ABSTRACT

VENKATAGIRI, RADHA. Predicting Compiler Optimization Performance for High-Performance Computing Applications. (Under the direction of Dr. Yan Solihin)

High performance computing application developers often spend a large amount of time in tuning their applications. Despite the advances in compilers and compiler optimization techniques, tuning efforts are still largely manual and require many trials and errors. One of the reasons for this is that many compiler optimizations do not always provide performance gain in all cases. Complicating the problem further is the fact that many compiler optimizations help performance in some cases, but hurt performance in other cases in the same application. To make it worse, it may help performance when it runs with a specific input set, but hurt the performance of the same application when it runs with a different input set.

The central idea that this work deals with is whether machine learning techniques can be used to automate compiler optimization selection. Artificial Neural Networks (ANN), and Decision Trees (DT) are modelled, trained and used to predict whether Loop Unrolling optimizations should be applied or not for loops of serial programs. Simple loop characteristics such as iteration count, nesting level, and body size, are collected and used as input to the ANN or DT. A very simple microbenchmark is used to train the ANN, and this is used to predict the benefit of loop unrolling across different NAS (Serial Version) benchmarks. We find that an ANN trained using the microbenchmark accurately predicts whether loop unrolling is beneficial in 62% of the cases. BT predicts correctly if loop unrolling is beneficial in 82% of the cases. Furthermore we find that benchmarks such as FT which perform poorly when tested with ANN trained with the microbenchmark yield accurate results in 69% of the cases when tested using an ANN trained with loops from other NAS benchmarks. Decision trees used to classify loops (as being benefitted from loop unrolling or not) from the NAS benchmarks were found to have an accuracy of 79.54%. A DT built using the microbenchmark correctly classified NAS loops 53% of the time. Although the results show promise, we believe that to accurately automate compiler optimization selection, more complex loops may need to be modeled in the microbenchmark and many other factors may need to be taken into account in characterizing each loop nest.
Predicting Compiler Optimization Performance for High-Performance Computing Applications

by

Radha Venkatagiri

A thesis submitted to the Graduate Faculty of North Carolina State University in partial satisfaction of the requirements for the Degree of Master of Science in Computer Engineering

Department of Electrical and Computer Engineering

Raleigh
2005

Approved By:

Dr. Gregory T. Byrd

Dr. Rada Y. Chirkova

Dr. Yan Solihin
Chair of Advisory Committee
Dedicated to my family and friends.
Biography

Radha Venkatagiri was born on 12th September, 1981, in Chennai, India. She graduated with a bachelor's degree in Electronics and Communication Engineering from the University of Madras in 2003. She enrolled in the graduate program in the Department of Electrical and Computer Engineering at North Carolina State University in Fall 2003. With the defense of this thesis, she is receiving the Master of Science in Computer Engineering degree.
Acknowledgements

My sincere gratitude to my family and friends for their encouragement and support.

I would like to take this opportunity to thank my advisor Dr. Yan Solihin for his invaluable guidance and patience throughout my Master's program.

I would also like to thank Dr. Rada Chirkova and Dr. Gregory T. Byrd for agreeing to be on my advising committee and evaluate this work.

Finally, I would like to thank the members of my research group, Mazen Kharbutli, Seongbeom Kim, Rithin Shetty, Fei Guo, and Brian Rogers for their advice and suggestions during the past year.
Contents

List of Figures iii
List of Tables ix

1 Introduction 1

2 Loop Unrolling Optimization 6

3 Machine Learning 10
  3.1 Artificial Neural Network 10
     3.1.1 Prediction Error Expression 12
     3.1.2 Training Termination Condition 12
  3.2 Decision Trees 13

4 Evaluation Setup 15
  4.1 Benchmarks 15
      4.1.1 TB Benchmark 15
      4.1.2 NAS Benchmark 16
  4.2 Compilation and Profiling 17
  4.3 Neural Network Training and Architecture 18
  4.4 Partitioning of Dataset 18
  4.5 Direction Prediction 19
  4.6 Decision Trees 19

5 Evaluation Results 21
  5.1 ANN trained using Trivial Benchmark 21
      5.1.1 Subsetting 21
  5.2 Intrapolation & Extrapolation 24
  5.3 NAS prediction using TB-trained ANN 27
      5.3.1 Direction Prediction 28
  5.4 NAS prediction using NAS-trained NN 30
  5.5 Decision Tree Results 37
      5.5.1 TB-trained DT 37
5.5.2 NAS-trained DT ........................................... 38

6 Conclusions ..................................................... 40

Bibliography ...................................................... 41
List of Figures

1.1 Unroll gain in loops in NAS benchmark BT with two different input sets.  
1.2 Unroll gain in loops in NAS benchmark SP with two different input sets.  
1.3 Unroll gain in loops in NAS benchmark LU with two different input sets.  

2.1 A simple Fortran loop and its corresponding x86 assembly.  
2.2 The unrolled assembly for the loop in Fig 2.1. The unroll factor in this case is 10.  

3.1 A simple mathematical model for a neuron.  
3.2 A fully connected feed-forward neural network with a 6-4-2 topology consisting of 6 input nodes, 1 hidden layer with 4 nodes and 2 output nodes.  
3.3 A basic diagram of a Decision Tree  

4.1 Loop structure from Fortran Trivial Benchmark  
4.2 Different types of partitionings (PLB & PIC) for the TB dataset  

5.1 Graph showing the error E with different types of PLB partitionings for each subset  
5.2 Graph showing the error E with different types of PIC partitionings for each subset  
5.3 Training over the entire TB benchmark with PLB partitions  
5.4 Training over the entire TB benchmark with PIC partitions  
5.5 Error graph for NAS benchmarks obtained from a NN trained over different TB PLB Partitionings.  
5.6 Error graph for NAS benchmark obtained from a NN trained over different TB PIC Partitionings.  
5.7 Unroll gain for different loops in Benchmark MG.  
5.8 Graph displaying the accuracy of NAS Benchmark predictions.
5.9 Direction Prediction accuracy shown for NAS Benchmark BT (using Inner_IC method). The training is done using loops from the other 5 benchmarks. Different validation sets are used to get different partitions. Thus, the bar corresponding to val_mg means that the training set is comprised of loops from the other 4 benchmarks i.e., EP, FT, LU and SP and the validation is done using loops from MG ........................................ 31
5.10 Direction Prediction accuracy shown for NAS Benchmark EP (using Inner_IC method) ............................................................... 32
5.11 Direction Prediction accuracy shown for NAS Benchmark FT (using Inner_IC method) ............................................................... 32
5.12 Direction Prediction accuracy shown for NAS Benchmark LU (using Inner_IC method) ............................................................... 33
5.13 Direction Prediction accuracy shown for NAS Benchmark MG (using Inner_IC method) ............................................................... 33
5.14 Direction Prediction accuracy shown for NAS Benchmark SP (using Inner_IC method) ............................................................... 34
5.15 Direction Prediction accuracy shown for NAS Benchmark BT (using Root_IC method) ............................................................... 34
5.16 Direction Prediction accuracy shown for NAS Benchmark EP (using Root_IC method) ............................................................... 35
5.17 Direction Prediction accuracy shown for NAS Benchmark FT (using Root_IC method) ............................................................... 35
5.18 Direction Prediction accuracy shown for NAS Benchmark LU (using Root_IC method) ............................................................... 35
5.19 Direction Prediction accuracy shown for NAS Benchmark MG (using Root_IC method) ............................................................... 36
5.20 Direction Prediction accuracy shown for NAS Benchmark SP (using Root_IC method) ............................................................... 37
List of Tables

4.1 Parameters of the various Fortran Trivial Benchmark loops. ............ 17

5.1 PLB Partitionings for subsetting. For example for the subset TB.1 the partitioning tr_val_test contains the following datapoints; tr: TB.1[1:200], TB.1_2[1:200], TB.1_3[1:200], TB.1_4[1:200]; val: TB.1_5[1:200] and test: TB.1_6[1:200]. ............................................. 24

5.2 Direction prediction for subsetting with different PLB partitions. .... 24

5.3 Direction prediction for subsetting with different PIC partitions. ...... 24

5.4 PLB Partitionings for entire TB Benchmark. ............................. 25

5.5 Direction prediction for entire TB benchmark with different partitions. 27

5.6 Classification accuracy for NAS Benchmarks with different IC allocation policies. The results are obtained from Decision Tree classification using the entire TB dataset for training. ............................ 38

5.7 Direction prediction accuracy for NAS Benchmarks with different IC allocation policies. The results for each individual benchmark is obtained from Decision Tree classification using other NAS benchmarks(with the same IC allocation policy) for training. ............................ 39
Chapter 1

Introduction

High performance computing application developers often spend a large amount of time in tuning their applications. Despite the advances in compilers and compiler optimization techniques, tuning efforts are still largely manual and require many trials and errors. One of the reasons for this is that many compiler optimizations do not always provide performance gain in all cases. The compiler decision in deciding whether to apply an optimization is usually based on a set of heuristics obtained from the observation of the average benchmark behavior used in the compiler development. However, high performance computing applications may not follow an average benchmark behavior. As a result, the compiler may decide to perform optimizations that hurt application performance.

Complicating the problem further is the fact that many compiler optimizations help performance in some cases, but hurt performance in other cases in the same application. For example, a loop unrolling optimization may help the performance of some loops but hurt the performance of other loops in the same application. To make it worse, it may help the performance of a loop when it runs with a specific input set, but hurt the performance of the same loop when it runs with a different input set.

Figure 1.1, 1.2 and 1.3 show the “unroll gain” for several important loops in three NAS Benchmarks with two distinct input sets: class S and class W. The unroll gain is defined as the percentage of improvement or degradation of the unrolled loop’s execution time, compared to the execution time of the same loop without unrolling. The figure shows that even for a single application, the benefit of loop unrolling varies significantly across
loops. In SP, for example, unrolling some loop can speedup its execution by 30%, but can slow down a different loop’s execution by more than 60%. Likewise, the unroll gain for some loops varies widely across different input sets. Each benchmark has loops that have a positive unroll gain on one input set, but a negative unroll gain on another input set.

![Figure 1.1: Unroll gain in loops in NAS benchmark BT with two different input sets.](image)

The variations of the unroll gain across different loops in the same application and across different input sets makes it hard for code developers to tune their application. They must envision what input set is likely to be used in order to decide what compiler optimizations that they should apply. The central question that this work deals with is whether it is possible to automate compiler optimization selection, given the characteristics of the code to optimize. Prior studies have not directly dealt with this question, although they offer approaches that could be complementary to our approach. For example, ATLAS [13] and harmony [12] have tried to automatically select the best library that improve the overall program performance. Autopilot [9, 8] and Apples [1] have tried to automatically adjust run-time parameters to improve performance. In the context of shared memory architectures, there are limited forms of self-optimizations, such as user-level page allocation [6], loop iteration distribution [5], and number of threads [2]. Prior studies have assumed that users would use run-time adaptation features in the compiler or run-time systems. However,
Figure 1.2: Unroll gain in loops in NAS benchmark SP with two different input sets.

Figure 1.3: Unroll gain in loops in NAS benchmark LU with two different input sets.
in some situations, users only want to get a well-optimized application code that does not include run-time adaptation. For such situations, one tool that would be useful is a tool that helps automate the selection of compiler optimizations.

The contribution of this work is in applying machine learning techniques, such as artificial neural networks (ANN) and Decision Tree (DT), to capture the performance behavior of compiler optimizations. For a given code section of an application, the ANN or DT predicts the performance impact of the compiler optimization. Such a tool can then be used by the compiler or application developers in deciding whether to apply the optimization on the loopnest. The benefit of this approach is that it eliminates trial-and-errors and manual-intensive tuning efforts and replaces it with a more automated framework. However, there are several challenges that need to be addressed.

For an ANN to capture a complex pattern, it needs to be trained with a lot of data points. Each data point consists of inputs that specify the code characteristic parameters, and an output that specify the effectiveness (or speedup) of a given compiler optimization. These training data points cannot be obtained using the application that we want to optimize, because then it would require a lot more runs of the application compared to a trial-and-error approach. Furthermore, a trained ANN must be general enough that it can be applied to many applications. This requirement ensures that an ANN can be trained once (or a few times) and applied many times. Finally, code characteristic parameter inputs to the ANN must not be beyond a few parameters that can be easily obtained with static code analysis [11]. Any schemes that require run-time parameter extraction, as well as complex static code analysis, are not preferable because they may add a significant complexity to the compiler and the run-time system, and would add significant performance overheads.

Our solution to the challenges outlined above is to use a very simple microbenchmark to train the ANN. This approach not only provides fast training, but also a dense parameter space coverage since the microbenchmark’s parameters can be easily changed in different runs.

To test the promise of the ANN approach, we choose loop unrolling optimization in the GNU Fortran compiler. Loop unrolling optimization is a popular optimization that is well-known and has been heavily optimized and incorporated in many compilers. In addition, we found that although loop unrolling helps performance in many cases, it also hurts performance in a significant number of cases that are hard to predict. The reason
for such unpredictability is that although it eliminates many backward branches in a loop and improves instruction level parallelism (ILP) in the loop body, it also increases register pressure. Loop unrolling performance depends on the loop body size, available ILP, and the register usage of the surrounding code. For these reasons, loop unrolling optimization is a suitable case study for our ANN approach.

The inputs to the ANN are chosen to be simple program characteristics such as the number of nesting levels, loop iteration count, and the number of instructions within the loop body. All these inputs are easy to obtain statically from high performance computing applications, without modifications to the compiler or code instrumentation.

Once trained with a microbenchmark, the ANN is used to predict the performance impact of loop unrolling for several NAS benchmarks. We found that the ANN is reasonably accurate to predict the performance impact of loop unrolling on several NAS benchmarks. Out of about 200 cases, the ANN correctly predicts that loop unrolling helps or hurts performance 62% of the time, while it incorrectly predicts 38% of the time. A simple DT built using the microbenchmark correctly classifies NAS benchmark loops 53% of the time. We believe that using a more representative microbenchmark for training, using more code characteristics as input to the ANN, as well as a better ANN organization can improve the accuracy. However, they are beyond the scope of this thesis.
Chapter 2

Loop Unrolling Optimization

To test the promise of the ANN approach, we choose loop unrolling optimization in the GNU Fortran compiler. Loop unrolling optimization is a popular optimization that is well-known and has been heavily optimized and incorporated in many compilers [3]. Loop unrolling simply replicates a loop’s body multiple times, and adjust the loop termination condition to reflect it. By making the loop body larger, loop unrolling obtains several benefits. First, it reduces loop overheads because there are less per-iteration overheads, such as the loop index computation and backward conditional branches. Second, it exposes more instruction-level parallelism (ILP) in the loop body that enables a compiler’s instruction scheduling to execute non-dependent instructions in parallel, improve memory parallelism, and reduce memory stall time.

Fig 2.1 shows an example of a simple Fortran loop and its unoptimised x86 assembly code. The unrolled version of the same loop is illustrated in Fig 2.2.

One important parameter of a loop unrolling optimization is the unroll factor. The unroll factor $u$ indicates the number of times a loop body is replicated in the transformed loop. If the original number of iterations of the loop is $n$, the unrolled loop will have $\frac{n}{u}$ iterations. However, since the number of iterations of the loop is not necessarily a multiple of the unroll factor, typically an additional code is inserted prior or after the loop to execute the remaining $n \ mod \ u$ iterations. By analyzing the resulting code from loop unrolling optimization, we found that in general the loop unroll factor is chosen to be a multiple of the number of loop iterations, unless the number of loop iterations is prime.
do 100 i = 1, 100
    a(i) = i * i
  100 continue

.L7:
    fxch %st(1)
.L6:
    fldcw -1212(%ebp)
    fistl -1216(%ebp)
    fldcw -1210(%ebp)
    fld %st(0)
    fmul %st(1), %st
    movl -1216(%ebp), %eax
    decl %edx
    fstps -1212(%ebp,%eax,4)
    fadd %st(1), %st
    js .L11
    fxch %st(1)
    jmp .L7

.L11:
    fstp %st(0)
    fstp %st(0)
    subl $8, %esp
    pushl $0
    pushl $LC1
    call s_stop

Figure 2.1: A simple Fortran loop and its corresponding x86 assembly.
L7:
fxch %st(1)
fldw -1212(%ebp)
fistl -1216(%ebp)
fld -1210(%ebp)
movl -1216(%ebp), %eax
fmul %st(1), %st
fstps -1212(%ebp,%eax,4)
fadd %st(1), %st
fldcw -1212(%ebp)
fistl -1216(%ebp)
fldcw -1210(%ebp)
fld %st(0)
movl -1216(%ebp), %eax
fmul %st(1), %st
fstps -1212(%ebp,%eax,4)
fadd %st(1), %st
fldcw -1212(%ebp)
fistl -1216(%ebp)
fldcw -1210(%ebp)
fld %st(0)
movl -1216(%ebp), %eax
fmul %st(1), %st
fstps -1212(%ebp,%eax,4)
fadd %st(1), %st
fldcw -1212(%ebp)
fistl -1216(%ebp)
fldcw -1210(%ebp)

(continued on the right)

fistl -1216(%ebp)
fldcw -1210(%ebp)
fld %st(0)
movl -1216(%ebp), %eax
fmul %st(1), %st
fstps -1212(%ebp,%eax,4)
fadd %st(1), %st
fldcw -1212(%ebp)
fistl -1216(%ebp)
fldcw -1210(%ebp)
fld %st(0)
movl -1216(%ebp), %eax
fmul %st(1), %st
fstps -1212(%ebp,%eax,4)
fadd %st(1), %st
fldcw -1212(%ebp)
fistl -1216(%ebp)
fldcw -1210(%ebp)

Figure 2.2: The unrolled assembly for the loop in Fig 2.1. The unroll factor in this case is 10.
Despite the benefits of loop unrolling, in practice it does not always improve performance. In fact, it can hurt performance due to several reasons. First, it increases register pressure since there are more live values that may need to be kept in the register after a loop is unrolled. In the extreme, such an increase in register pressure results in register spills and reloads that can significantly degrade performance. Such a degradation depends on the register usage of the surrounding code and the loop body itself. For example, if the unrolled loop is the innermost loop of a deep loop nest (it has a large number of nesting levels), there may be a few available registers for the unrolled loop. In addition, if the body of a loop is large, unrolling the loop can quickly exhaust the registers.

Overall, as a result of the interaction of loop unrolling transformation with the characteristics of the loop body, surrounding code, other compiler components, and system architecture, it is very difficult to predict whether loop unrolling is beneficial when it is applied to a given loop. Therefore, it is a good case study for testing whether an Artificial Neural Network (ANN) or Decision Tree (DT) can capture its performance behavior.

To train the ANN or DT, we need to choose input and output parameters that we supply to the ANN or DT. We choose simple program characteristics such as the number of nesting levels, loop iteration count, and the number of instructions within the loop body. All these inputs are easy to obtain statically from high performance computing applications, without modifications to the compiler or code instrumentation. The inputs are also important factors that likely determine the effectiveness of a loop unrolling optimization. For example, the loop iteration count directly affects the unroll degree. Both the loop nesting level and loop body size indirectly determine the existing register pressure in the loop before loop unrolling is applied to it. Other parameters, such as the amount of memory parallelism, data dependence graph, etc. may determine the performance of loop unrolling optimization. However, they are hard to obtain without modifying the compiler. As a result, we focus only on parameters that are easy to extract statically.
Chapter 3

Machine Learning

3.1 Artificial Neural Network

Artificial Neural Networks (ANN) are analytical techniques modeled after the (hypothesized) process of learning in the brain that are capable of learning complex patterns from a large data set. ANN has successfully been applied to fields that are characterized by the availability of large number of data points and complex relationships between the inputs that are difficult to model. Our goal in capturing the performance of loop unrolling optimization fits into this category.

ANN is a directed graph of nodes, where each node is a neuron. Neurons are the basic building blocks of ANNs. A single Neuron is illustrated in Figure 3.1. Each connection feeding into the Neuron has a specific weight attached to it. The final input to the Neuron is the summation of all inputs multiplied by the corresponding weights. An activation function (ex. standard Sigmoid Function) is then applied to this input to generate an output.

A number of nodes (Neurons) are grouped together to form a layer. An ANN has an input layer, where parameter inputs from outside the ANN are accepted, an output layer, where the output of the ANN to the outside world is generated, and zero or more hidden layers.

We use fully connected feed-forward networks such as shown in Fig. 3.2. Here each
node of a layer is connected to every other node in the next layer. Each such connection has a distinct weight attached to it. The goal of the training process is to arrive at a optimal set of weights. In Feed-forward ANNs the signals are allowed to travel only one way: from input to output. There is no feedback (loops) i.e. the output from any layer does not affect that same layer. Feed-forward ANNs are straight forward networks that associate inputs with outputs and are extensively used in pattern recognition.

To train the ANN, we use the standard backpropagation algorithm as the learning algorithm [10]. With backpropagation, learning occurs during a training phase in which each input pattern in a training set is applied to the input units and then propagated forward. The pattern of activation arriving at the output layer is then compared with the correct (target) output pattern to calculate an error signal. The error signal for each such
target output pattern is then backpropagated from the outputs to the inputs in order to appropriately adjust the weights in each layer of the network. After the ANN has been trained for a set of inputs, it can be tested on a second set of inputs to see how well it generalizes untrained data.

3.1.1 Prediction Error Expression

Many different error expressions can be used for network training. We use the mean square error function $E = \frac{1}{n} \sum_{i=1}^{n} (o_i - t_i)^2$, where $o_i$ represents the $i^{th}$ predicted output value generated by the ANN, $t_i$ represents the actual output value, and $n$ represents the number of data sets that are used in the training or prediction.

3.1.2 Training Termination Condition

We partition the training data sets into the training set and validation set. The training set consists of data points from the microbenchmark that are used for adjusting connection weights in the backpropagation learning algorithm. The validation set consists of data points from the microbenchmark that is used as a convergence criteria. The purpose of using a validation set is to determine when enough training has been performed, and to avoid overtraining, in which prediction errors may increase beyond the convergence point.

Let $E_{validation}(t)$ denote the error on the validation set after feeding the training set to the ANN $t$ times (i.e., at epoch-$t$). Let $E_{min}(t)$ denote the minimum validation error so far, i.e. it is equal to the minimum value of $E_{validation}(k)$ where $k = 1, 2, \ldots, t$. Sufficient training can be detected when the validation error only decreases very slightly (i.e., $E_{min}(t-1) - E_{validation}(t) < 0.0001$), or when the validation error starts to increase beyond $\alpha$ percent compared to the current minimum validation error, i.e.

$$\frac{E_{validation}(t) - E_{min}(t-1)}{E_{min}(t-1)} > \alpha\%$$

We also limit the maximum number of training epochs to put an upper bound on the training time.
3.2 Decision Trees

Decision Tree learning [4], is one of the most widely used methods for inductive inference and is used for approximating discrete-valued target functions. Decision Tree learning is suitable for cases instances are described by a set of attributes (e.g., height) and their values (e.g., tall) and the target function has discreet output values. Decision Trees are constructed by identifying regularities in data and this can then be used to make predictions on unseen data. A basic block diagram of a Decision Tree is shown in Figure 3.3. Decision Trees are used for the classification of instances by sorting them down the tree from the root node to some leaf node. Each node in the tree specifies a test of some attribute of the instance, with each branch descending from that node corresponding to one of the possible values for this attribute.

![Decision Tree Diagram](image)

Figure 3.3: A basic diagram of a Decision Tree

Classification of an instance is done by starting at the root node of the decision tree, testing the attribute specified by this node, then moving along the tree branch corresponding to the value of the attribute. This process is then repeated at the node on that branch and so on until a leaf node is reached.

One of the most popular algorithms for constructing a decision tree is Quinlan’s ID3 algorithm [7]. The goal of this method is to populate each leaf node by as homogeneous a sample set as possible. To accomplish this, a leaf node with an inhomogeneous sample set is selected. This leaf node is replaced by a test node that divides the inhomogeneous
sample set into minimally inhomogeneous subsets, according to an entropy calculation. More specifically, that attribute which minimises the entropy is added at the next level of the tree.
Chapter 4

Evaluation Setup

This chapter briefly describes our approach to training the Neural Network to predict loop unroll gains. A brief description of the experimental setup used is included followed by the details of the infrastructure.

4.1 Benchmarks

4.1.1 TB Benchmark

In our approach data from a set of Fortran Trivial Benchmarks(TB) are used to train the Neural Network(NN). The trivial benchmarks consist of 24 fully nested loops with different loop structures. The following four parameters are recorded for each loop.

1. Number of Nesting Levels (NL) : This represents the depth of the nested loop. Nesting levels of 1 to 4 are represented in the TB(6 loops per Nesting level).

2. Number of Statements (S) : This is the number of source code statements in the body of the innermost fortran loop.

3. Number of Multiplications(M) : The number of multiplication operations within the loop body.

4. Iteration count (IC) : This represents the number of iterations performed by the loop.
While the first three parameters are fixed for each of the 24 loops, IC is varied to get multiple points on the TB dataset (a point on the TB dataset consists of the above 4 parameters for each loop structure and the corresponding unroll gain for that loop). Figure 4.1 shows an example TB loop with NL = 2, S = 3, M = 4 and IC = num_iter.

```
do 100 j = 1, num_iter
   do 200 i = 1, num_iter
      a(i, j) = i \* j
      b(i, j) = i \* i
      c(i, j) = j \* j
   200  continue
100  continue
```

Figure 4.1: loop structure from Fortran Trivial Benchmark

Each TB loop is represented by the notation TB.NL.X, where NL is the number of nesting levels in the loop and X varies from 1 to 6, representing the 6 different loops having the same NL. The loop details of each TB loop along with the range of IC is listed in Table 5.1. A point on the TB dataset can be represented by TB.NL.X[IC], ie TB.2.4[50] represents a loop whose NL = 2 and IC = 50. It is the fourth loop with two nesting level, corresponding to S = 3 and M = 4. A range of ICs for a loop is denoted by TB.NL.X[min.IC : max.IC]. Datapoints containing all the possible ICs for a given loop is denoted by TB.NL.X[all]. For further simplicity all loops having the same NL may be denoted collectively as TB.NL. Therefore for example, TB.1 represents the loops TB.1.1, TB.1.2, TB.1.3, TB.1.4, TB.1.5 and TB.1.6.

### 4.1.2 NAS Benchmark

After training the ANN using Fortran TB Benchmarks, we use NAS Parallel Benchmarks 3.0 - Serial version(NPB3.0-SER) to test the network. Six Benchmarks BT, EP, FT, LU, MG and SP are used for experiments. The loops in the NAS benchmarks are also characterized using the same parameters as the TB Benchmark. Only perfectly nested loops are characterized for prediction as the NN is trained with perfectly nested loops from the TB Benchmark suite.
<table>
<thead>
<tr>
<th>Fortran TB Loops</th>
<th>Number of Nest levels</th>
<th>Number of Statements</th>
<th>Number of Multiplications</th>
<th>Smallest IC</th>
<th>Largest IC</th>
</tr>
</thead>
<tbody>
<tr>
<td>TB_{1,1}</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>200</td>
</tr>
<tr>
<td>TB_{1,2}</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>200</td>
</tr>
<tr>
<td>TB_{1,3}</td>
<td>1</td>
<td>3</td>
<td>3</td>
<td>1</td>
<td>200</td>
</tr>
<tr>
<td>TB_{1,4}</td>
<td>1</td>
<td>3</td>
<td>4</td>
<td>1</td>
<td>200</td>
</tr>
<tr>
<td>TB_{1,5}</td>
<td>1</td>
<td>3</td>
<td>5</td>
<td>1</td>
<td>200</td>
</tr>
<tr>
<td>TB_{1,6}</td>
<td>1</td>
<td>3</td>
<td>6</td>
<td>1</td>
<td>200</td>
</tr>
<tr>
<td>TB_{2,1}</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>200</td>
</tr>
<tr>
<td>TB_{2,2}</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>200</td>
</tr>
<tr>
<td>TB_{2,3}</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>1</td>
<td>200</td>
</tr>
<tr>
<td>TB_{2,4}</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>1</td>
<td>200</td>
</tr>
<tr>
<td>TB_{2,5}</td>
<td>2</td>
<td>3</td>
<td>5</td>
<td>1</td>
<td>200</td>
</tr>
<tr>
<td>TB_{2,6}</td>
<td>2</td>
<td>3</td>
<td>6</td>
<td>1</td>
<td>200</td>
</tr>
<tr>
<td>TB_{3,1}</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>200</td>
</tr>
<tr>
<td>TB_{3,2}</td>
<td>3</td>
<td>2</td>
<td>4</td>
<td>1</td>
<td>200</td>
</tr>
<tr>
<td>TB_{3,3}</td>
<td>3</td>
<td>3</td>
<td>6</td>
<td>1</td>
<td>200</td>
</tr>
<tr>
<td>TB_{3,4}</td>
<td>3</td>
<td>3</td>
<td>7</td>
<td>1</td>
<td>200</td>
</tr>
<tr>
<td>TB_{3,5}</td>
<td>3</td>
<td>3</td>
<td>8</td>
<td>1</td>
<td>200</td>
</tr>
<tr>
<td>TB_{3,6}</td>
<td>3</td>
<td>3</td>
<td>9</td>
<td>1</td>
<td>200</td>
</tr>
<tr>
<td>TB_{4,1}</td>
<td>4</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>50</td>
</tr>
<tr>
<td>TB_{4,2}</td>
<td>4</td>
<td>2</td>
<td>4</td>
<td>1</td>
<td>50</td>
</tr>
<tr>
<td>TB_{4,3}</td>
<td>4</td>
<td>3</td>
<td>6</td>
<td>1</td>
<td>50</td>
</tr>
<tr>
<td>TB_{4,4}</td>
<td>4</td>
<td>3</td>
<td>7</td>
<td>1</td>
<td>50</td>
</tr>
<tr>
<td>TB_{4,5}</td>
<td>4</td>
<td>3</td>
<td>8</td>
<td>1</td>
<td>50</td>
</tr>
<tr>
<td>TB_{4,6}</td>
<td>4</td>
<td>3</td>
<td>9</td>
<td>1</td>
<td>50</td>
</tr>
</tbody>
</table>

Table 4.1: Parameters of the various Fortran Trivial Benchmark loops.

### 4.2 Compilation and Profiling

All our timing and experimentation is done on a bus-based symmetric multiprocessor (SMP) with two 2 GHz Intel Xeon processors. Each processor is simultaneously multi-threaded (SMT) with two thread contexts, has a small L1 data cache, a small L1 instruction trace cache, and a unified 512KB L2 cache. Both processors share a common memory bus. The memory controller is part of the Intel 860 Chipset and the main memory is 512MB of Rambus RDRAM. The operating system on the machine is Red Hat Linux 8.0, kernel version 2.4.20, and supports SMP.

Timing for benchmarks (TB and NAS) is done by the Read Time Stamp Counter
macro `rtsc11()`. Each Benchmark is compiled using g77 with compiler option set to `-O2` for timing without loop unrolling and to `-funroll-loops -O2` to time with loop unrolling turned on. Function calls to `rtsc11()` are inserted within the code above and below the loops to time them. Initial run of the NAS benchmarks are compiled using the -pg option. Profiling information on the control flow of the application is obtained, so that timing overhead for individual loops can be minimised. Each benchmark (TB and NAS) is run 30 times with and without loop unrolling enabled, and the average execution time (in clock ticks) is calculated after removing five maximums and five minimums from the sample. The normalised `unroll gain` for each loop is calculated as:

\[
Unroll\ gain = \frac{Execution\ time\ without\ unroll - Execution\ time\ with\ unroll}{Execution\ time\ without\ unroll}
\] (4.1)

### 4.3 Neural Network Training and Architecture

Fully connected, multilayer feed forward networks with standard sigmoid hidden units are employed for training and testing. The inputs to the NN are the 4 parameters (NL, S, M, IC) collected for each loop and the output of the network is the unroll gain for each loop. The learning rate given is 0.01 and the error function used is `mean square error` which will be referred to as `E` or simply `error` from here on. The Network is initialized with random weights from 0.1...0.9. The `Strip Length` used is 5 epochs. Training is stopped as soon as the `GL5` stopping criterion is fulfilled or when training progress sinks below 0.1 per thousand or when a maximum of 5000 epochs have been trained. For each problem, 7 different network topologies are used: one-hidden-layer networks with 4, 6, 8 or 16 hidden nodes and two-hidden-layer networks with 4+4, 6+6 and 8+8 hidden nodes on the first and second hidden layer, respectively. Because random initialization of the network is employed, each Network architecture is run 10 times. The test set performance is then computed for that state of the network which has minimum validation set error during the training process.

### 4.4 Partitioning of Dataset

The dataset used for evaluation of the NN is divided into 3 sets: Training Set, Validation Set and Testing Set. From any partitioning dataset six partitions can be created
by reinterpreting the data into a different order of training, validation and test set. We have explored two basic types of partitioning for the TB dataset in our experiments, *Partitioning across Loop Body (PLB)* and *Partitioning across Iteration Count (PIC)*. Figure 4.2 illustrates both types of partitioning for the datasets of TB_2.

![Diagram](image)

Figure 4.2: Different types of partitionings (PLB & PIC) for the TB dataset

### 4.5 Direction Prediction

While minimizing the error function (for value of unroll gain) for each loop is important, the final goal of this work is to predict whether the unroll gains for loops is positive or negative in High Performance Applications. We want to use Neural Networks to predict whether the loop unrolling optimization will speedup or slowdown the loop. Towards this goal we check the accuracy of our ANN in predicting the direction of the gain. A negative unroll gain means predicting that the application will be slowed down by the optimization and a positive unroll gain means that the ANN predicts that loop unrolling will speedup the loop.

### 4.6 Decision Trees

Classification using Decision Trees are done to establish proof of concept and support the results obtained from Machine Learning using Neural Networks, and are not the
central focus of this work. We use the Decision tree algorithm in WEKA [14] for Classification. Weka is an open source tool and has a collection of machine learning algorithms for data mining tasks. In this work we use the J48 tree model in WEKA-3-4-4 which is a clone of the C4.5 decision tree algorithm. The various Fortran loops are classified according to whether or not they benefit from loop unrolling (Binary classification). If the unroll gain of a loop is positive or zero, it is then classified as, say, class X and if the unroll gain is negative, it is classified as, say, Class Y. The performance of the Classifier is determined by the Accuracy, which is measured by counting the number of correctly predicted samples from an unseen test dataset. The simplest case is one where the dataset is split into mutually exclusive training and testing set. Here, the training set is used to build the classifier model and the unseen test dataset is used to evaluate the generated model. Cross Validation is used to determine the accuracy of a training set when no test dataset has been specified. In this method the dataset is divided into n equal sized folds after random reordering. One fold is used for testing and the remaining n-1 folds are used to train the classifier. The estimate of accuracy is obtained by collecting the test results and averaging over all folds. Although there is ample support for data preprocessing (by resampling the dataset, adding attributes, removing examples and so on) in WEKA, we do not use it for our experiments. This is because no preprocessing is done in the dataset used to train the Neural Networks and we want to compare the results obtained from the two machine learning algorithms.
Chapter 5

Evaluation Results

5.1 ANN trained using Trivial Benchmark

In this section we describe the experimentation results obtained from training and testing the Neural Network using different TB Benchmark Partitionings. The purpose of this experiment is to find out if the ANN can capture the loop unrolling effectiveness behaviour on simple loop structures in the TB benchmark.

5.1.1 Subsetting

In this approach we train and test dedicated ANN for TB loops having the same Nesting level. Each nesting level is treated as a subset, with an independent ANN trained and tested using loops having the same nesting level. Thus, we have a total of four individual ANNs each for TB.1, TB.2, TB.3 and TB.4. The purpose of this experiment is to see how well the Neural Network captures the unroll behaviour for each Nesting Level. Both PLB and PIC partitionings are done. In PLB, out of the 6 loops per Nesting Level, 4 loops are used for training(tr), 1 for validation(val) and 1 for testing(test). This results in six possible partitions. Table 5.1 shows the different PLB partitions for each subset. In the PIC partitioning some datapoints from all the 6 loops of the subset are used for training, testing and validation. Approximately 60% of the datasets for each loop is used for training, 20% for validation and 20% for testing. As the datasets are ar-
ranged in ascending order of IC for each loop, the six different partitionings are created by reinterpreting the data into a different order of training, validation and test set. Thus a partitioning of tr\_val\_test means that the first 60% points (for each loop in the subset) are used for training the ANN, the next 20% used for validation and the last 20% for testing the ANN. For TB.1, TB.2 and TB.3, 120 datapoints (from each loop in the subset) are used for training, 40 for validation and 40 for testing. For TB.4, 30 datapoints (from each loop in the subset) are used for training, 10 for validation and 10 for testing. For example, a partitioning of val\_tr\_test for TB.3 uses the following datasets; validation: TB.3.1[1:40], TB.3.2[1:40], TB.3.3[1:40], TB.3.4[1:40], TB.3.5[1:40], TB.3.6[1:40]; training: TB.3.1[41:160], TB.3.2[41:160], TB.3.3[41:160], TB.3.4[41:160], TB.3.5[41:160], TB.3.6[41:160]; and testing: TB.3.1[161:200], TB.3.2[161:200], TB.3.3[161:200], TB.3.4[161:200], TB.3.5[161:200].

Figure 5.1 shows the Average mean square error E of the testing set per PLB partition per subset. The figure shows that, the minimum error is achieved across all subsets by using partitions which use the first four loops in the subset for training. This is because the first four loop datasets in each subset show more variation in the loop parameters, ranging from the lowest to the highest for both the Body size and Number of Multiplications, to capture the behaviour of the TB subset. Thus, the NN is well trained and achieves minimum E with a new test set.

As can be seen from the Figure 5.2, Maximum error E is obtained by PIC partitionings that do not use smaller matrix sizes for training. This is expected because unroll gains vary widely for very small matrix sizes. Hence, Networks which are not exposed to these points may not be very robust. Also, since these points are now used for testing, the average testing error goes up. PIC partitioning performs better on average than partitioning based on loop body (PLB), as some points from all the loops in the Subset are used for training, hence achieving better representation. The direction prediction accuracy for subsetting with different PLB partitions for each subset is given in Table 5.2, and that with different PLB partitions for each subset is given in Table 5.3. As can be seen from the table the prediction accuracy is very high in both PLB and PIC, almost 100% in most cases. This suggests that the ANN captures the loop unrolling performance behaviour of each subset very well.
Figure 5.1: Graph showing the error E with different types of PLB partitionings for each subset
<table>
<thead>
<tr>
<th>Partitioning</th>
<th>Training</th>
<th>Validation</th>
<th>Testing</th>
</tr>
</thead>
</table>

Table 5.1: PLB Partitionings for subsetting. For example, for the subset TB_1 the partitioning tr_val_test contains the following datapoints; tr: TB_1.1[1:200], TB_1.2[1:200], TB_1.3[1:200], TB_1.4[1:200]; val: TB_1.5[1:200] and test: TB_1.6[1:200].

<table>
<thead>
<tr>
<th>Subset</th>
<th>val_tr_test</th>
<th>val_test_tr</th>
<th>test_val_tr</th>
<th>test_tr_val</th>
<th>tr_val_test</th>
<th>tr_test_val</th>
</tr>
</thead>
<tbody>
<tr>
<td>TB_1</td>
<td>98</td>
<td>99</td>
<td>98</td>
<td>98</td>
<td>99</td>
<td>99</td>
</tr>
<tr>
<td>TB_2</td>
<td>98.5</td>
<td>99</td>
<td>99.5</td>
<td>100</td>
<td>99</td>
<td>99</td>
</tr>
<tr>
<td>TB_3</td>
<td>98</td>
<td>99</td>
<td>99.5</td>
<td>100</td>
<td>99</td>
<td>98.5</td>
</tr>
<tr>
<td>TB_4</td>
<td>98</td>
<td>99</td>
<td>98</td>
<td>96</td>
<td>98</td>
<td>98</td>
</tr>
</tbody>
</table>

Table 5.2: Direction prediction for subsetting with different PLB partitions.

5.2 Intrapolation & Extrapolation

In this section we present results for ANNs trained using the entire TB Benchmark suite and not just a subset of it. Again the entire TB set is partitioned into Training set, validation set and testing set. The PLB partitioning is illustrated in Table 5.4. Partition ordering val_tr_test, test_tr_val, test_val_tr and tr_val_test can be considered as extrapolation over different nesting levels and partition ordering val_test_tr and tr_test_val as intrapolations. Figure 5.3 shows that, on an average extrapolation of nesting levels yields lower error

<table>
<thead>
<tr>
<th>Subset</th>
<th>val_tr_test</th>
<th>val_test_tr</th>
<th>test_val_tr</th>
<th>test_tr_val</th>
<th>tr_val_test</th>
<th>tr_test_val</th>
</tr>
</thead>
<tbody>
<tr>
<td>TB_1</td>
<td>100</td>
<td>100</td>
<td>92.08</td>
<td>92.08</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>TB_2</td>
<td>100</td>
<td>100</td>
<td>95.83</td>
<td>95.83</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>TB_3</td>
<td>100</td>
<td>100</td>
<td>94.17</td>
<td>94.17</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>TB_4</td>
<td>100</td>
<td>100</td>
<td>88.33</td>
<td>88.33</td>
<td>100</td>
<td>100</td>
</tr>
</tbody>
</table>

Table 5.3: Direction prediction for subsetting with different PIC partitions.
Figure 5.2: Graph showing the error E with different types of PIC partitionings for each subset

than interpolation. The val_test_train partitioning yields the highest error. The very low and very high nesting levels usually have high unroll gain variations and it may be that the network learns for irregularity rather than the common case.

<table>
<thead>
<tr>
<th>Partitioning</th>
<th>Training</th>
<th>Validation</th>
<th>Testing</th>
</tr>
</thead>
<tbody>
<tr>
<td>tr_val_test</td>
<td>TB_1[all] ,TB_2[all]</td>
<td>TB_3[all]</td>
<td>TB_4[all]</td>
</tr>
<tr>
<td>tr_test_val</td>
<td>TB_1[all] ,TB_2[all]</td>
<td>TB_4[all]</td>
<td>TB_3[all]</td>
</tr>
<tr>
<td>val_tr_test</td>
<td>TB_2[all] ,TB_3[all]</td>
<td>TB_1[all]</td>
<td>TB_4[all]</td>
</tr>
<tr>
<td>val_test_tr</td>
<td>TB_3[all] ,TB_4[all]</td>
<td>TB_1[all]</td>
<td>TB_2[all]</td>
</tr>
<tr>
<td>test_val_tr</td>
<td>TB_3[all] ,TB_4[all]</td>
<td>TB_2[all]</td>
<td>TB_1[all]</td>
</tr>
<tr>
<td>test_tr_val</td>
<td>TB_2[all] ,TB_3[all]</td>
<td>TB_4[all]</td>
<td>TB_3[all]</td>
</tr>
</tbody>
</table>

Table 5.4: PLB Partitionings for entire TB Benchmark.

The PIC partitioning follows the same rules as in the case of subsetting except that the ANN now trains for the whole TB benchmark instead of just one subset. Figure 5.4 plots the mean square error of the testing set for different PIC partitionings. Because the irregularities are higher in smaller matrix sizes, the partition order which uses the lower matrix sizes to test, has predictably higher error E. Once again the Direction prediction accuracy for both types of Partitions, as illustrated in Table 5.5, show very high accuracy in prediction.
Error E for TB with PLB Partitions

Average mean square error over testing set

Figure 5.3: Training over the entire TB benchmark with PLB partitions

Error E for TB with PIC Partitions

Average mean square error over testing set

Figure 5.4: Training over the entire TB benchmark with PIC partitions
### 5.3 NAS prediction using TB-trained ANN

This section describes experimentation using NAS NPB3.0-SER Benchmarks. A fully connected ANN is trained using all loops from the TB benchmark suit and is used to predict the unroll gain for loops in the NAS Benchmark suite. Once again partitionings are done for TB based on Loop body (PLB) and Iteration count (PIC). But, here the TB is divided only into training and validation set, as loops from NAS benchmarks are used to test the NN. The loops in the Trivial Benchmark have the same number of iterations at each nesting depth for any given loop. Thus, the IC for these loops is characterised by just one number. But, this may not be the case for NAS loops. The Iteration Count (IC) for the loops from the NAS benchmarks are allocated using two methods:

a) **Root IC**: In this method the number of iterations from each nesting level of the loop is multiplied to yield a *Total Iteration Count (TIC)*. If N is the nesting depth of the loop, then the Nth root of the TIC yields the Iteration count associated with that particular loop.

b) **Inner IC**: This is a relatively simple method where the number of iterations of the innermost loop is given as the Iteration Count (IC).

For example, for a loop having two nesting levels with the outer loop having 9 iterations and the inner loop having 4 iterations, the IC using Root IC is 6 and using Inner IC is 4.

Figure 5.5 shows the error E for the six NAS benchmarks BT, EP, FT, LU, MG and SP for different PLB partitionings of the TB suite and using the two IC allocation methods described above. For example *val\_i\_root* means that the NN was trained using the TB\_i[all] as validation dataset and the rest of the TB dataset (TB\_2[all], TB\_3[all] and TB\_4[all] ) as the training set; and the IC for the loops in the NAS Benchmarks (the testing set) were allocated using the *Root IC* method. It can be seen from the graph, that the method of allocating IC to the NAS loops does not make a significant difference to the outcome of the test. We attribute this to the fact that many of the loops in the NAS benchmarks are singly nested loops and even loops with higher nesting levels do not vary much in the number of iterations per nesting level. Hence, there is no significant difference
between the two methods. As can be seen from the graph, there is not much difference using the various methods, since we are using all points in training the network.

![Error E for NAS using PLB partitions over TB](image)

**Figure 5.5:** Error graph for NAS benchmarks obtained from a NN trained over different TB PLB Partitionings.

Figure 5.6 shows the error from testing the NAS benchmarks using PIC partitioning. Once again, the TB benchmark is partitioned only into training and validation set. 75% of the samples are used for training and 25% for validation. Once again the IC allocation scheme does not make a difference in the prediction of the unroll gain for the NAS benchmark loops. With both partitionings MG achieves the highest mean square error E over all its loops. This can be understood by looking at the *Unroll Gain* of the loops in MG as illustrated in Fig 5.7. As can be observed many of the loops of MG have high Unroll Gains. Two loops in particular achieve around -350% unroll gain. These high numbers significantly affect the mean square error, raising it considerably.

### 5.3.1 Direction Prediction

On average, we find that of the 200-odd NAS Benchmark loops that we test, we are able to accurately predict the direction of around 62% of loops. Figure 5.8 gives us the accuracy of direction prediction for loops in the different NAS Benchmarks separately. BT has the highest prediction accuracy with around 80% of the loops being predicted accurately. SP comes second with a loop prediction accuracy of 65%. The Lowest is FT with only
Figure 5.6: Error graph for NAS benchmark obtained from a NN trained over different TB PIC Partitionings.

Figure 5.7: Unroll gain for different loops in Benchmark MG.
33.33% of the loops being correctly classified. In FT, we see that a large percentage of the loops exhibit negative unroll gains. In the TB dataset however a large majority of points have positive unroll gains and hence the Neural Network learns more for the general case. Also, although some NAS datapoints may closely match those found in the TB dataset (similar parameters used to characterize the loops), their unroll gains are vastly different. From this we can conclude that the simple parameters and loop structures (from TB) that we use may not be able to accurately capture the behaviour of some NAS loops.

Direction Prediction Accuracy for NAS Benchmarks

![Graph showing direction prediction accuracy for NAS Benchmarks]

Figure 5.8: Graph displaying the accuracy of NAS Benchmark predictions.

5.4 NAS prediction using NAS-trained NN

This section describes the results obtained for Direction Prediction of different NAS benchmarks using Neural Networks trained with other NAS Benchmarks. As always, we have a training set, a validation set and a test set. Out of the six different NAS Benchmarks, loops from one benchmark are used for testing, loops from another one are used for validation, and loops from the rest four benchmarks are used to train the ANN. In order to better analyze the results, we define a Statistical Confidence Interval for a Normal Distribution with unknown variance, which is calculated using the average error of the training set. We choose a confidence level of 97.5%. A loop unroll gain is classified
as unconfident if its upper and lower confidence limits lie on either side of zero. Fig 5.9, Fig 5.10, Fig 5.11, Fig 5.12, Fig 5.13 and Fig 5.14 show the accuracy of direction prediction for benchmarks BT, EP, FT, LU, MG and SP respectively with Inner JC. Fig 5.15, Fig 5.16, Fig 5.17, Fig 5.18, Fig 5.19 and Fig 5.20 show the accuracy of direction prediction for benchmarks BT, EP, FT, LU, MG and SP respectively with Root JC. Once again there is no clear winner between the two IC allocation policies. As can be seen from the graphs, the number of incorrect predictions for most benchmarks is high. For example, Fig 5.15 shows that BT has an correct prediction accuracy of only around 45%. To understand this, the Confidence Interval is applied to these loops. We find that in most cases the majority of the incorrectly classified loops are also unconfident. In the case of BT (Fig 5.15) almost all the incorrectly classified loops are unconfident. Surprisingly, as shown by Fig 5.11 and Fig 5.17, FT has a high degree of accuracy. Around 69% of loops in FT are correctly predicted. This leads us to conclude that a NN trained using NAS can more accurately capture the behaviour of those loops that are not adequately represented by the TB dataset. This also supports our argument for why FT achieves low prediction accuracy in the TB trained NN.

![Direction Prediction Accuracy for BT(Inner)](image)

Figure 5.9: Direction Prediction accuracy shown for NAS Benchmark BT (using Inner JC method). The training is done using loops from the other 5 benchmarks. Different validation sets are used to get different partitions. Thus, the bar corresponding to val_mg means that the training set is comprised of loops from the other 4 benchmarks i.e., EP, FT, LU and SP and the validation set is done using loops from MG.
Figure 5.10: Direction Prediction accuracy shown for NAS Benchmark EP(using Inner_IC method).

Figure 5.11: Direction Prediction accuracy shown for NAS Benchmark FT(using Inner_IC method).
Figure 5.12: Direction Prediction accuracy shown for NAS Benchmark LU(using Inner IC method).

Figure 5.13: Direction Prediction accuracy shown for NAS Benchmark MG(using Inner IC method).
Figure 5.14: Direction Prediction accuracy shown for NAS Benchmark SP(using Inner IC method).

Figure 5.15: Direction Prediction accuracy shown for NAS Benchmark BT(using Root IC method).
Figure 5.16: Direction Prediction accuracy shown for NAS Benchmark EP(using Root_IC method).

Figure 5.17: Direction Prediction accuracy shown for NAS Benchmark FT(using Root_IC method).
Figure 5.18: Direction Prediction accuracy shown for NAS Benchmark LU (using Root_IC method).

Figure 5.19: Direction Prediction accuracy shown for NAS Benchmark MG (using Root_IC method).
Direction Prediction Accuracy for SP(Root)

Figure 5.20: Direction Prediction accuracy shown for NAS Benchmark SP (using Root.IC method).

5.5 Decision Tree Results

The Decision tree (DT) Classifier was used on both the Trivial Benchmark as well as the NAS Benchmark loops to classify them according to whether they benefit (achieve speedup) from loop unrolling or not.

5.5.1 TB-trained DT

The classification on the Trivial Benchmark dataset using a ten-fold cross-validation yielded an accuracy of 90.3%. The 6 NAS Benchmarks were used to test the classifier model built by using the TB dataset. The results are shown in Table 5.6. The trend that is seen is similar to that seen using Neural Networks with BT yielding the best results (63.16% accuracy) and FT the worst (16.67% accuracy). The DT Classifier performs slightly worse than the Neural Network as only 53% of the (from around 200) NAS loops are accurately predicted as compared to the 62% obtained from NN.
<table>
<thead>
<tr>
<th>NAS Benchmarks</th>
<th>Percentage of Accurately Predicted Loops</th>
</tr>
</thead>
<tbody>
<tr>
<td>BT_Inner</td>
<td>63.16</td>
</tr>
<tr>
<td>BT_Root</td>
<td>62.5</td>
</tr>
<tr>
<td>EP_Inner</td>
<td>50</td>
</tr>
<tr>
<td>EP_Root</td>
<td>50</td>
</tr>
<tr>
<td>FT_Inner</td>
<td>16.67</td>
</tr>
<tr>
<td>FT_Root</td>
<td>15.38</td>
</tr>
<tr>
<td>LU_Inner</td>
<td>48.84</td>
</tr>
<tr>
<td>LU_Root</td>
<td>46.94</td>
</tr>
<tr>
<td>MG_Inner</td>
<td>53.33</td>
</tr>
<tr>
<td>MG_Root</td>
<td>53.33</td>
</tr>
<tr>
<td>SP_Inner</td>
<td>60</td>
</tr>
<tr>
<td>SP_Root</td>
<td>58.18</td>
</tr>
<tr>
<td>All Benchmarks(Inner)</td>
<td>53.7</td>
</tr>
<tr>
<td>All Benchmarks(Root)</td>
<td>52.3</td>
</tr>
</tbody>
</table>

Table 5.6: Classification accuracy for NAS Benchmarks with different IC allocation policies. The results are obtained from Decision Tree classification using the entire TB dataset for training.

5.5.2 NAS-trained DT

The individual NAS Benchmarks were also tested on classification models built by using other NAS Benchmarks. For example prediction accuracy for BT is obtained by testing it on a classifier model built using the loops of EP, FT, LU, MG and SP, with all the benchmarks using the same IC allocation policy. The results are tabulated in Table 5.7. On average the prediction accuracy yielded using DT is more or less similar to that using ANN (having similar training and testing data) in the case of LU, EP and MG. BT and SP perform much better (72.5% and 66% accuracy respectively) when classified using decision trees. This is attributed to the fact that in DT, all the loops in the other five benchmarks are used in training. In case of ANN, only loops from 4 benchmarks are used in training as the loops from one benchmark is used for validation. Thus, the information from some of the loops is lost in the case of ANN. FT has a lower accuracy when tested using DT, as compared to the ANN model.

In another experiment loops from all the 6 NAS benchmarks were used to build the DT. This is done to see how well the classification pattern is captured by the DT. The classification on the entire NAS dataset (loops from all 6 benchmarks) using a tenfold cross-validation yielded an accuracy of 79.54% for IC_Root policy and 70.37% using
<table>
<thead>
<tr>
<th>NAS Benchmarks</th>
<th>Percentage of Accurately Predicted Loops</th>
</tr>
</thead>
<tbody>
<tr>
<td>BT_Inner</td>
<td>68.42</td>
</tr>
<tr>
<td>BT_Root</td>
<td>72.5</td>
</tr>
<tr>
<td>EP_Inner</td>
<td>25</td>
</tr>
<tr>
<td>EP_Root</td>
<td>50</td>
</tr>
<tr>
<td>FT_Inner</td>
<td>33.33</td>
</tr>
<tr>
<td>FT_Root</td>
<td>30.77</td>
</tr>
<tr>
<td>LU_Inner</td>
<td>53.49</td>
</tr>
<tr>
<td>LU_Root</td>
<td>57.14</td>
</tr>
<tr>
<td>MG_Inner</td>
<td>26.67</td>
</tr>
<tr>
<td>MG_Root</td>
<td>40</td>
</tr>
<tr>
<td>SP_Inner</td>
<td>66</td>
</tr>
<tr>
<td>SP_Root</td>
<td>65.45</td>
</tr>
</tbody>
</table>

Table 5.7: Direction prediction accuracy for NAS Benchmarks with different IC allocation policies. The results for each individual benchmark is obtained from Decision Tree classification using other NAS benchmarks(with the same IC allocation policy) for training.

IC_Inner policy.
Chapter 6

Conclusions

The contribution of this work is in applying machine learning techniques, such as Artificial Neural Networks (ANN) and Decision Tree (DT), to capture the performance behaviour of Loop Unrolling Optimization. The experimental results show that Artificial Neural Network (ANN) and Decision Tree (DT) can be used to some degree of accuracy in predicting the effectiveness of loop unrolling.

Trivial Benchmarks were used to train ANN and these were then used to predict the benefit of loop unrolling across different NAS (Serial Version) benchmarks. We find that such an ANN trained using the Trivial Benchmarks accurately predicts whether loop unrolling is beneficial in 62% of around 200 cases. BT predicts correctly if loop unrolling is beneficial in 82% of the cases. Furthermore we find that Benchmarks such as FT which perform poorly when tested with ANN trained with the microbenchmark yield accurate results in 69% of the cases when tested using an ANN trained with loops from other NAS benchmarks. The experiments using DT also reflect the trend seen in the ANN approach. A DT built using the Trivial Benchmark correctly classified NAS loops (as being benefitted from loop unrolling or not) 53% of the time. A Decision tree used to classify loops from the NAS benchmarks using cross-validation was found to have an accuracy of 79.54%.

Although the results show promise, we believe that using a more representative microbenchmark for training, using more code characteristics as input to the ANN, as well as a better ANN (and DT) organization, can improve the accuracy.
Bibliography


