.4: Performance with respect to evaluations for the random (RAND) and genetic (GA) search algorithms on the MIPS board.

Oracle-based models

In order to test the effectiveness of models, a perfect 'oracle' model is constructed.

Each model is contructed using the results obtained from searching a particular program's space and then tested on each model-enabled search algorithm on the *same* benchmark; we call these two learned models: *IID-oracle* and *Markov-oracle*. This allows the models themselves to be tested independently from the knowledge transference process, since each benchmark is evaluated using a model constructed from its own data.

These 'oracles' form an upper-bound on the performance we can expect to achieve when later trying to learn each model, assuming perfect knowledge transference. This helps 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. Although it is clearly not valid to assume perfect knowledge transference and to use models constructed for the very program being evaluated as is done here when reporting results, it is useful to test the effectiveness of the modelling process.

Each baseline search algorithm is compared against this same algorithm using each predictive model. For the random algorithm, instead of having a uniform probability

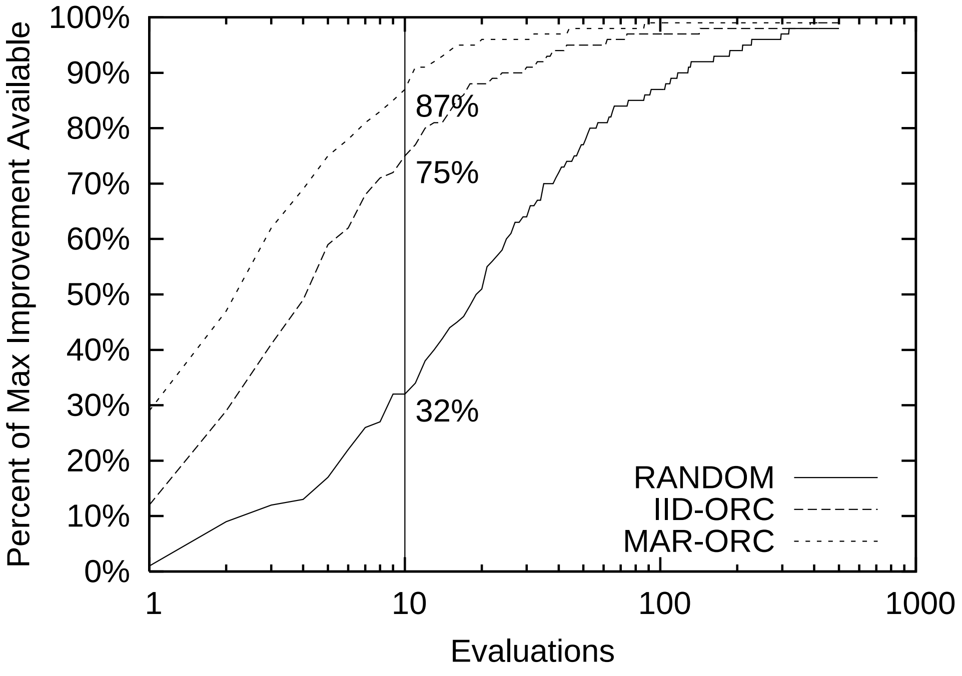


Figure 6.5: TI: Random search versus IID-oracle and Markov oracle. Results averaged over all benchmarks.

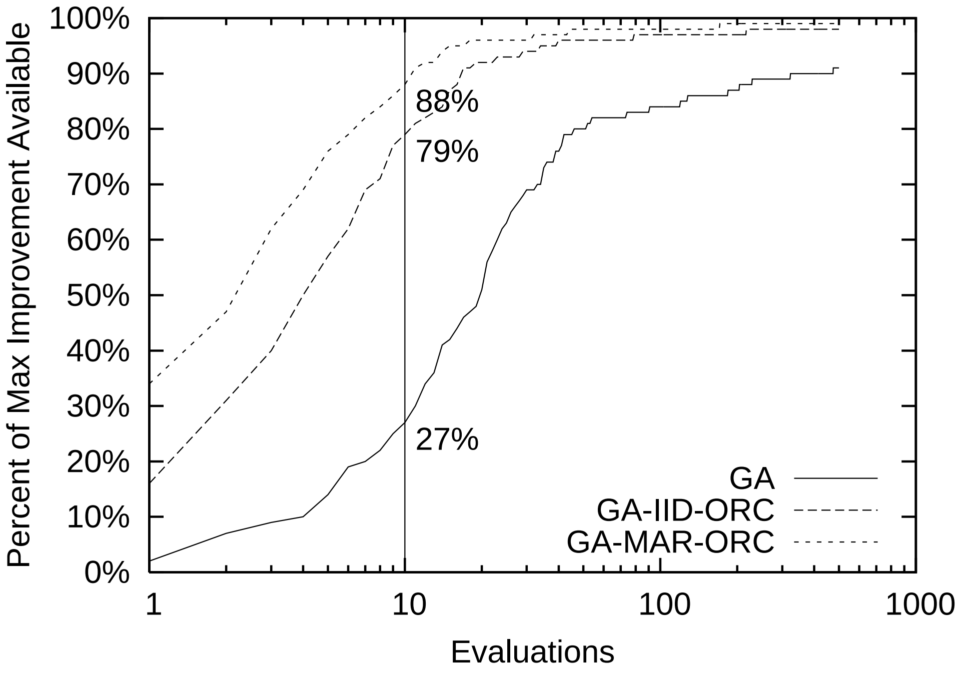


Figure 6.6: TI: GA search versus IID-oracle and Markov oracle. Results averaged over all benchmarks.

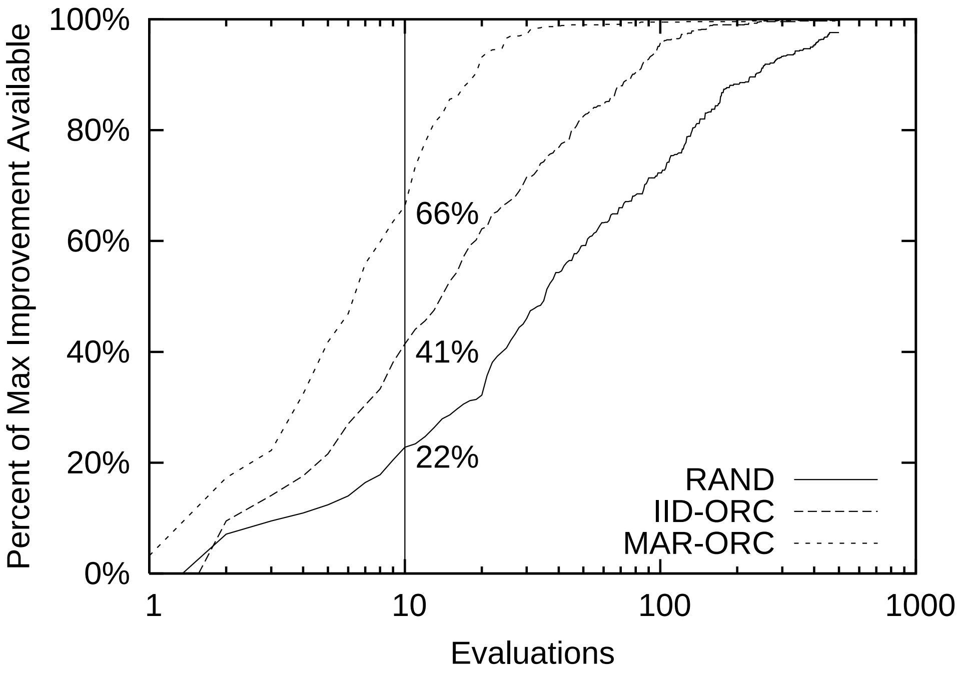


Figure 6.7: AMD: Random search versus IID-oracle and Markov oracle. Results averaged over all benchmarks.

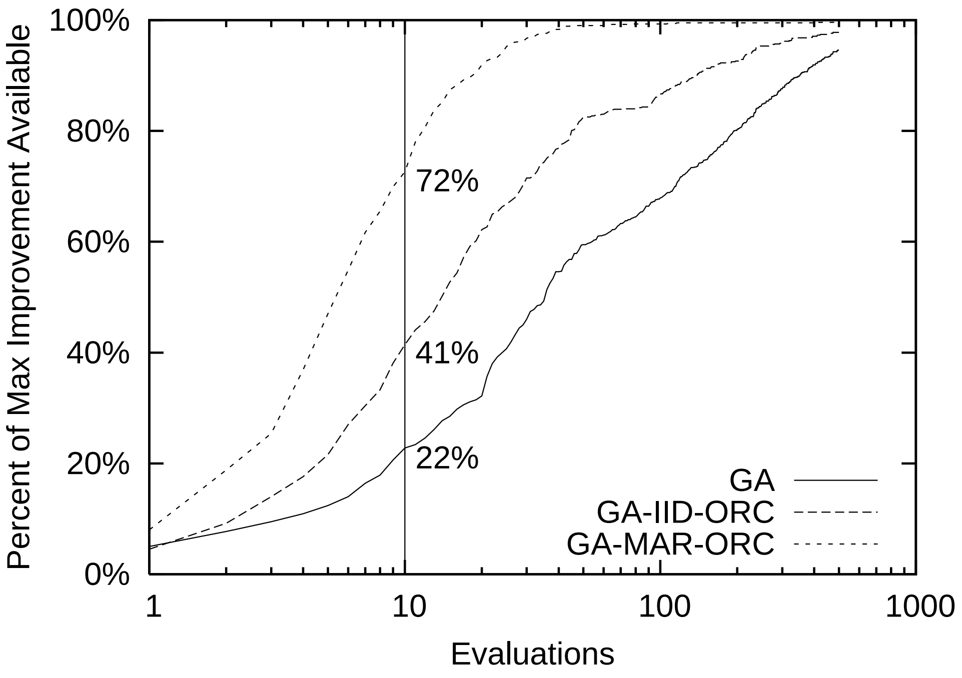


Figure 6.8: AMD: GA search versus IID-oracle and Markov oracle. Results averaged over all benchmarks.

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.

Figure 6.5 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 6.6 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 35% of the maximum available performance. In contrast, `random + IID-oracle` achieves more than 70% of the available performance and `random + Markov-oracle` achieves around 87% of the performance. Figures 6.7 and 6.8 depict 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 22% of the available performance after 10 evaluations. In contrast, `random + IID-oracle` achieves about 40% of the available performance (twice better than base) and `random + Markov-oracle` achieves 66% of the available performance. On average, the baseline algorithm needs 100 evaluations to achieve the same performance as the baseline + the Markov oracle achieves with just 10 evaluations.

From these figures, it can be seen that the IID and Markov models have the potential to dramatically improve the performance of both search algorithms. The next section describes how we can learn these models from previous off-line runs to build a predictive model.

6.6 Feature selection

The biggest difficulty in applying knowledge learned off-line to a novel input is considering exactly which portions of this knowledge are relevant to the new program. It is shown 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 learned off-line knowledge.

Initially, thirty-three loop-level features were identified, which were thought to describe the characteristics of a program well. These are given in table 6.3.

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 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 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 6.3: Features used

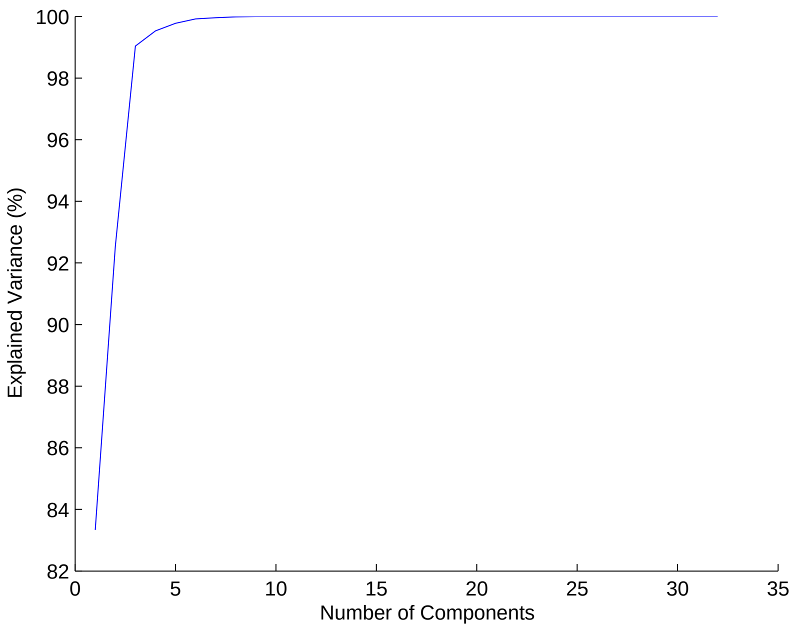


Figure 6.9: Percentage of the total information of the dataset explained by increasing number of principle components.

6.6.1 Principal Component Analysis

Obviously, the selection of these program features is critical to the success of this method, and so a well known statistical technique, principal component analysis (PCA) [6], is employed to assist the selection.

In general, any reduction in the dimensionality of a space will inevitably result in some loss of information. A good dimensionality reduction technique will preserve as much of the information that can be used to differentiate between different classes as possible. Details of PCA are given in section 4.5.5.

The 36 chosen features are used as input for the PCA process. 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 (see figure 6.9). The output of this process is a 5-D feature vector for each benchmark, containing these five condensed feature values, which be used in our nearest neighbour classifier, explained in the next section.

6.7 Knowledge Transference

Vital to the success of our modelling technique is the ability to apply the correct model to a novel program input. This section shows how the selected features can be used to gauge program similarity.

6.7.1 Nearest Neighbours

This chapter employs a nearest neighbours classifier (see section 4.5.4 and [6])to select which of our previously analysed programs our new program is most similar to. Learning using nearest neighbours is simply a matter of mapping each 5-D feature vector of our training programs (all our benchmarks) onto a 5-D feature space.

When a novel program is compiled, it is first put through a feature extractor, and those features processed by PCA, as described aboive in 6.6.1. 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 is calculated. The closest point is considered to be the ‘nearest neighbour’ 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 (see section 4.7.2), 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 neighbour. Having selected a neighbour, a previously learned probability distribution for that selected neighbour is then used as the model for the new program to be iteratively optimised.

6.7.2 Evaluating learning

It is useful to know how close our learned distribution is to the oracle distribution for both models, IID and Markov. Averaged across all benchmarks, the learned 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.

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 learned models should give significant performance improvement over existing schemes. This is evaluated in the next section.

6.8 Results and Evaluation

This section evaluates the focused search approach on two optimisation spaces. The first space is the exhaustively enumerated 14^5 space, described throughout this chapter. 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 [29]. This was achieved using the standard leave one-out-cross-validation scheme (see section 4.7.2) i.e. learn the IID and Markov models based on the *training* data from all other programs *except* for the one about to be optimised or *tested*.

6.8.1 Evaluation on exhaustively enumerated space

Initially, both the baseline random and GA search algorithms were evaluated for 500 program evaluations, and their speedups recorded, using both the TI and AMD. The same algorithms were then evaluated again in the same way, this time using the two learned models: IID and Markov.

The results for the TI are shown in figures 6.10 and 6.11, for the AMD in figures 6.12 and 6.13. On the TI, the learned IID based models achieve approximately twice the potential performance of either baseline algorithm after 10 evaluations (60%/62% vs 32%/27%) . The learned Markov model does even better, achieving 79% of the performance 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 learned Markov based algorithms achieves more than twice the performance of the baseline algorithms after 10 evaluations.

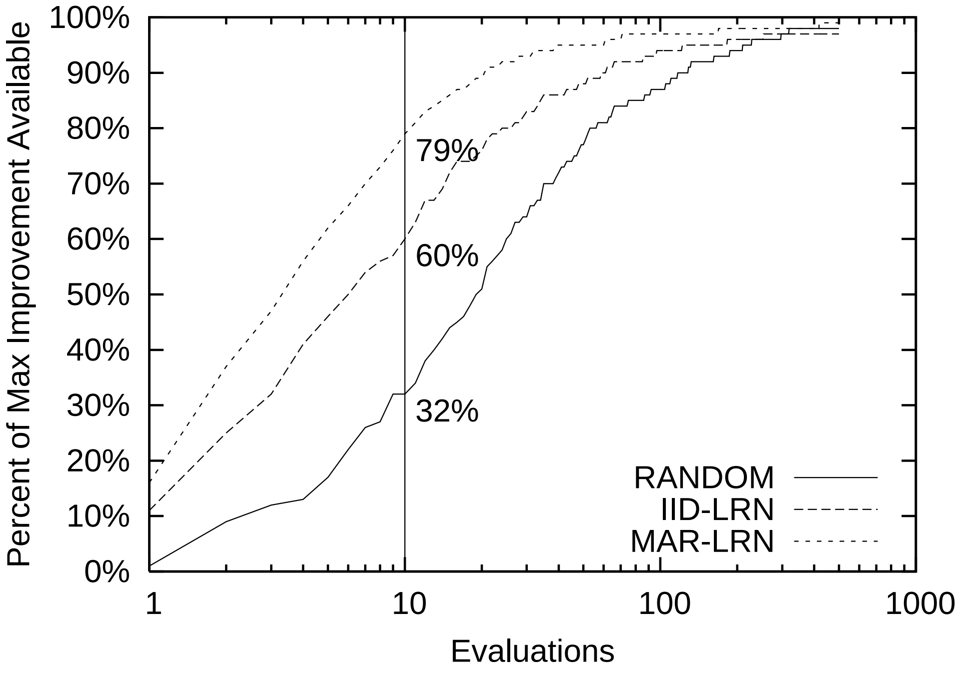


Figure 6.10: TI: Random search versus IID-learned and Markov-learned. Results averaged over all benchmarks.

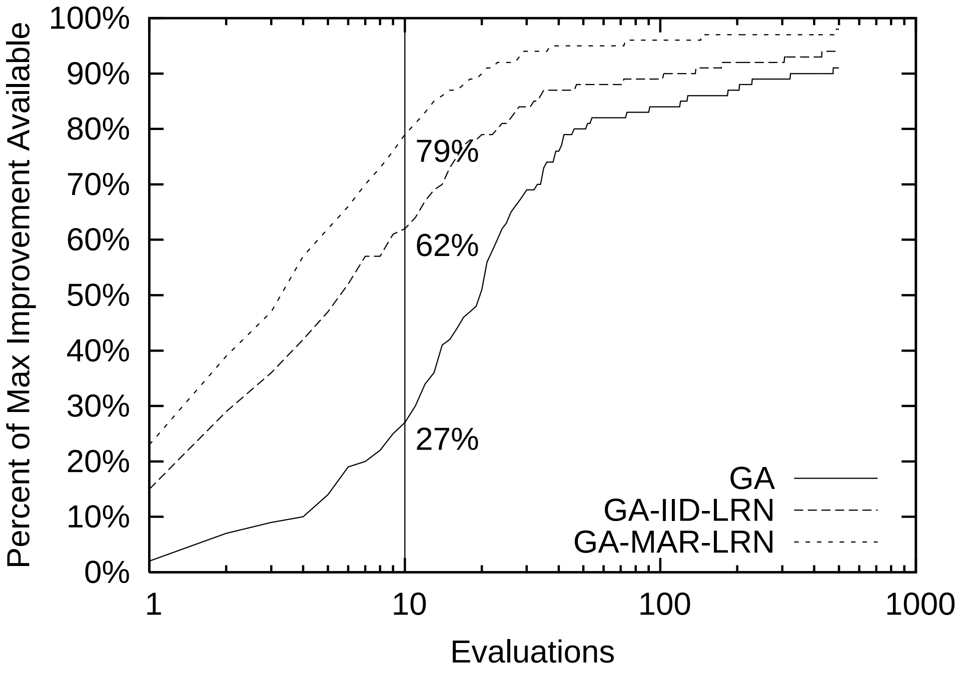


Figure 6.11: TI: GA search versus IID-learned and Markov-learned. Results averaged over all benchmarks.

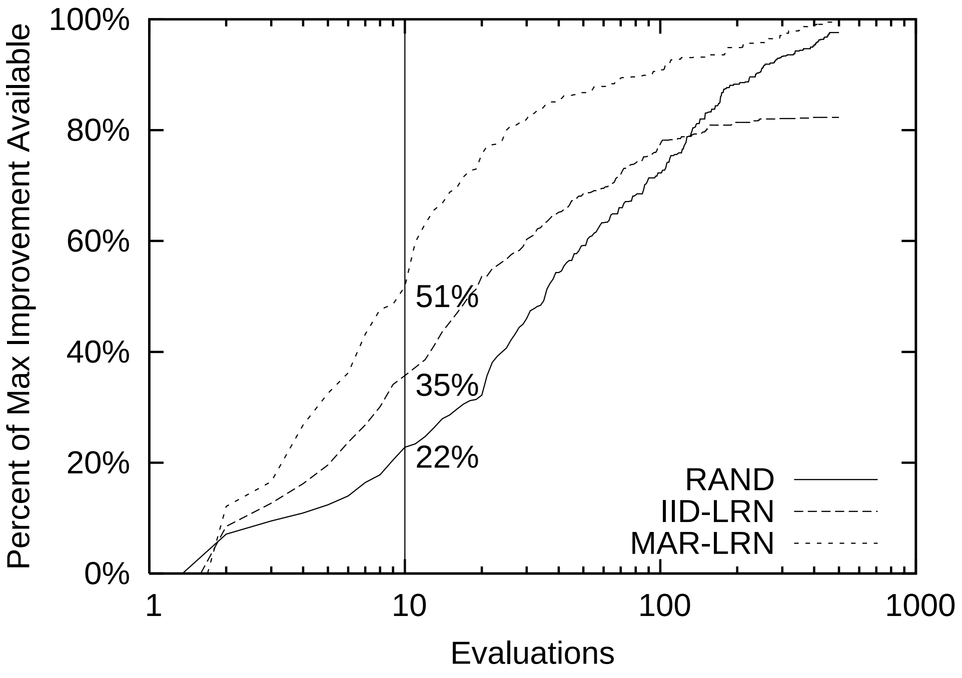


Figure 6.12: AMD: Random search versus IID-learned and Markov-learned. Results averaged over all benchmarks.

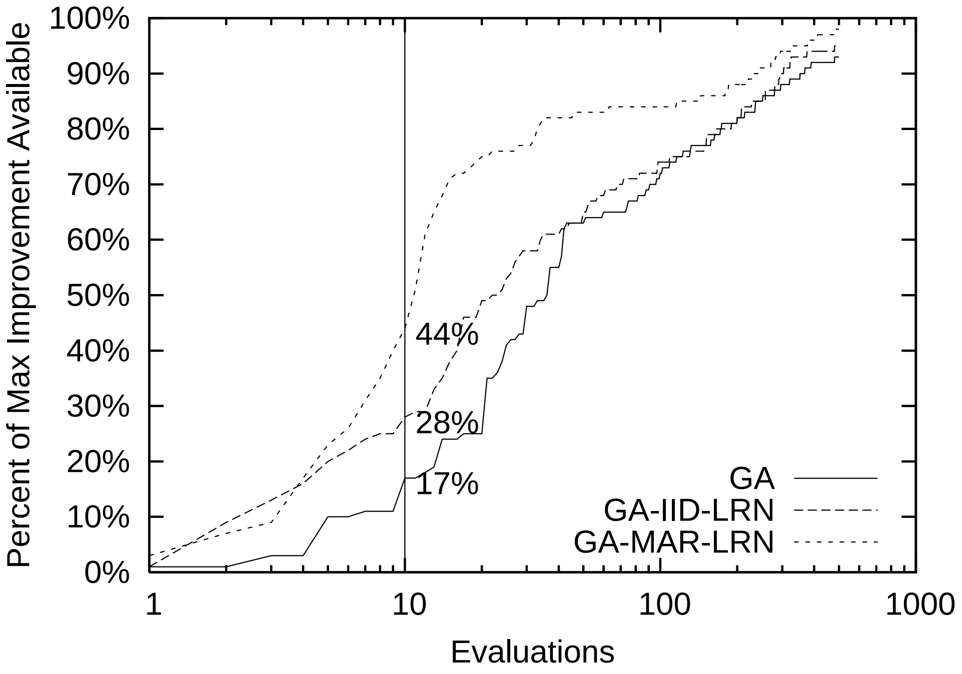


Figure 6.13: AMD: GA search versus IID-learned and Markov-learned. Results averaged over all benchmarks.

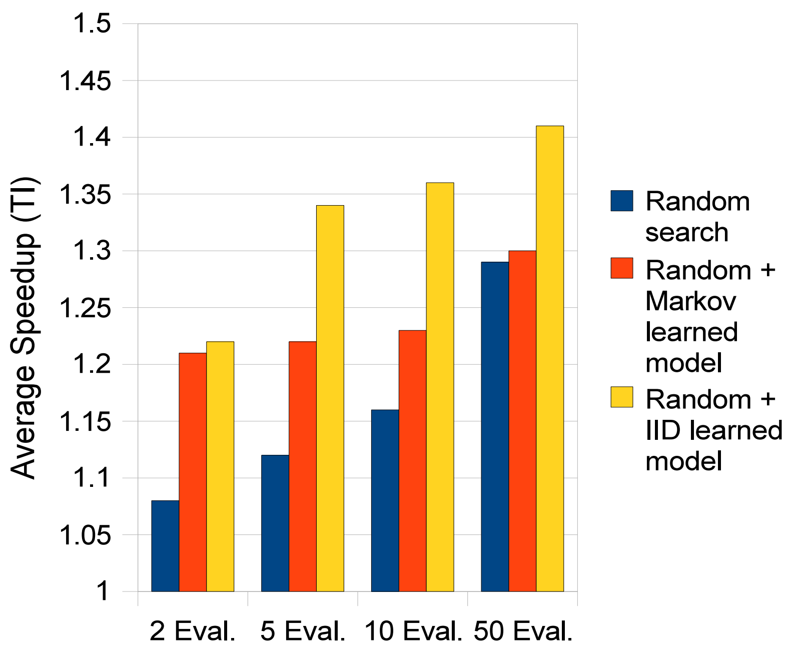


Figure 6.14: Average speedups achieved over all benchmarks on TI for random search, and the two learned models

6.8.2 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 practice when we wish to search across a large range of transformations, it is infeasible to run exhaustive experiments. Instead, a random search for 1000 evaluations is done on each program space as off-line training data.

This time the evaluation centres around the performance achieved in the early parts of iterative optimisation, and so the baseline random search algorithm and both learned models are allowed to run 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 are shown in figures 6.14, 6.15 and 6.16. Due to time constraints, only those benchmarks with non-negligible speedup on the exhaustively enumerated space are evaluated. The learned models both deliver good performance and the random + IID learned model achieves an average speedup of 1.26 after just 2 evaluations. Furthermore, the random

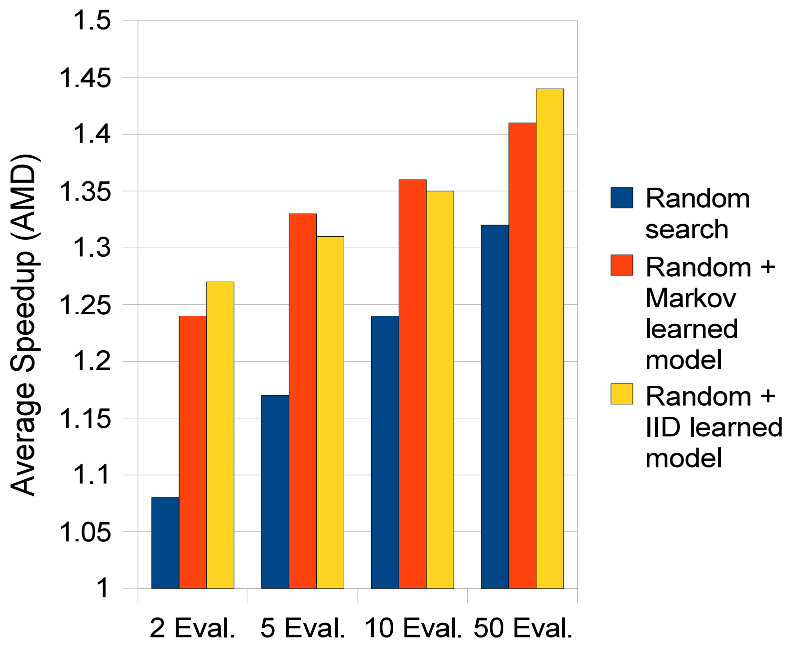


Figure 6.15: Average speedups achieved over all benchmarks on AMD MIPS for random search, and the two learned models

+ IID learned 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 learned model achieves better performance than the Markov learned model after 50 evaluations 1.41 vs 1.30 speedup in contrast to the results of the exhaustively enumerated space (see figures 6.10-6.13). The reason 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 after 2, 5, 10 and 50 evaluations on average in figure 6.15, and for each benchmark in figure 6.17. Again both learned models significantly outperform the baseline random algorithm. In fact the random + Markov learned 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

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
adpcm	1.08	1.33	1.17	1.18	1.33	1.18	1.25	1.35	1.24	1.28	1.43	1.28
edge	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 6.16: Speedups up achieved by random search (R), random + Markov learned model (M), random + IID learned model (I) after 2, 5, 10 and 50 evaluations on each benchmark on the TI processor. Random + IID learned model achieves greater average performance (1.34) after 5 evaluations than random does after 50 evaluations (1.29).

in the small space is `himc3`. 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.

The Markov predictor performs less well on the large 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 a relatively small amount of training data available. Once sufficient new data is accrued by iterative optimisation, it can be used for a second stage of learning using the Markov model.

6.9 Conclusion

This chapter develops a new methodology to speed up iterative compilation. By employing predictive modelling, we can automatically focus any search on those areas likely to give greatest performance increases, and thus dramatically reduce the number of iterations necessary to achieve a given level of performance. Program features are

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
edge	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 6.17: Speedups up achieved by random search (R), random + Markov learned model (M), random + IID learned model (I) after 2, 5, 10 and 50 evaluations on each benchmark on the AMD processor. Random + Markov learned model achieves greater average performance (1.33) after 5 evaluations than random does after 50 evaluations (1.32)

used to identify the most profitable areas of the optimisation space to search. Results demonstrate that this approach is highly effective in speeding up iterative optimisation for the embedded systems domain, but with 10 evaluations still necessary, it is not yet well suited to the general purpose domain. The logical extension of this work is to cut the number of evaluations right down to just one – making it no longer a search-based system, but a *smart compiler* which competes with, and surpasses a traditional compiler, with little or no extra time/resource outlay. The next chapter will demonstrate a methodology which achieves just that.

Chapter 7

Learning More Efficiently

This chapter presents a method for dramatically reducing both the one-off training time required to initialise the compiler, and reducing the number of compile-time iterations required down to just one, bringing the utility of this approach into the general purpose world. This is achieved by removing the inefficiency in learning and searching by focusing on the programs which best characterise the optimisation space of all programs.

In section 7.1, clustering is introduced as a means to gain coverage; in section 7.2, the reasons why previous techniques have been inefficient are presented; section 7.3 details how these problems can be tackled by statistical techniques; in section 7.4 the experimental set up is explained; section 7.5 presents empirical results and analysis thereof, and section 7.6 draws some brief conclusions.

7.1 Introduction

As has been seen in chapters 5 and 6, it is possible to obtain considerable improvement in execution speed by applying search and learning techniques to the problem of transformation selection – however these techniques have considerable drawbacks. Primarily, they take a long time to initialise; the training time for the system is a significant cost, which can run to weeks, or even months. Secondly, the number of iterations required to achieve significant performance gains at compile time is still too large for many domains.

Although the work presented in chapter 6 improves on the number of iterations required significantly, the 10 compile iterations still needed leaves it unsuitable for use in most general purpose compiler work. In addition, even in the embedded domain the traditional compiler paradigm – only compiling once with no feedback – is still by far more frequently used than any iterative technique. I call this paradigm ‘one-shot compilation’.

The partitioning of the feature-space and its effectiveness in selecting the best subset of programs to learn on is at the heart of this chapter. It is demonstrated that, by standing back and using unsupervised learning before embarking on a time and resource expensive iterative compilation search, we can significantly reduce the time and effort required to train a smart, learning compiler.

Additionally, we show that by gaining a greater coverage of the whole space of programs in our training data, we can dispense with search altogether, and produce a simple-to-use, one-shot compiler that gives excellent performance, exceeding the maximum O3 level of the compiler by 14% on average across all benchmarks considered.

7.2 Reasons for Inefficiency

In the approach taken in chapter 5, each program is considered individually. The probabilistic model of the optimisation space is updated online as the search continues, and then discarded at the end of the process. The inefficiency of discarding this information is discussed in section 6.2 and remedied in section 6.5 of that chapter by employing models to allow transference of knowledge of the optimisation space between different programs.

However, there still exists considerable inefficiency in the learning process. This is due to the often arbitrary nature of benchmarks – when we learn a model for a platform, we want to learn about as much of the optimisation space as possible, and, given a constrained amount of learning time, it is unclear as to how to best focus our efforts. If it is the case that two sections of program code are virtually identical, then learning two separate models for such a scenario is futile at best, and could actively inhibit some learning algorithms. Alternatively, if a new program is given to a pre-initialised system for analysis, and that program is significantly different from what the models in the compiler have used as their learning data set, then the result is unlikely to be a positive one.

In order to counter this, a statistical approach to the selection of programs is used to build the models at the learning stage of the compiler.

7.3 Learning what to Learn

In order to make the best use of the available time for training a smart compiler, we must consider which programs to train over. In chapter 5, we considered only one program at a time; in chapter 6, we proposed a scheme for allowing transference of knowledge about one program to another – but in this case we had no choice as to which programs we could use for training. The limited benchmark set used in chapter 6 necessitated the use of the entire benchmark suite for the purpose of training the compiler, and so we might consider the distribution of the points chosen as essentially random (or at least random with a bias towards the benchmarks chosen by the compiler of the suite), or at the very least, arbitrary.

When considering a larger benchmark suite, it is not possible to take this approach because of the huge amount of time and resources it would require to train on each of the programs in the suite. Indeed, even if one were somehow able to make enough time and resources available for such an action, the space of programs considered would still be limited to the space defined by the benchmark suite, and not the infinite number of possible programs one might conject.

Thus it is clear that a method is necessary to select which programs are the best candidates for use in training. This chapter proposes using clustering of the feature-space to achieve this. Using this technique, we can consider a large number of programs. These programs are then partitioned according to their features into different clusters, which broadly share similar features; by selecting the most typical program from each cluster and using these to train our smart compiler, significantly greater speedup is achievable than randomly selecting the programs.

7.3.1 Features and Feature Extraction

The features used in this chapter are the ratios of each assembly level instruction to the total number of instructions executed by a program, i.e. the proportions of each type of instruction used. This is a very simple feature set, which is easy to capture, and yet provides excellent performance. In addition, we use only ARM assembly instructions as our features, even through we evaluate on an Intel x86 processor. This is to ensure that our technique is properly capturing information about the program, and not some facet peculiar to a particular architecture. Hoste and Eeckhout [30] argue that of a generic RISC architecture is capable of representing and characterising program performance better than x86, and the ARM is used as an approximation a generic RISC core.

Feature Extractor

Features were extracted by using the simulator SimIt-ARM v2.1 [55, 56]. The benchmarks were compiled using a GCC v3.3.1, and then ran through the simulator. The simulator counts the number of machine instructions used by each benchmark, which we use as the basis for our features. The use of GCC ensures that almost all code will run through the simulator, making it relatively easy to extract features.

A pertinent question at this point is: why use a simulator to extract features? The answer is that this is mainly due to time and resource constraints.

Primarily, using a simulator is an efficient use of time in infrastructure work – the approach used in chapter 6 requires the use of the SUIF compiler [29] from Stanford, which is old and unmaintained piece of software, which only accepts the ANSI C-89 standard. It therefore requires a considerable amount of work to make each program in a modern benchmark suite compatible with SUIF. A new feature extractor could be written from scratch, but this again is prohibitively expensive in terms of time.

The feature extraction stage should be considered as a lightweight profile stage, taking very little time on a modern, fast machine. Although a simulator is normally a slow tool for feature extraction, in principle, there is no reason why these instruction counts could not be captured by an extremely lightweight and fast profiling tool – modern JIT simulators can run significantly faster than real hardware[60] – and therefore we assume this feature capture stage is fast.

A simulator also provides much additional information about a program not available to a lightweight profiler, such as the efficacy of the cache lines and information about the pipeline, etc., so all such information from the simulator output is discarded, and does not form part of the feature set. The simulator is simply used as a shortcut to

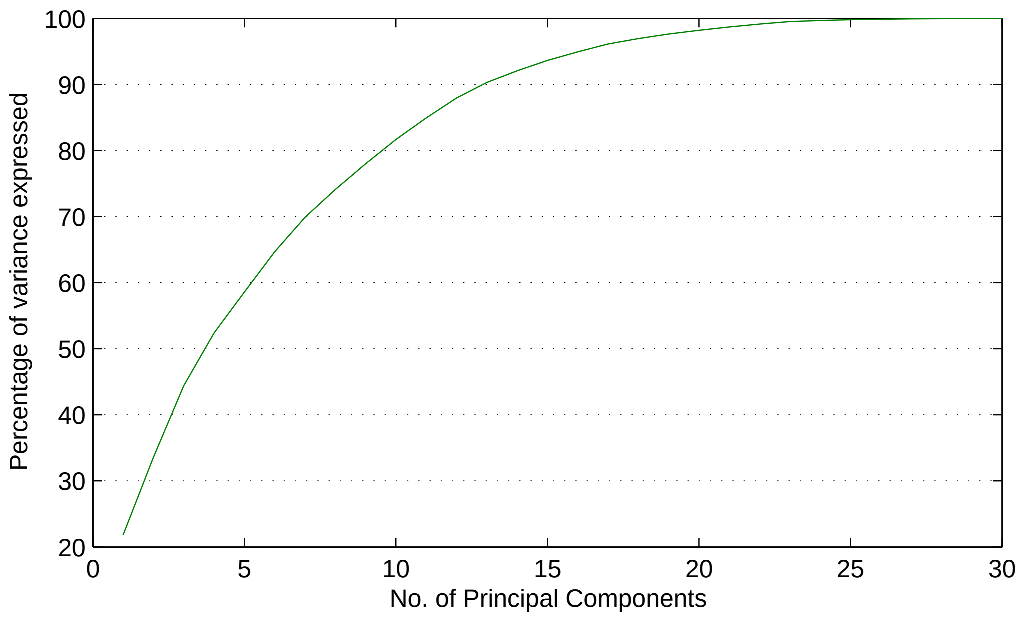


Figure 7.1: Percentage of the total information of the dataset explained by increasing number of principle components

obtain this information, without having to write a tool for this purpose.

Indeed, it is crucial to the success of this work that the analysis of a large number of programs is fast as it relies on observing more programs than has been attempted before.

Feature reduction with PCA

These features were then further reduced in number, using a technique called *Principal Components Analysis* (as was used in chapter 6). This technique reduces the dimensionality of the feature space by examining the variance within the data, and while preserving as much variance as possible, generating a new set of features which are a linear combination of the original set. These resulting features are called Principal Components (see section 4.5.5 and [6]).

It is possible to chart how much of the variance of the data is expressed with each additional principal component, and this is shown in figure 7.1. Here we can see that 80% of the variance of the data can be expressed in just 9 principal components, and thus we use these 9 principal components as our features. By expressing the features in this way, we can attribute less importance to features which are highly correlated and thus may skew the results.

7.3.2 K-means clustering

The feature-space is clustered using the *k-means* algorithm, which is a simple stochastic technique [6]. K-means clustering was selected because it is the simplest of the clustering algorithms, and it makes sense to try the simplest first. The input to the algorithm was the reduced feature set of 9 principal components, as described in section 7.3.1. Since the results of the k-means algorithm contain an element of randomness due to the initial placement of the cluster centroids, the algorithm was executed 50 times, which is enough to ensure the selection of the centroids with the lowest total intra-cluster variance to maintain replicability of the experiment. We employed the Euclidian distance metric to test for similarity, which is simply result of a standard difference-squared distance equation between two points, over *n* dimensions (9 in this case).

The algorithm treats the feature-space as a continuous space, where in reality it consists of discrete points. For this reason, the program with the smallest Euclidian distance between itself and each centroid is chosen as the archetype for each cluster.

Selecting the correct number of clusters

The k-means technique cannot determine the correct number of clusters which most accurately depict the space, which must be supplied *a priori*. This presents the problem of how to choose how many clusters to represent a complex high-dimensional space. This is a well known and difficult problem, and is a subject worthy of significant research on its own merits. Since this is not the primary purpose of this work, we employed the technique suggested by Ray and Turi [53] which builds on the simple premise of considering the proportion of the intra-cluster variance in respect to the inter-cluster variance, and selecting the first local minimum of this value as the number of considered clusters increases.

Intra-cluster variance can be defined as:

$$S_{intra} = \frac{1}{N} \sum_{i=1}^{K} \sum_{x \varepsilon C_i} ||x - z_i||^2$$

and inter-cluster variance as:

$$S_{inter} = min\left(\left\|z_i - z_j\right\|^2\right), i = 1, 2, ..., K-1, j = i+1, ..., K$$

Full details of the algorithm are provided in their publication[53].

The first local minimum encountered using this technique was at 6 clusters, and this is the value used throughout this chapter. This postulates that the optimisation space considered in this chapter consists of 6 distinct regions which share similar characteristics.

7.4 Experimental work

These experiments were carried out on an Intel Core 2 Duo E6750 processor running at 2.66GHz. The machine was running a stripped down version of Ubuntu Linux 8.04 with linux kernel version 2.6.24. The compiler used was the MILEPOST [25] version of the GCC compiler version 4.2.2, which allows additional optimisations over and above the default GCC to be accessed via compiler flags. The timing was carried out using the CCC optimisation framework, also part of the MILEPOST project. The techniques presented are evaluated on the EEMBCv2 [19] benchmark suite. When a choice of dataset was offered by EEMBC, the default dataset was chosen. A few programs were excluded due to difficulties with the MILEPOST GCC compiler.

The following sections describe each experimental approach algorithmically:

7.4.1 Cluster-based Approach

Evaluation of the cluster-based approach was carried out as follows:

1. Features are extracted from each of 44 programs in the benchmark suite (see section 7.3.1). The features are then reduced to 9 principal components, using the PCA technique.

2. The feature-space is then clustered into 6 clusters, and the most typical programs selected for each cluster, one for each (see section 7.3.2). Leave-one-out cross-

validation (see section 4.7.2) is used, excluding each considered benchmark in turn.

3. These 6 benchmarks are executed 4000 times, using random optimisation flags in a similar manner to previous chapters. The set of flags providing the best performance for each of these benchmarks is recorded.

4. Each benchmark's features are inputted into the clustering model, which now has 6 fixed cluster centroids. It then assigned to a cluster by considering the cluster centroid nearest in Euclidian distance. This can be considered similar to the nearest neighbours approach employed in chapter 6.

5. Having been assigned a cluster, the benchmark is compiled and executed, using the best performing compiler flags associated with its cluster, and the execution time recorded. The benchmark is also compiled and executed using the O3 optimisation setting on the compiler as a baseline.

Although cross-validation (see section 4.7.2) is employed, its use or otherwise does not affect the outcome of the experiments in this case, as removing a single point from the feature-space does not affect the partition boundaries sufficiently to cause a change in the classification of any benchmark represented in the space. This is confirmed by the empirical data.

7.4.2 Random Approach

Given that a limited amount of time and resources is available to train a smart compiler, there must be some way of determining which benchmarks to use for learning in a large benchmark suite. The most obvious is simply to choose randomly.

Evaluation of the random approach was carried out as follows:

1. 6 benchmarks are randomly chosen from the set of 44, and their features extracted as above.

2. These 6 benchmarks are executed 4000 times, using random optimisation flags in a similar manner to previous chapters. The set of flags providing the best performance for each of these benchmarks is recorded.

3. Each benchmark's features are inputted to a nearest neighbour classifier, plotting the randomly selected programs as potential neighbours in the feature space. It

can be then assigned of the 6 random points by considering the point nearest in Euclidian distance.

4. Having been assigned the nearest random point, the benchmark is compiled and executed, using the best performing compiler flags associated with that point, and the execution time recorded. The benchmark is also compiled and executed using the O3 optimisation setting on the compiler as a baseline.

5. This process is then repeated 1000 times to minimise the effect of randomly choosing particularly good or bad points.

7.4.3 Iterative Approach

The iterative approach is included for purposes of comparison. This is a typical iterative optimisation [26] implementation where each benchmark is executed 4000 times, using random optimisation flags, as is done in the training stages of the previous two approaches. The best execution time found is recorded. Using this value, it is possible to see the potential for optimisation each program has.

7.5 Results and Analysis

The results of two experimental approaches are presented in the table below. Firstly, the results of our cluster-based smart compiler, where 6 cluster centroids have been chosen by analysis, and in comparison, a smart compiler which uses an equal number of randomly selected points. Additionally, the best available result found using standard iterative compilation over 4000 runs is presented to show to scope for improvement available for each benchmark. All speedups are given relative to the standard GCC compiler with the highest O3 optimisation level enabled.

7.5.1 Results table

	Speedups		
Benchmark	Clustered Approach	Random approach	Iterative Search

	Speedups		
Benchmark	Clustered Approach	Random approach	Iterative Search (4000 runs)
a2time01	0.96	0.99	1.05
aifftr01	1.17	1.18	1.45
aifirf01	1.05	0.67	1.18
aiifft01	1.04	1.03	1.29
basefp01	0.92	0.90	1.04
bitmnp01	1.02	0.90	1.07
cacheb01	1.48	1.23	1.72
canrdr01	1.02	0.86	1.32
idctrn01	1.09	0.66	1.14
iirflt01	0.99	0.96	1.11
matrix01	1.27	1.14	1.57
pntrch01	1.26	0.66	1.24
puwmod01	1.89	1.33	1.89
rspeed01	1.18	1.15	1.36
tblook01	1.31	1.14	1.44
ttsprk01	1.22	1.02	1.24
cjpeg	1.08	1.07	1.18
djpeg	1.05	1.09	1.28
autcor00	1.39	1.63	1.80
conven00	0.96	0.85	1.15
fbital00	0.96	0.93	1.13
fft00	1.20	1.23	1.39
viterb00	0.94	0.85	1.20
ospf	1.05	0.95	1.23
pktflow	1.52	1.34	1.50
routelookup	0.93	0.89	1.08
beizer	0.96	0.98	1.20
dither	1.19	1.07	1.31
rotate	0.99	0.98	1.02
text	0.98	0.96	1.02
aes	0.75	0.81	1.12
cjpegv2	1.05	1.11	1.21

	Speedups		
Benchmark	Clustered Approach	Random approach	Iterative Search (4000 runs)
djpegv2	1.09	1.19	1.41
huffde	1.09	1.03	1.23
mp4encode	1.03	0.98	1.13
mp4decode	0.90	0.88	1.06
rgbcmy	0.97	0.99	1.18
rgbhpgv2	0.93	1.00	1.22
rgbyiqv2	1.06	1.10	1.34
mp3player	2.71	1.41	2.65
tcp	1.09	0.92	1.10
ip reassembly	0.99	0.99	1.05
ospf	0.97	0.98	1.08
ip pktcheck	1.26	1.07	1.25
AVG	1.14	1.02	1.29

On average, using just one evaluation, our clustering-based approach yields a speedup of 1.14 over the whole benchmark suite. This compares to a speedup of only 1.02 if the smart compiler uses points selected at random.

The correctness of each of these best points was verified using the EEMBC internal verification, which compares the produced output to the desired output.

7.5.2 Analysis

Our clustering-based approach performs significantly better than the random selection approach because it represents the space of programs much better. By considering a large number of programs before any training occurs, we can successfully choose which programs to use to train on. Since the space is not uniform, a random selection is likely to bias itself towards selection of overly represented sections of the program-space, whilst neglecting others. In this very complicated space, such disparity between the actual program-space and what has been chosen to represent it inevitably results in poor performance.

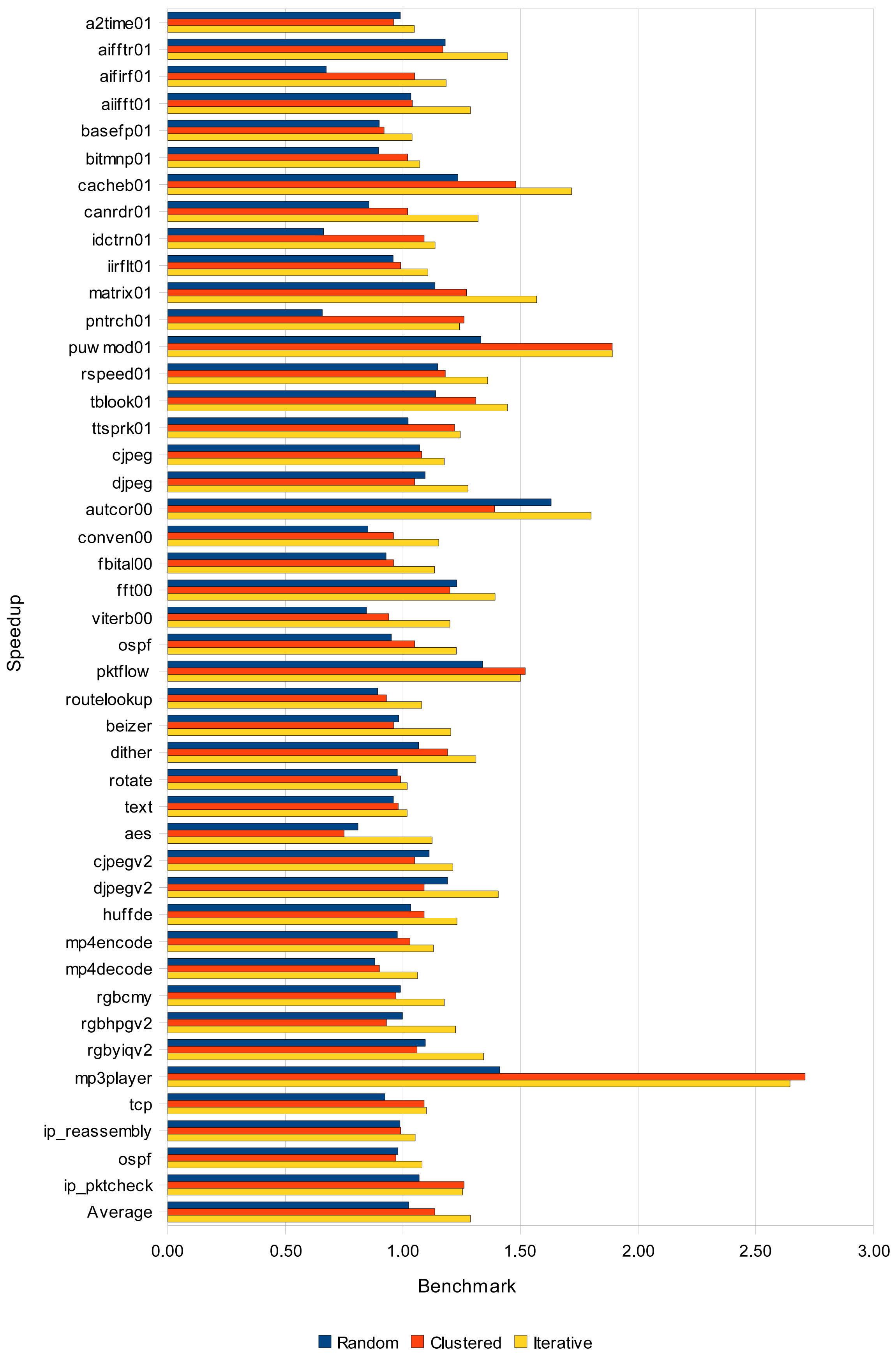


Figure 7.2: Results for random, clustered and iterative approaches

There are benchmarks for which our clustering trained smart compiler fails to beat, and in some cases does worse than, the baseline. While regrettable, this is not surprising. Both compilers are taking a very difficult problem – the problem of program optimisation – and giving the result most likely to be good according to some internal model. However much pre-analysis is put into this problem, there is likely to be an element of randomness in the output, resulting in occasional decreases in speed for a small number of programs, even when using a generally more accurate model. This pattern can also be observed by comparing two commercial compilers for the same platform. The important aspect is that the clustering based smart compiler performs significantly better on average.

It might be argued that one may wish to avoid the chance of a heavy drop in performance, as is seen in the *aes* benchmark, and thus should choose the baseline over our smart compiler, but even this is not so. The use of the GCC O3 optimisation level as 1.00 – the baseline – is entirely arbitrary. It is possible to redraw the results table using our smart compiler as the baseline, and instead comparing the GCC O3 performance to this; given these values, it would be obvious that far larger performance drops would occur by switching from our smart compiler back to GCC at O3.

Some particularly large speedups are achieved, such as 2.71 for *mp3player*, and 1.89 for *puwmod01*. This is likely to be because of the very kernelised nature of these codes, where changing a small section of code which is frequently used can have a large impact on the resulting speedup. These codes also seem to be particularly amenable to optimisation using particular loop unrolling factors.

There are also some benchmarks which perform poorly using clustering smart compiler, such as *aes*, which significantly slower than the baseline at O3. This is likely because of the unusual coding style often inherent to encryption algorithms which is not captured by our features. Optimisations which are useful other programs near to *aes* in the feature space may in fact inhibit performance due to this unusual style.

This means that, on average, we can achieve slightly under half of the performance improvement attained by iterative optimisation using 4000 runs, in just a single evaluation. Additionally, we achieve a 700% increase in the additional optimisation possible by using our clustering-based approach rather than random selection, which shows itself not to be a viable option when no search of the space is allowed (indeed, as was also shown in the previous chapter).

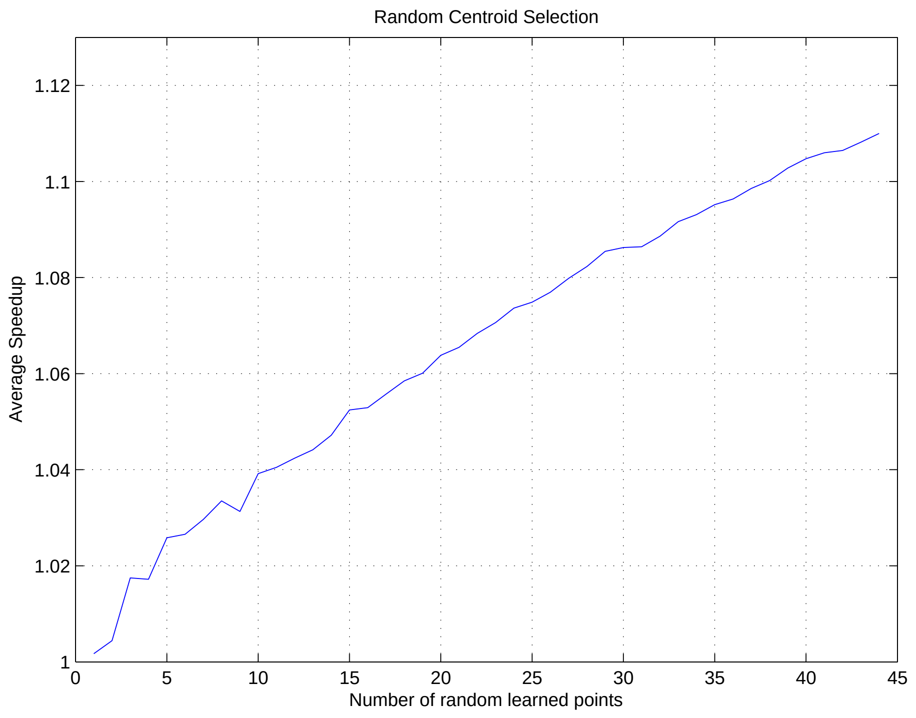


Figure 7.3: Speedup achieved by using learning with an increasing number of randomly selected programs

7.5.3 Increasing learning time and the efficacy of features

As shown above, training a smart compiler using only 6 randomly chosen points yields very poor results when allowing only one evaluation (as opposed to search in the previous chapter). This, as has already been said, is the result a poor representation of the space of all programs. However, the number of randomly chosen points, 6, was only considered as a fair comparison to the 6 cluster centroids chosen by the clustering approach, and more, or indeed, fewer, can be considered:

Figure 7.3 shows how increasing the number of randomly selected programs used for training the smart compiler increases the performance of the compiler. The experiments were carried out in the manner described in section 7.4.2 except using different numbers of points. Two important conclusions can be drawn from this: firstly, that in order to achieve performance getting even near to our cluster based approach, we need a very large number of programs to train on – 36 for a 1.10 speedup. This translates as 120,000 extra training runs when compared to using 6 clustered points (assuming 4000 runs per point).

The second, and possibly more important conclusion is as follows: the features used in this work are indeed indicators of how to optimise a program. It is difficult to quantify how good these features are as we have no other similar data to compare it to, yet the fact we can empirically show a correlation between closeness in terms of Euclidian distance in the feature-space and the speedup obtained using a smart compiler is an important one in itself, as it shows the features used here really make a difference.

If proximity in terms of distance in the feature-space made no difference to how a program should be optimised, we should see no difference when training over an increasing number of random points. Increasing the number of random points selected for training decreases the average distance between any point in the space and its nearest training point. The fact that this decrease in distance correlates very well with an increase in performance indicates the features are performing well.

In machine learning, unsupervised, feature-only based techniques are usually validated by success in empirical experiment, and this has been achieved in this chapter.

7.5.4 Applicability of results

These experiments were performed on a general-purpose architecture, the Intel Core 2 Duo. It is clear how single evaluation compilation is useful in the general-purpose domain, however it is also useful in the embedded domain. Indeed, even though iterative and search-based techniques have been available for years in the embedded domain, their use is the exception rather than the rule. There are a number of good reasons for this, including the difficulty in setting up search-based and iterative techniques, lack of track-record producing a lack of trust, and the time and resources required. The technique presented here is simple and easy to deploy, only requiring training in the production stage of the compiler.

This work is relevant to the embedded domain in two important ways: firstly, it useful by its own merits for the reasons given above and, secondly, it can be used as the starting point for search-based techniques like the one proposed in chapter 6 – time constraints prevented any such experiments in this work, but would be interesting as future work.

Although there is no guarantee this work on a general-purpose processor is transferable to the embedded domain, I believe we can and must take it as a ’proof of concept’.

In reality, this experiment would be very hard to carry out on any embedded architecture. This is because of the very large number of experiments required to prove the effectiveness of the technique (over 176,000 were carried out in this work). If such an experiment were attempted on even a real embedded processor, yet alone a cycle-accurate simulator, the time and resources required would quickly make the project infeasible.

However, the main reason why so many experiments are necessary to show the technique works is not because of the time taken by the actual cluster-based program selection proposed, but because of the multiple random selections needed to show the improvement is not down to luck. If one accepts the ‘proof of concept’, then an embedded smart compiler could be trained on this benchmark suite using only 24,000 training runs – a reduction of over 152,000. Indeed, the use of the ARM instruction set as a basis for the features in this chapter which is evaluated on x86 indicates this technique can work across platforms.

7.6 Conclusion

We have demonstrated that, by clustering in the feature-space, we can dramatically reduce the amount of training required to achieve good performance using a smart compiler, a reduction of over 120,000 runs. By cleverly selecting the training data to be used, we can much better characterise the program-space with a small number of points, rather than randomly selecting them.

In addition, we have shown that a smart compiler trained in this way gives an average of 1.14 speedup on the EEMBCv2 benchmark suite over the O3 baseline in just one evaluation. This was achieved by training on only 6 programs. We have further shown that instruction ratios are useful features to use when considering this problem, and that these features work across two different architectures.

Chapter 8

Conclusion and Future Work

This thesis has presented a new approach to constructing a compiler optimiser. Instead of using expert knowledge, which is often based on rules of thumb, intuition or unquantifiable past experience, this approach relies on statistical evidence upon which to base optimisation decisions. The method has proven highly effective in producing significant improvement in execution time over heavily developed compilers, both proprietary and open source, and in addition, significantly speeds up the process of building a good optimiser for a new platform.

This chapter presents a a summary of the work achieved in section 8.1, an evaluation of the work in section 8.2 and a look ahead to possible directions for future work in section 8.3.

8.1 Contributions

The number of different options available to a compiler has been shown to be truly vast (see chapters 5 and 6). It is clear that it is not possible to search all, or even much more than a tiny fraction of this compiler optimisation space. Therefore, a compiler engineer must consider a strategy for selecting the best optimisations, and the order in which these optimisations apply. This thesis argues for a strategy employing the power of statistical analysis through the use of machine learning to accomplish the task of searching this n-dimensional, highly non-linear [14, 26] search space.

This thesis has addressed the issues of improving the performance of optimising compilers, and producing general, fast and inexpensive methods for tuning a compiler's optimisation stage, capable of customising itself to a variety of different architectures. Using machine learning to control its optimiser, statistical information about the behaviour of programs can be analysed, evaluated and exploited by a compiler, enabling it to make performance gains.

Further, this thesis suggests a means to bring the functionality of machine learning based compilers toward the level of a traditional compiler, allowing the user to gain the utility of statistical analysis without the need for search. This is achieved by allowing the consideration of a larger and more representative training space.

8.1.1 Intelligently searching the optimisation space

Previous work [7, 26] in iterative compilation recognised the potential to outperform traditional compiler heuristics by randomly searching through the optimisation space – while this met with some success, it was at the cost of extremely long compilation and evaluation time.

This thesis has proposed a probabilistic method to search the optimisation space more effectively and gain additional speedup, concentrating on profitable areas and steering away from code transformations which cause slow down, or no gain.

A probability vector is created, representing the likelihood of each transformation being chosen in a particular evaluation, and this is updated constantly, using runtime feedback, to allow the search to focus. This is combined with a random search, so as to avoid becoming stuck in local minima. Using this technique, a speedup of 1.71 is achieved over the UTDSP benchmark suite on average, outperforming previous work.

8.1.2 Using prior knowledge

Probabilistic search helps a compiler focus on the good areas of the optimisation space, but it does not do so quickly. This is because the knowledge of which optimisations are profitable to apply on a single program is simply discarded at the end of compilation, and the process must start over from scratch on a new program.

This thesis has proposed a system to capture this knowledge, and transfer it to new, unseen programs, allowing the same performance gains to be made in fewer evaluations.

This knowledge transference is achieved by employing code features (see 4.3) to characterise each program. Code features are used as a metric to compare each program's similarity, and thus the likely applicability of similar code transformation. Knowledge capture is achieved by employing models to represent the success, or otherwise, of transformations or transformation sequences for a particular program. Using both of these techniques together, this thesis shows how the number of iterations required for equivalent performance can be reduced by an order of magnitude.

8.1.3 Eliminating search, characterising the program space and selecting benchmarks

Although some work has been done on non-search based statistical compiler techniques [57, 43], this research concentrated on single parameter tuning (like loop unrolling options) or evaluating two optimisations. The optimisation spaces examined in these tasks do not suffer from the kind of combinatorial explosion seen when a large number of transformation options are evaluated, nor do they give the same scope for improvement. This thesis has presented a one-shot compiler solution, based on statistical analysis of the program space. No search occurs and the compiler must decide on an optimisation strategy based purely on prior knowledge, with no feedback.

Generating the initial training data for a learning compiler as seen in chapter 5, and in [14, 15], is time-consuming. The selection of benchmarks for training has not been considered before, and instead, all available benchmarks in relatively small suites have been used, using the same suite for evaluation of the technique. This is for two reasons: firstly, because the effort required to get benchmark suites running through proprietary compilers and tools is not insignificant, and secondly, because the time available to train the compiler is limited, and thus only a small number may be considered.

This thesis has proposed using unsupervised learning to circumvent these two issues. Unsupervised learning considers the feature space of programs, without the final objective function – in this case, runtime – and so the cost of adding a new program to the training set is simply that of extracting the features; there is no need to run the program through the real hardware. This dramatically reduces the cost of adding a new program to the training set, allowing a greater number to be considered, and, in turn, increases the training set's coverage of the program space.

Using a simple form of unsupervised learning called k-means clustering, this thesis has shown how a learning compiler can best make use of its training time by concentrating on those programs which best represent the wider program space. By exploiting this extra coverage, the learning compiler was able to achieve a speedup of 1.14 on average across the EEMBCv2 benchmark suite, in just a single evaluation. This represents just under half of the speedup possible when using iterative search for 4000 evaluations.

The learning compiler required only 24,000 training runs in the training stage to achieve this result. In order to approach this result using random benchmark selection, 120,000 extra training runs were required. Additionally, when both approaches were limited to just 24,000 training runs, we achieved a 700% increase in the additional optimisation possible by using our clustering-based approach rather than random selection.

8.2 Critical Analysis

The major goal of this thesis was to increase compiler performance by means of better optimisation strategies, which reduce execution time of benchmarks. This goal has been achieved, however there are additional costs that must be borne – in chapters 5 and 6, improvements in program execution time are traded off against longer compilation time. In addition, a profiling stage is necessary to provide feedback, adding additional infrastructure to a compiler toolchain, and meaning only code that has some readily measurable goal can be optimised in this way.

In the field of embedded systems, such extra cost is often acceptable due to long runtimes for programs and mass replication. Even in the general purpose world, additional effort may be put into optimising a final release which may be copied many times. However, a disparity between final code performance and that during development, where compile time is more critical, is undesirable in both cases; an embedded systems manufacturer may not be aware of the extent to which performance gain is possible until after the specifications of the system are set. This could render the improvement moot, as the system is already capable of performing the needed task with the unoptimised code. Research into performance prediction and potential for optimisation may counter this problem [18].

This thesis only considers execution time as an optimisation goal. While this is usually the primary concern of compiler users, other factors such as code size and power consumption can be important. The thesis would have benefitted from considering these optimisation goals too, however this was not possible due to the extra time and infrastructure it would have required.

Moreover, optimisation based on search could be said to fall into a ‘gap’ between two areas of utility – that is that runtime is either not critical, in which case long compilation time is unacceptable, or it is critical, in which case hand-coding of assembly code is likely to produce the better result. Yet, embedded code is increasingly being written in high-level languages due to the maintainability and time-to-market advantages it confers, and it is necessary to provide a good solution for this growing market.

Comparison between different techniques is made more difficult by inconsistent infrastructure within this thesis – two different benchmark suites are used: UTDSP [29] and EEMBCv2 [19]. The change of benchmark suite between chapters 6 and 7 inhibits comparison between the techniques. However, this was necessary because the UTDSP benchmark suite is small in number of benchmarks, and it is unlikely that the clustering based ‘whole picture’ technique of chapter 7 would be interesting or successful on this suite.

Another change between chapters 5 and 6, and chapter 7 is the change in compiler infrastructure between COLO Tool/SUIF and Milepost GCC due to the progression of infrastructure work over time. This makes the works more difficult to compare, and is regrettable.

Difficulty with infrastructure and compiler compatibility also led to a mismatch in the number of benchmarks considered on the two different architectures in chapter 6, where an additional five benchmarks are considered on the AMD platform over the TI. This makes any comparison between the two architectures less persuasive, however it was thought preferable to cutting the benchmark suite further on both systems.

In chapter 7, the use of a general-purpose processor differentiates this chapter from the others. The use of an embedded processor in this chapter would have been preferable to evaluate an embedded benchmark suite, however, this was not possible (see section 7.5.4).

Finally, parallelism is the greatest compiler challenge in an increasingly multi-core world. This thesis does not address the issue of parallelism at all, purely dealing with single-threaded optimisation. It is likely that the relevance of single-threaded optimisation will decrease as that of parallelism increases.

8.3 Future Work

This thesis has used some simple machine learning techniques to obtain significant improvement in compiler performance by transformation selection and reordering. We have seen that the optimisation space considered by these techniques is so vast and complicated, that it is likely that more sophisticated machine learning algorithms will be useful in improving them further.

One such example is *semi-supervised learning*, where both unsupervised learning and supervised learning are combined, giving a good coverage of the space while still having the potential to learn complex space properties. Another is the field of *active learning*, where the learning system itself can choose where there is a lack of information in the optimisation space and ask for a particular point to be sampled. There is every reason to believe that there is still gain to be made.

This thesis considered only execution time as an optimisation goal, however, as has been previously stated, code size and power consumption are important too. Simply targetting them individually could be done in exactly the same way as is done in this thesis by changing the objective function. This is less interesting. However, multi-objective learning might be considered, where execution time, code size and power consumption are balanced against each other depending on requirements, which is a difficult task not addressed in this thesis.

Finally, optimisation for multi-cores is becoming increasingly important. These more complex processors pose different challenges to the compiler, and machine learning could help in solving them. For instance, in the past, auto-parallelisation techniques have performed poorly due to missing out on opportunities for parallelisation where the dependencies in the code were not fully formally analysable, or where unnecessary dependencies forbid the transformation. Machine learning can assist by examining the whole picture – auto-parallelisation techniques from the 90's focused on loop parallelism using only local analysis, and cannot take into account larger scale issues such as data layout in memory or conflicting cache behaviour. Machine learning can be used to take into account both the local and the global picture, making decisions at both levels to benefit the program as a whole.

8.4 Conclusion

This thesis has demonstrated that machine learning is a powerful tool which can be harnessed by compiler engineers to automatically optimise high-level code. Statistical analysis of program structure, content and runtime feedback has been shown to be a better way to inform compiler optimisation than existing ad-hoc techniques.

Appendix A

Appendix

Transformations in SUIF
Aggressively scalarise constant array references
Apply default SUIF transformations
Array Delinearisation
Array Padding
Bit Packing
Bounds Comparison Substitution
Break load constant instructions
Break up large expression trees
Chain multiple array references
Common Subexpression Elimination
Common Subexpression Elimination (no pointers)
Constant Folding
Constant Propagation
Control Simplification
Copy Propagation
Dead-Code Elimination
Dismantle abs instructions
Dismantle array instructions
Dismantle composite float and integer instructions
Dismantle composite float instructions
Dismantle divceil instructions
Dismantle divfloor instructions
Dismantle divmod instructions
Dismantle empty TREE FORs
Dismantle integer abs instructions
Dismantle integer max instructions

Dismantle integer min instructions
Dismantle max instructions
Dismantle memcpy instructions
Dismantle min instructions
Dismantle multi-way branches
Dismantle non-constant FORs
Dismantle TREE BLOCKs
Dismantle TREE BLOCKs with empty symbol table
Dismantle TREE FORs
Dismantle TREE FORs with modified index variable
Dismantle TREE FORs with spilled index variables
Dismantle TREE LOOPs
Eliminate enumeration types
Eliminate struct copies
Eliminate sub-variables
Elimination of unused symbol
Elimination of unused types
Explicit array references
Extract array upper bounds
Find Fors
Fix address taken
Fix bad nodes
Fix LDC types
For Loop Normalisation
Forward Propagation
Global variable privatisation
Globalise local static variables
Guard FORs
Hoisting of loop invariants
If Hoisting
Improve array bound information
Induction Variable Detection
Kill redundant line marks
Lift call expressions
Loop flattening
Loop Tiling
Loop Unrolling
Mark constant variables
mod/ref Annotations
Move loop-invariant conditionals

Privatisation
Put in explicit loads/stores for non-local variables
Reassociation
Reduction Detection
Replace call-by-reference
Replace constant variables
Scalarisation
Scalarise constant array references
Split deep fors
Strictly fix bad nodes
Turn imperfectly nested loops into perfectly nested loops
Unstructured control flow optimisation

Table A.1: List of source-to-source transformations used in SUIF

Bibliography

[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

[2] Analog Devices, The TigerSHARC processor.

http://www.analog.com/en/embedded-processing-dsp/tigersharc/content/

tigersharcprocessorarchitecturalfeatures/fca.html

[3] S. Baluja. *Population-Based Incremental Learning: A Method for Integrating Genetic Search Based Function Optimization and Competitive Learning Source.* Technical Report: CS-94-163, Carnegie Mellon University, Pittsburgh, PA, 1994

[4] Richard Bennett and Alastair Murray and Bjoern Franke and Nigel Topham, Combining *Source-to-Source Transformations and Processor Instruction Set Extensions for the Automated Design-Space Exploration of Embedded Systems*, Proceedings of the ACM SIGPLAN/SIGBED 2007 Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES), 2007.

[5] D. Bernstein, D. Goldin, M. Golumbic, H. Krawcyk, Y. Mansour, I. Nahshon, and R. Y. Pinter. *Spill Code Minimization Techniques for Optimizing Compilers*. In Proceedings of the SIGPLAN 89 Conference on Programming Language Design and Implementa- tion, pages 258263, 1989.

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

[7] 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.

[8] F. Bodin, Z. Chamski, C. Eisenbeis, E. Rohou, and A. Seznec. GCDS: A *compiler strategy for trading code size against performance in embedded applications*. Technical Report RR-3346, INRIA, France, 1998.

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

[10] John Cavazos, Grigori Fursin, Felix Agakov, Edwin Bonilla, Michael F. P. O'Boyle, and Olivier Temam *Rapidly Selecting Good Compiler Optimizations using Performance Counters.*, Proceedings of the International Symposium on Code Generation and Optimization (CGO), 2007.

[11] John Cavazos, Christophe Dubach, Felix Agakov, Edwin Bonilla, Michael F. P. O'Boyle, Grigori Fursin, and Olivier Temam, *Automatic Performance Model Construction for the Fast Software Exploration of New Hardware Designs*, Proceedings of International Conference On Compilers, Architecture, and Synthesis for Embedded Systems (CASES), 2006

[12] John Cavazos and Michael F. P. O'Boyle., *Method-Specific Dynamic Compilation using Logistic Regression.* OOPSLA, 2006

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

[14] K. D. Cooper, D. Subramanian, and L. Torczon. *Adaptive Optimizing Compilers for the 21st Century.* In Proceedings of the 2001 LACSI Symposium, Los Alamos Computer Science Institute, October 2001.

[15] K. D. Cooper, A. Grosul, T.J. Harvey, S. Reeves, D. Subramanian, L. Torzon, and T. Waterman *Exploring the Structure of the Space of Compilation Sequences Using Randomized Search Algorithms* In Proceedings of the 2004 LACSI Symposium, Santa Fe, NM, October 2004.

[16] 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.

[17] 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, 2005.

[18] Christophe Dubach, John Cavazos, Björn Franke, Grigori Fursin, Michael O'Boyle, and Olivier Temam. *Fast Compiler Optimisation Evaluation Using Code-Feature Based Performance Prediction.* International Conference on Computing Frontiers (CF), 2007

[19] EEMBCv2 benchmark suite. http://www.eembc.org/

[20] A. Epshteyn and Maria Garzaran and Gerald Dejong and David Padua and Gang Ren and Xiaoming Li and Kamen Yotov and Keshav Pingali, *Analytic models and empirical search: A hybrid approach to code optimization*, In Proceedings of the International Workshop on Languages and Compilers for Parallel Computing (LCPC), 2005.

[21] H. Falk. *An approach for automated application of platform-dependent source code transformations.* http://ls12-www.cs.uni-dortmund.de/˜falk/, 2001.

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

[23] B. Franke and M. O'Boyle. *Array recovery and high-level transformations for DSP applications.* ACM Transactions on Embedded Computing Systems (TECS), 2(2):132–162, May 2003.

[24] Matteo Frigo and Steven G. Johnson. *The Design and Implementation of FFTW3,* Proceedings of the IEEE, vol. 93, no. 2, pp. 216–231, 2005

[25] Grigori Fursin, Cupertino Miranda, Olivier Temam, Mircea Namolaru, Elad Yom-Tov, Ayal Zaks, Bilha Mendelson, Phil Barnard, Elton Ashton, Eric Courtois, Francois Bodin, Edwin Bonilla, John Thomson, Hugh Leather, Chris Williams and Michael O'Boyle. *MILEPOST GCC: machine learning based research compiler.* In Proceedings of the GCC Developers' Summit 2008

[26] G. Fursin, M. O'Boyle, and P. Knijnenburg. *Evaluating iterative compilation.* In Proceedings of Languages and Compilers for Parallel Computers (LCPC'02), College Park, MD, USA, 2002.

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

[28] Stephen H. Gunther, Frank Binns, Douglas M. Carmean, and Jonathan C. Hall, *Managing the impact of increasing microprocessor power consumption.* In Intel Technology Journal, Q1, 2001.

[29] 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

[30] Kenneth Hoste and Lieven Eeckhout, *Comparing Benchmarks Using Key Microarchitecture-Independent Characteristics*, IISWC, pp. 83-92, 2006.

[31] Intel, The Intel Celeron Processor. "http://www.intel.com/products/processor /celeronm/index.htm"

[32] Intel, The Intel Core2Duo Processor. "http://www.intel.com/products/processor/ core2duo/index.htm"

[33] Interactive Compilation Interface for GCC. "http://gcc-ici.sourceforge.net/"

[34] Engin Ipek and Sally A. Mckee, *Efficiently exploring architectural design spaces via predictive modeling,* In Proceedings of Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2006

[35] M. Kandemir, N. Vijaykrishnan, M. J. Irwin, and H. S. Kim. *Experimental evaluation of energy behavior of iteration space tiling.* In Proceedings of the 13th

International Workshop on Languages and Compilers for Parallel Computing (LCPC'00), pages 142–157, Yorktown Heights, NY, USA, 2000.

[36] 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, 2003.

[37] P. Kulkarni, S. Hines, J. Hiser, D. Whalley J. Davidson and D. Jones. *Fast searches for effective optimization phase sequences.* In ACM PLDI, May 2004.

[38] Michail G. Lagoudakis and Michael L. Littman, *Algorithm selection using reinforcement learning*, In Proceedings of the International Conference of Machine Learning (ICML), 2000.

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

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

[41] S.Liao, S. Devadas, K. Keutzer, A. Tjiang and A. Wang *Optimization Techniques for Embedded DSP Micro-processors* In Proceedings of 33rd ACM Design Automation Conference (DAC '95), 1995

[42] C. Liem, P. Paulin, and A. Jerraya. *Address calculation for retargetable compilation and exploration of instruction-set architectures.* In Proceedings of 33rd ACM Design Automation Conference (DAC '96), pages 597–600, Las Vegas, NV, USA, 1996.

[43] 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.

[44] Eliot Moss, Paul Utgoff, John Cavazos, Doina Precup, Darko Stefanovi C, Carla Brodley, David Scheeff, *Learning to Schedule Straight-Line Code*, In Proceedings of Neural Information Processing Symposium, 1997

[45] A Joshi, A. Phansalkar, L. Eeckhout, L. Kurian John. *Measuring Benchmark Similarity Using Inherent Prgram Characteristics*, IEEE Transactions on Computers, Vol. 55, No. 6, 2006

[46] Roy Ju, Sun Chan, Tin-Fook Ngai, Chengyong Wu, Yunzhao Lu, Junchao Zhang. *Open Research Compiler (ORC) 2.0 and Tuning Performance on Itanium*, Presented at the 35th International Symposium on Microarchitecture (MICRO), 2002

[47] T. Kisuki, P.M. Knijnenburg, and M.F. O'Boyle. *Combined selection of tile sizes and unroll factors using iterative compilation.* In Proceedings of the 2000

International Conference on Parallel Architectures and Compilation Techniques (PACT'00), pages 237–248, October 2000.

[48] P. Knijnenburg, T. Kisuki, K. Gallivan, and M. F. OBoyle.

The Effect of Cache Models on Iterative Compilation for Combined Tiling and Unrolling. In Concurrency and Computation: Practice and Experience, volume 16, pages 247270, 2004.

[49] S. Long and M. OBoyle. *Adaptive Java Optimisation Using Instance-Based Learning.* In International Conference on Supercomputing, 2004.

[50] Christoforos Kozyrakis, David Judd, Joseph Gebis, Samuel Williams, David Patterson, Katherine Yelick, *Hardware/Compiler Codevelopment for an Embedded Media*, Proceedings of the IEEE pp1694-1709, 2001

[51] R.P.J. Pinkers, P.M.W. Knijnenburg, M. Haneda and H.A.G. Wijshoff. *Statistical Selection of Compiler Options.* In Proceedings of MASCOTS, pp. 494-501, 2004.

[52] Markus Püschel, José M. F. Moura, Jeremy Johnson, David Padua, Manuela Veloso, Bryan Singer, Jianxin Xiong, Franz Franchetti, Aca Gacic, Yevgen Voronenko, Kang Chen, Robert W. Johnson and Nicholas Rizzolo *SPIRAL: Code Generation for DSP Transforms* Proceedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation", Vol. 93, No. 2, pp. 232- 275, 2005

[53] S. Ray and R. Turi, *Determination of number of clusters in k-means clustering and application in colour image segmentation,* In Proceedings of the 4th International Conference on Advances in Pattern Recognition and Digital Techniques, pp. 137–143, 1999.

[54] M. Saghir, P. Chow, and C. Lee. *A comparison of traditional and VLIW DSP architecture for compiled DSP applications*. In International Workshop on Compiler and Architecture Support for Embedded Systems (CASES '98), Washington, DC, USA, 1998.

[55] W. Qin. *SimIt-ARM*. http://simit-arm.sourceforge.net, 2007

[56] W. Qin and S. Malik. *Flexible and formal modeling of microprocessors with application to retargetable simulation.* In Proceedings of Design Automation and Test in Europe (DATE), 2003

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

[58] 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.

[59] Texas Instruments, *The TI C6713 processor*. http://focus.ti.com/docs/prod/ folders/print/tms320c6713.html

[60] N.P. Topham and D. Jones, *High Speed CPU Simulation using JIT Binary Translation*, 3rd Annual Workshop on Modeling, Benchmarking and Simulation, held in conjunction with ISCA-34, San Diego CA, 2007.

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

[62] Jan-Willem van de Waerdt, *The TM3270 Media-processor*, Philips, 2005.

[63] M.J. Voss and R. Eigenmann. *High-level adaptive program optimization with ADAPT*. ACM SIGPLAN Notices, 36(7):93–102, 2001.

[64] R. Clint Whaley and Antoine Petitet and Jack J. Dongarra, *Automated empirical optimizations of software and the ATLAS project*, Parallel Computing, 2001.

[65] The XOR Problem – $http://home.agh.edu.pl/vlsi/AI/xor_t/en/main.htm$, Oct 2008

[66] 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.

[67] V. Zivojnovic, J. Velarde, C. Schlager, and H. Meyr. *DSPstone: A DSP-oriented benchmarking methodology.* In Proceedings of the International Conference on Signal Processing Applications & Technology (ICSPAT ’94), pages 715–720, Dallas, TX, USA, 1994.