Learning Instruction Scheduling Heuristics from Optimal Data

by

Tyrel Russell

A thesis
presented to the University of Waterloo
in fulfilment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2006

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

The development of modern pipelined and multiple functional unit processors has increased the available instruction level parallelism. In order to fully utilize these resources, compiler writers spend large amounts of time developing complex scheduling heuristics for each new architecture. In order to reduce the time spent on this process, automated machine learning techniques have been proposed to generate scheduling heuristics. We present two case studies using these techniques to generate instruction scheduling heuristics for basic blocks and super blocks. A basic block is a block of code with a single flow of control and a super block is a collection of basic blocks with a single entry point but multiple exit points. We improve previous techniques for automated generation of basic block scheduling heuristics by increasing the quality of the training data and increasing the number of features considered, including several novel features that have useful effects on scheduling instructions. Our case study into super block scheduling heuristics is a novel contribution as previous approaches were only applied to basic blocks. We show through experimentation that we can produce efficient heuristics that perform better than current heuristic methods for basic block and super block scheduling. We show that we can reduce the number of non-optimally scheduled blocks by up to 55% for basic blocks and 38% for super blocks. We also show that we can produce better schedules 7.8 times more often than the next best heuristic for basic blocks and 4.4 times more often for super blocks.

Acknowledgments

For the past two years, I have been working at the University of Waterloo and I have been helped by innumerable people in a variety of ways. I would like to acknowledge some of these people for their specific contributions to the thesis. I would like to thank my supervisor, Peter van Beek, for his supervision and advice on this thesis. I would also like to thank my two readers, Robin Cohen and Ali Ghodsi, for their comments. I would like to acknowledge Abid Malik for his help in understanding the IBM block output and for the solver that he and Peter developed to solve the blocks optimally. I would like to thank Mike Chase for his advice on the project as a whole and, specifically, for helping with understanding the different architectural constraints within our model. For helpful advice and occasionally sidetracking me into playing a game or two of chess, I would like to thank Matt Enss, Kevin Regan and Fred Kroon. I would like to thank Martin Talbot, Laurent Charlin and Mark Belcarz for the encouragement they provided and for occasionally reminding me that I was stating the obvious. I would also like to thank Rado Radoulov, Greg Hines, John Whissel and Reid Kerr for the advice they have given on many occasions. Last but not least, I would like to thank my family for their constant support.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The development of processors with pipelines and multiple functional units has increased the demands on compiler writers to write complex instruction scheduling algorithms. These algorithms are required to ensure that the most efficient use of resources, i.e. the functional units and pipelines of the processor, is made due to the increased complexity of processor architectures. In this thesis, the specific problem of automatically creating instruction scheduling heuristics will be addressed. Instruction scheduling is the problem of scheduling the assembly instructions output from the code generator to increase the efficiency of the final code.

The instruction scheduling problem is mainly solved heuristically since finding an optimal solution requires significant computational resources and, in general, the problem of optimally scheduling instructions is known to be NP-Complete [21]. For many years, different heuristics have been developed to schedule instructions for various architectures. The majority of these heuristics have been list scheduling heuristics (see Section 2.1) though other scheduling techniques have been used. Developing good techniques for instruction scheduling is costly as it can require many months to develop the heuristics needed to create a good instruction scheduling heuristic for new processors. For example, the heuristic developed for the IBM XL family of compilers "evolved from *many years* of extensive empirical testing at IBM" [22, p. 112, emphasis added]. Moss et al. [8, 43, 38] proposed using machine learning to automate the task of generating a heuristic for scheduling. Our work is an extension and expansion of this methodology. The task of testing and revising the set of rules needed to find a solution is perfectly suited for a machine learning approach when good data exists to learn good rules.

While the main goal of the research is to develop a method for automated heuristic design, it is important to create heuristics that are also fast, efficient,

accurate and understandable. A deficiency with one of the previous approaches [14] is the overall complexity of the algorithm and the cost associated with executing the scheduler. Avoiding this type of costly heuristics is one of the aims of this research. There are two factors in developing a simple and inexpensive heuristic: the number of features involved in making a heuristic decision and the complexity of calculating each of the features. A feature is defined to be "a quantity describing an instance [28]." For example, if we are classifying weather, a feature could be the temperature. Many features require $O(n^2)$ time to calculate where n is the number of instructions in the block of code, and if many of these of features are used the accuracy may be improved at the expense of the execution time. If too many features are used the execution time may be too long for practical use.

We study two different types of heuristics, one for basic blocks and one for super blocks. We research and categorize a large set of features found within the literature for both basic block and super block scheduling heuristics. In addition to these features, we synthesize new features and develop several novel features that are found to be effective in both basic block and super block scheduling. Using standard machine learning techniques, we identify both irrelevant and useful features. We generate two heuristics using a decision tree learning algorithm and those heuristics have increases in terms of both optimality and worst case behaviour over previous heuristics found in the literature. Most importantly, we demonstrate that it is possible to automatically generate good heuristics for instruction scheduling of basic blocks and super blocks.

This thesis is broken into 4 main chapters. Chapter 2 covers the background material necessary to understand the remainder of the thesis. Chapter 3 covers the related work including previous heuristics and previous automated methodologies. Chapter 4 outlines the basic block heuristic developed by the machine learning techniques. Chapter 5 outlines the super block heuristic developed by the same techniques.

Chapter 2

Background

In this chapter, we describe the background material necessary to explain the contributions of this thesis. The major areas covered are the basic block and super block instruction scheduling problems, and machine learning, with the focus of the latter being on supervised learning techniques.

2.1 Instruction Scheduling

The instruction scheduling problem can be characterized as a set of instructions and a set of functional units [44, 18, 10]. A functional unit can be defined as a unit within a processor assigned to complete as specific task. For example, most processors have a floating point unit that processes floating point instructions. The instructions must be scheduled on the functional units so that all constraints on the problem are satisfied. Two major constraints are issue width and precedence constraints. The issue width of a processor limits the number of instructions that can begin execution in any clock cycle. Precedence constraints ensure that certain operations are executed in a specific order. Modern processors typically have several different functional units that can process one or more different types of instructions. On top of this machine level parallelism, the addition of pipelining allows a new non-conflicting operation to execute in every clock cycle on a single functional unit.

There are two major architecture types, superscalar and VLIW. Both allow for instruction level parallelism and pipelining. The major difference between superscalar and VLIW is that superscalar does not parallelize instructions until runtime while VLIW combines the instructions during compilation and issues them in a

block. Either type of architecture benefits from reordering the instructions to improve the overall schedule cost.

With these advanced architectures, to fully utilize the resources available the instructions must be scheduled after code generation [15]. The focus of instruction scheduling research has been to find good heuristics as finding optimal solutions remains costly. Hennessy and Gross [21] showed that instruction scheduling is NP-Complete for realistic architectures.

For this work, we make the simplifying assumption that all functional units are fully pipelined. A functional unit is fully pipelined if a new instruction can begin execution on that functional unit in every clock cycle.

Recall that precedence constraints ensure that instructions are executed in a certain order to maintain program correctness. These constraints are modelled as a directed acyclic graph (DAG) where the nodes are the instructions and the edges represent the precedence constraints. The labels of the edges represent the latencies of the constraints. The latency of a constraint represents the number of cycles that must elapse before scheduling the succeeding instruction. Each DAG has an associated order as generated by the code generator and is equivalent to the id assigned each node. Once a DAG has been constructed from the precedence relations, it is possible to determine the latest start time and earliest start time of each instruction. These properties are derived from the graph by tracing the paths through the graph and determining the maximum distances from the root and leaf nodes of the graph. The path that maximizes the distance between any two nodes is called the critical path. Table 2.1 shows notation for properties of the DAG or nodes within the DAG. To illustrate the concept of a DAG, a simple DAG with six instructions can be seen in Figure 2.1. Figure 2.1 also shows the optimal schedule, i.e. the schedule with the shortest length, for that DAG assuming a single issue processor–a processor where a single instruction can be issued at each clock cycle.

Example 2.1 *Consider the basic block in Figure 2.1. For succinctness, only node C of the graph will be considered. Since the only successor of node C is the sink node, $succ(C) = 1$ and $desc(C) = 1$. We can calculate the critical path length distances to both the root node and the sink node and we determine them to be $cp(A, C) = 1$ and $cp(A, F) = 1$, respectively. The latency between C and the only successor F is 1 and execution time of C is also 1. The path lengths to the respective nodes are also 1. The earliest start time using the critical path distance to the root is 1 and, using the fact that $cp(A, F) = 5$), the latest start time for C is $lst_F(C) = 4$. Therefore, the slack is calculated to be the difference between the two or $slack(C) = lst_F(C) - est(C) = 4 - 1 = 3$.*

Table 2.1: Notation for the Instruction Scheduling Problem.

$succ(i)$	The number of immediate successors of instruction i
$desc(i)$	The number of descendants of instruction i
$cp(i,j)$	The critical path distance between instruction i and instruction j
$pl(i,j)$	The longest path length between instruction i and instruction j
$lat(i,j)$	The latency between instruction i and instruction j
$lst_b(i)$	The latest start time of an instruction j with reference to a branch node, typically the sink (or leaf) node
$est(i)$	The earliest start time of an instruction j
$etime(i)$	The execution time of an instruction i
$slack(i)$	The difference between the latest start time and the earliest start time

Basic Block and Super Block Scheduling

Instructions scheduling is typically divided into two specific categories, local and global scheduling. Local scheduling, also known as basic block scheduling, is the scheduling of any sequence of code that executes consecutively without branch instructions, with the possible exception of call instructions. Global scheduling defines all other types of scheduling including those which consider the entire program.

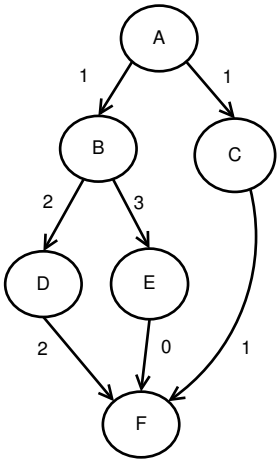
Local scheduling is the most common type of instruction scheduling. The cost function for basic blocks is a straight-forward measure of the length of a given schedule. Without loss of generality, we assume that a basic block has a single root node (a node with no precedence constraints) and a single sink node (a node with no successor constraints). The cost function for basic blocks can be formalized as,

$$cost(S) = ts(n) + etime(n), \tag{2.1}$$

where S is the schedule, $ts(n)$ is the time slot of the n^{th} (or sink) instruction of schedule S and $etime(n)$ is the execution time of the n^{th} instruction. The execution time of an instruction is the number of cycles needed to process the instruction on a functional unit.

Example 2.2 *Consider the six node basic block in Figure 2.1. The latencies between instructions vary depending on the parent and successor instruction. For*

Figure 2.1: A simple basic block and corresponding optimal schedule for a single issue processor.



Time Slot	Instruction
1	A
2	B
3	C
4	D
5	E
6	F

example, the latency between instruction B and E is three, which means that instruction E cannot begin execution until three clock cycles have elapsed after B has begun execution. Assume the execution time of the final instruction F be one clock cycle. Applying Equation 2.1, the cost of the optimal schedule is $ts(F) + etime(F) = 6 + 1 = 7$ cycles for a single issue machine.

It has long been conjectured [9] that the scope of local scheduling ignores certain parallelism and creates inefficient code. While global scheduling could include the entire scope of the program as in the work of Bernstein et al. [3], many of the global scheduling techniques try to find a balance between the amount of parallelism available and the manageability of the code size. Programs can contain millions of instructions and thousands of branch instructions, especially in the presence of aggressive optimization techniques like loop unrolling [57], and it becomes difficult to apply techniques to the entire program efficiently.

Many different types of global scheduling have been proposed to determine the best way to break up code into manageable code sections. These include trace scheduling [16], super block scheduling [24], treegions [20] and hyperblock scheduling [35]. In this research, we focus on super block scheduling as this global scheduling technique is used in many production compilers. Super blocks are constructed from a sequence of basic blocks, $B_1, \ldots, B_n$, where there exists a precedence constraint such that the exit of B_i must be scheduled before the exit of B_j if B_i precedes

B_j in the super block. Super blocks are constructed to have a single entrance into the code but multiple possible exits or branches out of the code.

In order to schedule a super block, we must define a cost function that we wish to minimize. For basic blocks, the cost function is simply the length of the schedule but for the super block scheduling problem a more complex function is used. The increased complexity comes from the existence of multiple exits allowing for different paths, and path lengths, out of a super block. Since there are multiple paths that need to be minimized, it is possible that minimizing some paths may increase the path lengths for other paths. This leads to a situation where it is necessary to weight the paths by relative importance. This weighting is normally done by using profile data collected about the exits. Profile data is collected by compiling the program without scheduling and executing a standard reference set of data. For each exit, the frequency that the exit is taken is recorded and these values are converted into percentages. The weights, called exit probabilities, are used in determining the cost function. So instead of minimizing the path length, we minimize the weighted path lengths to all branches. The cost of a schedule S, which we wish to minimize, can be defined more formally as,

$$cost(S) = \sum_{b \in B} w(b) ts(b), \tag{2.2}$$

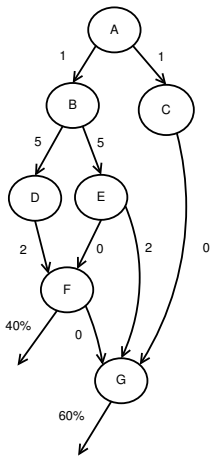
where B is the set of all branches in the schedule S, $w(b)$ is the exit probability of branch b and $ts(b)$ is the time slot of b in the given schedule S.

Example 2.3 *Consider the super block shown in Figure 2.2. The two exits from the graph are from instructions F and G. Each exit is marked with a corresponding exit probability. The minimum cost schedule is shown for a single issue processor and the minimum cost of the schedule can be calculated using Equation 2.2. Instruction F is scheduled at time cycle nine and instruction G is scheduled at time cycle ten. Thus, the cost of the schedule is* $\sum_{b \in B} w(b) ts(b) = 0.40 \times 9 + 0.60 \times 10 = 9.60$ *clock cycles.*

Scheduling Algorithms and the List Scheduling Algorithm

The most commonly used algorithm for basic block and super block scheduling is list scheduling [44]. The list scheduling algorithm uses a dependence DAG and a partial schedule to create a list of instructions that can be scheduled at a given

Figure 2.2: A simple super block and corresponding minimum cost schedule for a single issue processor. A nop (No OPeration) instruction is used as a place holder when their are no instructions available to be scheduled.



Time Slot	Instruction
1	A
2	B
3	C
4	nop
5	nop
6	nop
7	D
8	E
9	F
10	G

time slot dependent on the functional unit being scheduled. List scheduling is a greedy algorithm which uses a heuristic to select instructions from the ready list at a given time step and the best instruction is added to a partial schedule. The ready list is defined to be the list of instructions that can be scheduled at a given time respecting precedence constraints for a given functional unit. The primary goal of the heuristic is to determine, either by comparing the values of pairs of instructions or comparing the whole list, the instruction most likely to lead to a good schedule. By comparing pairs of instructions, the total list can be compared in a manner similar to finding the maximum within a list of unsorted numbers. The list scheduling algorithm increments the time after all functional units have been filled or there are no instructions available to schedule in the remaining functional units. The algorithm terminates when all instructions have been scheduled. The scheduling algorithm does not necessarily find optimal schedules since the scheduling algorithm performs a greedy search and is primarily used for simplicity and efficiency. The quality of the list scheduling algorithm is entirely dependent on the quality of the heuristic used by the scheduler to select instructions from the ready list. Section 3.1 discusses several different list scheduling heuristics for instruction scheduling.

There have been other scheduling algorithms used for instruction scheduling. These include linear methods [40], backtracking algorithms [1], and optimal solu-

tions using enumeration [50], integer linear programming [58] and constraint programming [56, 37]. Linear methods are more efficient than the list scheduling algorithm but the results produced are not as accurate. The backtracking algorithms and optimal solutions can generate better solutions but they can be inefficient.

2.2 Machine Learning

Machine learning, as a discipline, can be broken into different sub-fields including supervised, semi-supervised and un-supervised learning. Supervised learning comprises the set of techniques where all learning is done from a data set that is labelled with the correct answer. Un-supervised learning attempts to determine structure within a problem without ever consulting a labelled data set. The material in this section is based on the presentation in [41].

Supervised learning comprises a broad range of techniques that can include rule based learners, decision tree learners, neural networks, and Bayesian networks. The premise behind most supervised learning techniques is to give the program a set of labelled data and to induce a classifier from that data. The classifier induced from the labelled data is a function, which takes the feature values as input and returns a classification as an output. More formally, a classifier C is a function which takes a vector of inputs, x, and produces a classification, c, where every element in the vector x is a feature value and c is an element of the set of possible classifications for a given problem. The labelled data is typically called a set of instances, where an instance is a set of feature values and a correct classification. The difference between the supervised techniques lies primarily in the bias of the method and the algorithm by which the classifier is induced. The inductive or learning bias denotes the inherent limitation of a method that restricts the possible set of learned classifiers. Decision trees are formed by making a hierarchy of available features therefore allowing only functions that can be expressed in this form. A simple linear method may use a sum of weighted features allowing for an entirely different set of possible classifier functions. It is practically and theoretically impossible to remove all inductive bias from a classifier. Furthermore, it is detrimental to the functionality of the learning algorithm as without a bias there is no basis for making predictions on unseen data.

Recall that a supervised learning method requires a set of data labelled with correct answers. To properly learn a function, this set of instances is often broken into three separate sets of data. These sets are called the training set, the validation set and the testing set. The training set comprises the set of instances used to

directly induce a classifier function. To improve the accuracy of the classifier, it is often useful to keep a set of instances, called the validation set, to decide when to stop the learning algorithm. The process of using the validation set to iteratively improve results is called cross-validation. The testing set is a set of instances kept aside until the final classifier is complete to test the accuracy of the classifier on unseen data.

The inclusion of the validation set allows the classifier to be refined in order to avoid the pitfall of over-fitting. The pitfall of over-fitting can be defined as the learning of a classifier function that performs better on the training set over another function that performs better on the entire testing set. In other words, a classifier is over-fit when that function performs better only on a specific set of training data and not on the general data.

2.2.1 Decision Trees

Decision tree learning is a supervised learning technique that builds classifier functions forming a tree where each internal node is a feature test and the leaves are classifications. Each internal node represents a branching point where the tree branches based on a feature test.

The classification of a instance is made by traversing a path within the decision tree until a leaf node is reached. The value of that leaf node gives the classification of the decision tree.

Example 2.4 *Consider the small decision tree in Figure 2.3. The decision tree has two features, $F1$ and $F2$, and two classes, $C1$ and $C2$. The features are both binary, yes/no, features. Let the instance to be classified have feature values of $F1$="yes" and $F2$="no". Referring to the classifier, the first feature test compares $F1$ and the value of "yes" leads down the left branch. The second feature test compares $F2$ and the value of "no" leads down the right branch to a leaf node. The value of the leaf node is $C2$ so the instance is classified as $C2$.*

A decision tree is built using a simple algorithm. First, we determine a heuristic value for the quality of each feature given the training data set. Second, we select the feature with the highest heuristic value. Third, we break up the set of training instances into new subsets for each possible value of the selected feature, where new sets contain only those instances that have identical values for the selected feature. We continue this process of applying the heuristic and selection recursively on the

new subsets of data. When no more features are left to select or no reasonable division of the data can be made, the algorithm creates a leaf node with the most common classification of the instances. A reasonable division of data is considered to be any division where the heterogeneity of the classifications of the instances decreases. This case prevents the algorithm from adding more depth to the tree if the data is entirely of the same classification or if splitting the data on any of the remaining feature results in the same mix of classifications. Some decision tree algorithms also use a pruning criteria to limit the division of the set of instances to prevent over-fitting. One such algorithm is Quinlan's C4.5 algorithm [45]. One common heuristic for selecting instructions is to measure the information gained by adding that feature at that branch point over all other remaining features. Information gain can be defined in terms of the entropy within a set of instances. The entropy of a set of instances S is defined as,

$$Entropy(S) = \sum_{i=1}^{c} -p_i log_2 p_i,$$

where c is the set of all classes in the problem and p_i is the proportion of S with classification i. Once entropy is defined in terms of a given class, we define information gain for a given feature A on a set of instances S as,

$$Gain(S, A) = Entropy(S) - \sum_{v \in Values(A)} \frac{|S_v|}{|S|} Entropy(S_v),$$

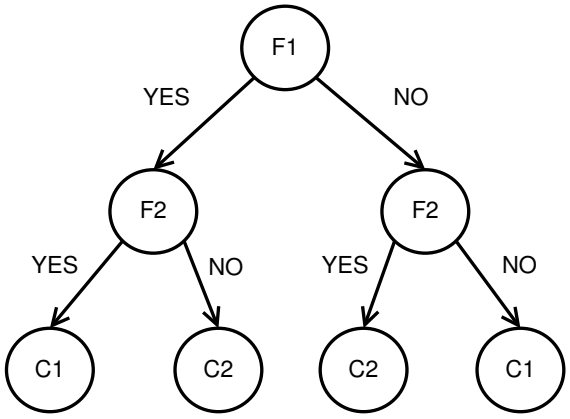
where $Values(A)$ is the set of possible values for the feature A and S_v is the subset of S where the feature A has value v.

Decision tree learning is one the most common simple learning techniques in use today. The advantages of decision trees lay in the relative simplicity of the output of the algorithm and efficiency of the algorithm. However, decision tree learning does have an inductive bias towards shorter more compact trees and requires extra techniques to handle continuous domains.

2.2.2 Feature Selection

Feature selection is the process by which features are removed from consideration based on certain criteria. There exist several different techniques for feature filtering (see [5, 19] and the references therein). Blum and Langley [5] state that

Figure 2.3: A decision tree example with two classes, C1 and C2, and two binary features, F1 and F2.



the existence of irrelevant features can affect not only the efficiency of the learning algorithm but also the ability to avoid over-fitting and to create good generalized classifiers. They describe three different techniques that can be used to perform feature selection: embedded, filter and wrapper. Before each technique is described, it should be pointed out that all of these techniques can be performed in either direction: forward or backward. A forward selection technique starts with an empty set of features and adds features into the set based on the selection technique. A backward selection technique starts with the entire set of features and removes features from the set using the selection technique.

An embedded feature selection technique uses the algorithm for finding the classifier to perform feature selection. An example of an embedded feature selection technique is decision tree learning. The decision tree learning includes the features individually, stopping when there is no reasonable division of the instances or a pruning criteria has been met. This means that some of the features may be left out if not necessarily required. This is an embedded technique because the procedure of the algorithm selects the best features and removes the features if they are not necessary.

A filter feature selection technique uses a criteria separate from the learning algorithm to search the space of features and remove those features labelled as irrelevant. In other words, a filter technique scans the set of features calculating a given property, like information gain, and removes those features whose properties do not meet a given threshold criteria. Filter techniques are inexpensive and can

be applied to features individually.

A wrapper feature selection technique uses a learning technique as a sub-method for performing feature selection. The learning technique learns a classifier using only a single feature. This classifier is then applied to the training set and the individual accuracy of that feature is recorded. Using this data, it is possible to identify those features that classify instances poorly and a thresholding method can be applied to remove the features.

2.3 Summary

In this chapter, we introduced the basic block and super block instruction scheduling problems. We described the assumptions about the instruction scheduling problem that we are making for this research. The necessary background material on machine learning and decision trees was also presented. In the next chapter, we describe two different areas of work related to this thesis. First, we describe previous hand-crafted instruction scheduling heuristics for both basic blocks and super blocks. Second, previous attempts to generate automated heuristics for scheduling are explored.

Chapter 3

Related Work

The problems of instruction scheduling for pipelined processors has be studied since the late seventies and many different heuristics have been proposed. [3, 14, 13, 9, 21]. One of the main components of our research was to extract the features used in previous approaches, along with defining new features and synthesizing old features. As well, it will be important to implement these algorithms as a comparison to the algorithm developed using the machine learning technique.

This chapter is separated into two main sections. The first section will describe previous heuristics for both basic block and super block instruction scheduling. The second section will describe the previous research in machine learning in scheduling and compilers.

3.1 Instruction Scheduling Heuristics

3.1.1 Local Scheduling Heuristics

In this section, we describe some of the previously proposed heuristics used for instruction scheduling. Each heuristic is described and an algorithm is provided for clarity. The notation introduced in Table 2.1 is used to describe the features in the heuristics. There have been a wide variety of previous heuristics for solving this problem. However, many of these heuristics are based on the same main features. In this section, we will present two general heuristic types, earliest start time and critical path heuristics, and describe how these general heuristic types can be used to generalize each of the previously proposed heuristics. Each of these heuristics is a hand crafted heuristic by the designer.

Earliest Start Time Heuristics

The earliest start time heuristics are the set of heuristics that use the critical path distance from the root to the node as the primary feature. The advantage of using this heuristic first is that it favours those instructions that have been delayed from their original release time. It does not, however, take into account the possibility of instructions that can and should be delayed due to slack in the schedule. Therefore, it may schedule instructions unnecessarily early with respect to other more important heuristics. Krishnamurthy [29] presents an earliest start time heuristic that includes execution time, path length to sink and critical path to sink as the other features in the heuristic. Warren [27] presents another earliest start time heuristic using alternating types of instructions, critical path to sink, number of uncovered children and original order as the other features of the heuristic. Algorithm 3.1 shows a typical earliest start time heuristic.

Algorithm 3.1: Earliest Start Time Heuristic

```
input  : Instructions i and j
output: Return true if i should be scheduled before j; false otherwise
if cp(1,i) < cp(1,j) then return true;
else if cp(1,i) > cp(1,j) then return false;
else
    if cp(i,n) > cp(j,n) then return true;
    else if cp(i,n) < cp(j,n) then return false;
    else
        if order(i) < order(j) then return true;
        else return false;
```

Critical Path Heuristics

The critical path heuristics are the set of heuristics that use the critical path distance to the sink node as the primary feature. This feature approximates the number of cycles that must execute prior to finish scheduling if the instruction is scheduled in the current slot. By using this heuristic, the instruction most likely to delay the schedule is selected to be scheduled. However, this measurement does not take into account the resource constraints on the schedule, which could indicate that another instruction is more likely to delay the schedule. Muchnick [44] presents a critical path heuristic that also uses earliest start time, execution time, the number of instructions uncovered, and original order as secondary features. Tiemann [55] presents a heuristic used in the GCC compiler that uses critical path distance

and original order as features. Schlansker [47] presents a heuristic that used critical path to sink and slack as features. Algorithm 3.2 shows a typical critical path heuristic where $updated\_est(1, i)$ is a recalculation of earliest start time to include any resource delays that may have occurred during scheduling.

Algorithm 3.2: Critical Path Heuristic

```
input : Instructions i and j
output: Return true if i should be scheduled before j; false otherwise
if cp(i,n) > cp(j,n) then return true;
else if cp(i,n) < cp(j,n) then return false;
else
    if updated_est(1, i) < updated_est(1, j) then return true;
    else if updated_est(1, i) > updated_est(1, j) then return false;
    else
        if order(i) < order(j) then return true;
        else return false;
```

Shieh and Papachristou's Heuristic

Shieh and Papachristou [49] created another critical path list scheduling heuristic that includes features such as execution time, number of children, number of parents and earliest start time (see Algorithm 3.3).

3.1.2 Global Scheduling Heuristics

In this section, we describe some of the previously proposed super block heuristics. We describe each feature and then present the heuristic used in the paper. An example, found in Figure 3.1, is used to describe how each of the heuristics work. The schedule found using the original order can be found in Figure 3.2 and the optimal schedule can be found in Figure 3.3. Super block scheduling is a simplification of the general scheme of trace scheduling proposed by Fisher [16, 17]. The super block simplifies the trace by only allowing basic blocks to be connected to the super block if there are no side entrances. This simplification allows the designer to more accurately approximate the cost of the code region as the problem is simplified from a many to many problem to a one to many problem. Berstein and Rodeh [3] present an early global scheduling heuristic based on the critical path heuristic from basic block scheduling. With the exception of Eichenberger and Meleis's balance scheduling heuristic [14, 39], each of the following heuristics are compared

Algorithm 3.3: Shieh and Papachristou's Heuristic

```
input  : Instructions i and j
output: Return true if i should be scheduled before j; false otherwise
if cp(i,n) > cp(j,n) then return true;
else if cp(i,n) < cp(j,n) then return false;
else
    if etime(i) > etime(j) then return true;
    else if etime(i) < etime(j) then return false;
    else
        if #_children(i) > #_children(j) then return true;
        else if #_children(i) < #_children(j) then return false;
        else
            if #_parents(i) > #_parents(j) then return true;
            else if #_parents(i) < #_parents(j) then return false;
            else
                if pl(1,i) ≤ pl(1,j) then return true;
                else return false;
```

against our generated heuristic in Chapter 5. The balance scheduling heuristic was removed because we were unable to obtain a copy of the code from the authors of the paper and we were not able to implement ourselves because of ambiguities in its description.

The G* Heuristic

Chekuri et al. [9] proposed a global scheduling technique that uses a profile independent scheduler (see Section 2.1) to create a ranking function for each possible branch-rooted subgraph of the dependence graph. Figure 3.5 shows Figure 3.1 broken into G* partitions. The ranking function for each subgraph G_b is defined as,

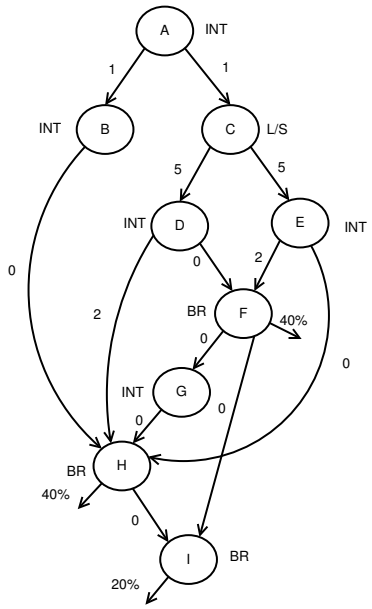
$$mrank(G_b) = \frac{length(G_b)}{\sum_{b \in branches(G_b)} w_b},$$

where $length(G_b)$ is the profile independent schedule length and $branches(G_b)$ is the set of all branches in the subgraph G_b.

They select, from the graph G, the subgraph with maximal rank or,

$$\underset{G_b \subseteq G}{\operatorname{argmax}}\, mrank(G_b).$$

Figure 3.1: A small super block used to describe the instruction scheduling heuristics.



Once the subgraphs are ranked, we need to define a heuristic function for scheduling the entire graph. We number the subgraphs removed from the graph by the selection process, in the order of removal, $1, \ldots, n$, where 1 is the first subgraph removed and n is the last subgraph removed. We define a function $sorder(i)$ which returns the ordering of the subgraph containing the instruction i. The heuristic is shown in Algorithm 3.4 where $ts_{sorder(x)}(x)$ is the time slot of instruction x in the profile independent schedule of subgraph $sorder(x)$.

Algorithm 3.4: G* Heuristic

input : Instructions i and j
output: Return true if i should be scheduled before j; false otherwise
if `sorder(i)` $>$ `sorder(j)` **then return** true;
else if `sorder(i)` $<$ `sorder(j)` **then return** false;
else
  if $\texttt{ts}_{sorder(i)}$(i) $<$ $\texttt{ts}_{sorder(j)}$(j) **then return** true;
  else return false;

Example 3.1 *Consider the super block shown in Figure 3.1. First we need to define the order of branches in the schedule. Using the mrank criteria, we find that,*

Figure 3.2: Schedule for the super block example scheduled using the original order of instructions from the code generator. The table shows the time slot of each instruction plus the corresponding assembly.

Time Slot	Instruction	Assembly
1	A	ADD $r1, $r0, $r2
2	B	ADD $r0, $r2, 1
3	C	LD $r1, #_mem_addr
4		nop
5		nop
6		nop
7		nop
8	D	MUL $r4, $r1, 1
9	E	MUL $r5, $r1, $r2
10		nop
11	F	BT $r5, location
12	G	ADD $r6, $r4, 1
13	H	BF $r4, location
14	I	BT $r2, location

Figure 3.3: Optimal schedule for the super block example showing the time slot of each instruction plus the corresponding assembly.

Time Slot	Instruction	Assembly
1	A	ADD $r1, $r0, $r2
2	C	LD $r1, #_mem_addr
3	B	ADD $r0, $r2, 1
4		nop
5		nop
6		nop
7	E	MUL $r5, $r1, $r2
8	D	MUL $r4, $r1, 1
9	F	BT $r5, location
10	G	ADD $r6, $r4, 1
11	H	BF $r4, location
12	I	BT $r2, location

Figure 3.4: Non-optimal schedule for the super block example showing the time slot of each instruction plus the corresponding assembly.

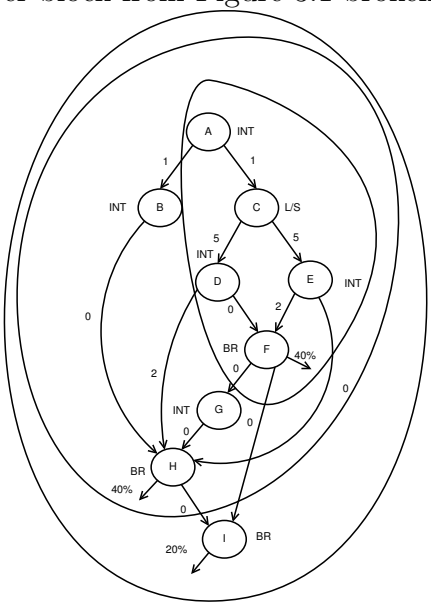
Time Slot	Instruction	Assembly
1	A	ADD $r1, $r0, $r2
2	C	LD $r1, #_mem_addr
3	B	ADD $r0, $r2, 1
4		nop
5		nop
6		nop
7	D	MUL $r4, $r1, 1
8	E	MUL $r5, $r1, $r2
9		nop
10	G	BT $r5, location
11	F	ADD $r6, $r4, 1
12	H	BF $r4, location
13	I	BT $r2, location

describing only the first calculation in detail, $mrank(G_F) = \frac{length(G_F)}{\sum_{b \in branches(G_F)} w_b} = 9/40 = 0.225$, $mrank(G_H) = 0.1375$ *and* $mrank(G_I) = 0.12$ *where* G_b *is the subgraph rooted at b. Therefore F is selected first and removed from the graph. When applying the mrank criteria on the remaining graph, we find* $mrank(G_H) = 0.05$ *and* $mrank(G_I) = 0.05$ *so we select H because of order and remove the subgraph from the remaining graph. The mrank of the remaining subgraph is* $mrank(G_I) = 0.05$ *and I is selected last. Now that we have defined the order of the branches, the first scheduling decision between instruction B and C at time two can be decided. Since B and C are both in* G_F*, we need only to look at their time slots in the corresponding schedule.* $ts_F(B) = 3$ *and* $ts_F(C) = 2$*, so C is scheduled first. For the second scheduling decision, we note that D and E are also within the first subgraph and, since* $ts_F(D) = 6$ *and* $ts_F(E) = 7$*, we schedule instruction D first. Therefore, we obtain the schedule shown in Figure 3.4.*

The Speculative Hedge Heuristic

Speculative Hedge [13] attempts to measure the number of branches that are affected by scheduling the instruction and the impact of that scheduling action. In

Figure 3.5: The super block from Figure 3.1 broken into G* partitions.



order to properly formalize the heuristic, we first need to define some terms. Let the set of branches that are helped by scheduling an instruction i be denoted $HB(i)$ and the set of all branches that are descendants of instruction i be denoted $B(i)$. We say that a branch is helped by an instruction if failing to schedule that instruction will delay the branch in the schedule. Second, we need to define three of the features used in the heuristic: helped weight, helped count and minimum late time difference.

Helped weight can be formalized as,

$$helped\_weight(i) = \sum_{b \in HB(i)} w(b)$$

Helped count can be formalized as,

$$helped\_count(i) = |HB(i)|$$

Minimum late time difference can be defined as,

$$minimum\_lst(i) = \min_{b \in B(i)} cp(i, b)$$

Once we have defined these features, we can define the heuristic as shown in Algorithm 3.5.

Algorithm 3.5: Speculative Hedge Heuristic

```
input  : Instructions i and j
output: Return true if i should be scheduled before j; false otherwise
if helped_weight(i) > helped_weight(j) then return true;
else if helped_weight(i) < helped_weight(j) then return false;
else
    if helped_count(i) > helped_count(j) then return true;
    else if helped_count(i) < helped_count(j) then return false;
    else
        if minimum_lst(i) > minimum_lst(j) then return true;
        else if minimum_lst(i) < minimum_lst(j) then return false;
        else
            if order(i) < order(j) then return true;
            else return false;
```

Example 3.2 *Consider the super block shown in Figure 3.1. For the first scheduling decision, we need to define helped branches sets for B and C at time two to be $HB(B) = \{\}$ and $HB(C) = \{F, H, I\}$ (see Eichenberger et al. [14] for details). Therefore, the helped weight of instruction C is clearly larger than that of instruction B so C is preferred. For the scheduling decision at time seven, we define the helped branches set for D and E to be $HB(D) = \{H, I\}$ and $HB(E) = \{F, H, I\}$. Again, it is clear that the helped weight of E must be larger and instruction E is preferred and the final schedule can is shown in Figure 3.3.*

Bringmann's Priority Function for Superblocks

Bringmann proposes [6] a modification of Fisher's Speculative Yield heuristic for super blocks. It is a simple heuristic, which attempts to rank instructions by determining the weighted sum of the latest start times of the instruction to each successor branch. More concretely, Bringmann's Priority Function called Dependence Height and Speculative Yield (DHASY) can be described mathematically as done by Deitrich and Hwu [13] as,

$$dhasy(i) = \sum_{b \in B(i)} (w_b(cp(1, n) + 1 - (cp(1, b) - cp(i, b))) \tag{3.1}$$

where $B(i)$ is the set of all branches that are descendants of i.

The heuristic is shown in Algorithm 3.6.

Algorithm 3.6: Bringmann's Heuristic (DHASY)

input : Instructions i and j
output: Return true if i should be scheduled before j; false otherwise
if `dhasy(i)` > `dhasy(j)` **then return** true;
else return false

Example 3.3 *Consider the super block shown in Figure 3.1. First note that* $cp(A, I) = 8$, $cp(A, H) = 8$ *and* $cp(A, F) = 8$. *For the first scheduling decision of B and C at time two, we see that* $cp(B, F) = 0$, $cp(B, H) = 0$, $cp(B, I) = 0$, $cp(C, F) = 0$, $cp(C, H) = 0$ *and* $cp(C, I) = 7$. *Using these values in the dhasy formula, we obtain* $dhasy(B) = 100$ *and* $dhasy(C) = 800$ *so C is preferred. For the scheduling decision of D and E at time seven, we see that* $cp(D, F) = 0$, $cp(D, H) = 2$, $cp(D, I) = 2$, $cp(E, F) = 2$, $cp(E, H) = 2$, *and* $cp(E, I) = 2$ *so* $dhasy(D) = 220$ *and* $dhasy(E) = 300$. *Therefore, instruction E is preferred and we obtain the schedule shown in Figure 3.3.*

The Balance Scheduling Technique

Eichenberger and Meleis [14, 39] propose a heuristic for solving super blocks and a lower bound on the total cost of the schedule with respect to the exit probabilities. The heuristic proposed for solving super blocks relies primarily on determining a good set of instructions to keep on the ready list. To describe this heuristic, we define several sets. First, we define $NeedEach(b)$ to be the set of operations that must be scheduled in order to keep from delaying a branch or, more formally, $\{i \mid r(i, b) + currentTime \geq r(1, b)\}$ where $r(i, j)$ is the resource bound distance defined by applying the Langevin and Cerny bounds [30] to the graph. Second, we define $NeedOne$ to be the set of instructions where one of the instructions must be scheduled in order to keep from delaying the branch or, more formally, $\{i \mid i \in ERC_{b,cmin}\}$ where $ERC_{b,cmin}$ is the bound proposed by Hu [23] where c is minimum and there is no empty slots in the schedule. Third, we need to define $TakeEach$ to be the union of $NeedEach(b)$ sets such that there are enough resources for all instructions in each set. Last, we define $TakeOne$ to be the union of $NeedOne(b)$ sets such that there are enough resources for all instructions.

Once we have the sets, $TakeEach$ and $TakeOne$, we apply the Speculative Hedge heuristic using the union of the $TakeEach$ and $TakeOne$ sets as the ready

lists. If the sets are empty, Speculative Hedge is applied to a ready list computed in the normal manner.

Example 3.4 *Consider the super block shown in Figure 3.1. First we note that the latest start time with respect to every branch for instruction B is eight and for instruction C is one and the earliest start times for both instructions is one. Therefore using the techniques in the paper, we define the sets to be* $NeedEach[F] = \{C\}$, $NeedEach[H] = \{C\}$, $NeedEach[I] = \{C\}$, $NeedOne[F] = \{C\}$, $NeedOne[H] = \{C\}$ *and* $NeedOne[I] = \{C\}$. *Therefore, the TakeEach and TakeOne set is also* $\{C\}$ *and instruction C is scheduled first. For the second scheduling decision, we note that* $lst_F(D) = 8$, $lst_H(D) = 6$, $lst_I(D) = 6$, $lst_F(E) = 6$, $lst_H(E) = 6$ *and* $lst_I(E) = 6$. *We obtain* $NeedEach[F] = \{E\}$, $NeedOne[F] = \{E\}$, $NeedEach[H] = \{D, E\}$, $NeedEach[9] = \{D, E\}$, $NeedOne[8] = \{D, E\}$ *and* $NeedOne[9] = \{D, E\}$. *Given the selection heuristic for branches (order and weight), branch F is selected first and the TakeEach and TakeOne sets are both* $\{E\}$. *Therefore, we schedule instruction E first and we obtain the schedule shown in Figure 3.3.*

Comparison of Super Block Scheduling Algorithms

The super block heuristics were developed over a large period of time so the evaluation test sets differ from experiment to experiment. The summary of the information can be found in Table 3.1. The table shows the test set of each of the heuristics, the heuristics that each algorithm was tested against and a brief summary of the results. The test set of each describes the programs or benchmarks used to test the algorithms. The heuristics used for comparison are for the most part described in the preceeding sections with the exception of Successive Retirement, which is a simple heuristic that tries to schedule branches as soon as possible without consideration for following branches. The results column shows a variety of results. For Bernstein et al., the results show the percentage of speed up of their heuristic against the test heuristic on four programs. DHASY summarizes the complete results as a 1-4 times speedup on various programs. G* and Speculative Hedge show the improvement, in terms of speedup, over a variety of heuristics across all test blocks. The Balance Scheduling results shows the percentage of blocks that each heuristic solves optimally.

Table 3.1: A summary of previous experimental results for super block scheduling heuristics.

Heuristic	Test Set	Comparison Heuristics	Results
Berstein et al. [3]	SPEC 89 Benchmarks {li, eqntott, espresso, gcc}	CP heuristic similar to [27]	li–6.9%, eqntott–7.3%, espresso–0%, gcc–0%
DHASY [6]	6 non-numeric programs from SPEC 92 CINT plus 9 other non-numeric unix programs	CP heuristic	1-4 times speedup over CP
G* [9]	SPEC 92 Benchmarks	CP, Successive Retirement, DHASY	CP: 3.1%–4.1%, SR: 0.6%–3.6%, DHASY: 1.6%–2.8%
SPEC HEDGE [13]	SPEC 92 CINT Benchmarks {espresso, li, eqntott, compress, sc, ccl}	CP, Sucessive Retirement, DHASY	CP: 0%–15.6%, SR: 0%–4.6%, DHASY: -0.2%–1.6%
BALANCE [14]	SPEC 95 INT Benchmarks	CP, SR, G*, DHASY, SPEC HEDGE	% of optimal blocks: CP: 26.0%, SR: 29.2%, G*: 25.8%, DHASY: 36.9%, SPEC HEDGE: 64.1%, BALANCE: 65.7%

3.2 Using Machine Learning for Scheduling and Compilers

Moss et al. [8, 38, 43] discuss the feasibility of using machine learning to learn good instruction scheduling heuristics for basic blocks. They used several different kinds of machine learning techniques including function approximators, rule based learners and reinforcement learning. They used a small sample set of features taken from a hand-crafted compiler and produced a heuristic similar but slightly worse than the DEC heuristic [51] they were comparing against. One the deficiencies associated with this work is the limited data set available to the learning algorithm as only small programs could be solved optimally and, therefore, only small blocks were used to form the data set. McGovern et al. [38] try to alleviate this problem by using reinforcement learning. However, this improvement further complicates the learning process as reinforcement learning is a much more difficult problem, and supervised learning, as used in this thesis, is preferred whenever possible. We used an optimal scheduler to generate optimal schedules for large blocks allowing us to generate instances from large blocks and to compare our results not only in terms of other heuristcs but also in terms of optimality. On top of the data limitation, the small set of features used by Moss et al. reduces the power of the technique and relies heavily on the work of the hand crafted designer. We improve on this technique by using a much larger set of features along with a feature selection technique to improve the accuracy of our heuristic. Furthermore, we extended the technique to look at global as well as local scheduling (see Chapter 5).

Beaty et al. [2] developed a technique for instruction scheduling using genetic algorithms and discriminative polynomials. He used a set of features taken from the Rocket compiler [25] and used a genetic algorithm to weight these features. This technique while interesting suffers from several problems. First, the experimentation lacks transparency. While the set of features are reported, the specifics of the test data are not reported in terms of either numbers of blocks or source of blocks. Second, they state that they gain only a five percent improvement in accuracy over initial random solutions.

In recent work, Li and Olafson [33] learned heuristics for single machine job shop scheduling using decision tree learning. However, they created their training data using existing heuristics to classify the instances they created. This means that some of their training instances would have been incorrectly classified. As a result, the heuristics that they learn are never better than the original heuristics used to label the data. Our work has an advantage over this work in that we use a slower optimal scheduler to generate the training data (see Chapter 4).

Lee et al. [32] also present a job scheduling method that uses machine learning to learn release dates for jobs and schedule them using a genetic algorithm. This algorithm is quite expensive and requires several minutes to solve medium size problems.

Correa et al. [12] used a genetic algorithm to learn schedules for parallel processing schedules. To increase the accuracy and decrease the cost of the technique, they used a list scheduling algorithm to seed the initial genetic population. However instead of using real data, they generated a set of synthetic problems designed to approximate real world DAGs.

Calder et al. [7] performed static branch prediction using neural networks and decision trees. They proposed that branches could be predicted by profiling a set of corpus data and extracting the features of that data to predict future branch execution without profiling. Their approach is similar to ours as they use a wide variety of features for the problem, selecting the best features using machine learning. However, they do not prune irrelevant features from their feature set and may be introducing inefficiencies to the process. Jimenez and Lin [26] also propose a machine learning technique for learning branch predictions. They use a perceptron learning method where branches exit profiles are learned using the perceptrons.

There are other papers in the literature that deal with machine learning and compilers. Stephenson et al. [53, 54] look at machine learning for register allocation and for building predicated hyperblocks. Monsifrot et al. [42] look at a simple learning technique to determine when to unroll loops. Similar to the paper by Monsifrot et al., Long et al. [34] present a paper which uses machine learning to perform loop optimization in Java.

3.3 Summary

This chapter covered the instruction scheduling heuristics described for both basic blocks and super blocks. The heuristics were described in terms of the features used and the features themselves were formally defined. We also described previous work in the literature on the problem of developing automated techniques for creating compiler heuristics.

The next chapter describes our method for generating basic block heuristics. We describe the feature selection techniques, the features used, the collection technique and the heuristic selection algorithm. The experimental evaluation of our work is then described at the end of the chapter.

Chapter 4

Learning Basic Block Heuristics

The simplest form of instruction scheduling is the scheduling of basic blocks. The primary concern of basic block scheduling is to reduce the schedule length to improve the use of instruction level parallelism available in modern processors. Production compilers use the list scheduling algorithm coupled with a heuristic to schedule basic blocks. Many of these compilers use a critical path based heuristic. These heuristics are developed by compiler experts by choosing and testing many different subsets of features and different possible orderings on standard benchmarks—a potentially time-consuming process. For example, the heuristic developed for the IBM XL family of compilers "evolved from *many years* of extensive empirical testing at IBM" [22, p. 112, emphasis added].

In this chapter, we extend a methodology for automating the process of constructing instruction scheduling heuristics proposed by Moss et al.[43]. We extracted, from the literature, a wide variety of different features used in instruction scheduling. To better approximate the resource constraints for a given architecture, we developed new features using architectural information and added these features to the feature set. We used machine learning techniques from the artificial intelligence literature to extract the useful features from a wide set of possible features. Once we had a set of features, we used these features to collect instance data about scheduling decisions using an optimal scheduler to correctly label data. With these instances, we used a decision tree learning algorithm and a search technique to identify a good heuristic for instruction scheduling. The automatically constructed decision tree heuristic was compared against a popular critical-path heuristic on the SPEC 2000 benchmarks. On this benchmark suite, the decision tree heuristic reduced the number of basic blocks that were not optimally scheduled by up to 55% compared to the critical-path heuristic, and gave improved performance guarantees

in terms of the worst-case factor from optimality. This work has been published in a paper by Russell, Malik, Chase and van Beek [46].

4.1 Learning Function

Given the accuracy and efficiency of supervised learning as well as the large amount of labelled data available for our problem, we decided to use supervised learning techniques. In order to make good scheduling decisions, it is necessary to build a function—a classifier—that generalizes to all scheduling problems within the domain. In order to effectively build this function, the designer must first decide on the correct representation of the function and the correct features to build into that function.

We are learning a heuristic to be used within a list scheduling algorithm. Given this scheduling framework, we define the ready list within the list scheduling framework to be a list of instructions $[I_1, \ldots, I_k]$ which could be scheduled at the current time, given latency constraints, where $I_j \in I$ and I is the set of all instructions within the problem. We wish to develop a function that returns the instruction that leads to a schedule with the least cost as defined in Equation 2.1. Since the ready list is of variable size, it is simpler to define our heuristic, in the same manner as Cavazos [8], using pairs of instructions and to scan through the list to find the best instruction in a similar manner to finding the maximum element in an unsorted array. Given this framework and the notion of supervised learning classifiers, we can define our heuristic function as,

$$F : I \times I \rightarrow \{0, 1\},$$

where the function has a value of one, or returns true, if the first instruction is better than or equal to the second instruction and zero, or false, otherwise.

4.2 Feature Construction

In order to determine which instructions to select, it is necessary to give the classifier function a good set of features. The full set of features that we considered can be found in Appendix A. We used three types of features within the instruction scheduling domain: DAG related features, ready list related features and instruction level features. Features can be further separated into dynamic and static features. Dynamic features are those features that changed depending on the clock cycle and

the previously scheduled instructions. Static features do not change and can be calculated prior to scheduling.

The DAG related features are those features that are the same for each instruction within a given DAG. They are static features and can be calculated prior to scheduling. The DAG related features include many counting features including the number of instructions, number of instructions per type, number of functional units, and number of edges in the graph. Also included in DAG related features are averages, maximums and minimums of any property of the graph.

The ready list features include those features that are identical for each of the instructions for a given ready list but differ depending on which instructions have already been scheduled. Examples of ready list related features include the number of different types of instructions left to be scheduled, current time, ready list for each type of instruction, currently available spaces for instructions and the number of excess slots for instructions.

The instruction level features include all features that have values that vary depending on the given instruction and its location in the DAG and schedule. The majority of instruction level features were taken from the paper by Smotherman et al. [52]. We used all of the features in the Smotherman paper except those features relating to register pressure. Register pressure was omitted because we did not include register effects in our model. There are several of these features that require further explanation as they are central to this thesis. Recall the notation that we introduced in Table 2.1. The two most important features from Smotherman's paper are the path length and critical path features. The path length between two instructions i and j is defined to be the maximum number of arcs on any path from i to j The critical path from i to j is the maximum distance between two nodes i and j and is denoted $cp(i, j)$. For a given node i, we can measure these features from both the root node and the sink node. The critical path distances to the root node and sink node are more commonly referred to as earliest start time and latest start time, respectively. We can also look at the updated values of the critical path features at each time step. As instructions are delayed from their original earliest start time, the path length from the root node increases. The difference between earliest and latest start times is called the slack in the schedule with respect to an instruction. The slack of an instruction denotes the number of cycles an instruction can be delayed without delaying the entire schedule. The original order of the instruction sequence should also be considered as this precedence relation can often be an effective tie breaking feature. It is important to consider the remainder of instructions when scheduling an instruction. Determining the count of the number of immediate successors and descendants provides estimates of the effect that

scheduling an instruction will have on the remainder of the schedule. Determining, the latencies of the successors of an instruction provides an approximation of the cumulative effect of delaying an instruction. The remainder of the features, i.e., those not found in Smotherman's paper, were synthesized and are described below.

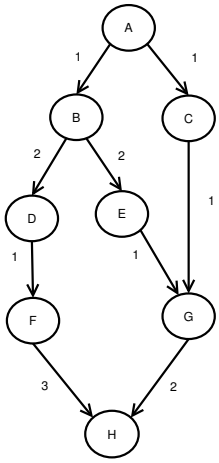
We synthesized two different types of novel instruction level features. The first type of features are measurements of the resource-based distances within the graph. The resource-based distances attempt to accurately measure the delay of each instruction in the graph due to resource constraints. The second type of features measures the availability of resources at a given time period and the ability of the current instruction to uncover instructions to fill the available resources.

Table 4.1: Notation for the resource-based distance to leaf node feature.

$desc(i,t)$	The set of all descendants of instruction i that are of type t in a DAG. These are all of the instructions of type t that must be issued with or after i and must all be issued before the leaf node can be issued.
$cp(i,j)$	The critical path distance from i to j.
$r_1(i,t)$	The minimum number of cycles that must elapse before the first instruction in $desc(i,t)$ can be issued; i.e., $\min\{cp(i,k) \mid k \in desc(i,t)\}$, the minimum critical-path distance from i to any node in $desc(i,t)$.
$r_2(i,t)$	The minimum number of cycles to issue all of the instructions in $desc(i,t)$; i.e., $\lvert\text{desc(i, t)}\rvert \ / k_t$, the size of the set of instructions divided by the number of functional units that can execute instructions of type t.
$r_3(i,t)$	The minimum number of cycles that must elapse between when the last instruction in $desc(i,t)$ is issued and the leaf node l can be issued; i.e., $\min\{cp(k,l) \mid k \in desc(i,t)\}$, the minimum critical-path distance from any node in $desc(i,t)$ to the leaf node.

One of the novel features that we synthesized, the resource-based distance to the leaf node, turned out to be one of the best basic block feature among all of the features that we studied. Consider the notation shown in Table 4.1. For convenience of presentation, we are assuming that a DAG has a single leaf node; i.e., we are assuming a fictitious node is added to the DAG and zero-latency arcs are added from the leaf nodes to this fictitious node. The resource-based distance from a node

Figure 4.1: (a) A small 8 node basic block where all instructions are integer instructions. (b) The table shows the resource-based ($rb(i)$) distance to the leaf node, critical path ($cp(i, H)$) distance to the leaf node and the maximum of the two distances for every node in the DAG.



(a)

Node i	$rb(i)$	$cp(i, H)$	Max
A	8	7	8
B	7	6	7
C	3	3	3
D	3	4	4
E	3	3	3
F	3	3	3
G	2	2	2
H	0	0	0

(b)

i to the leaf node is given by,

$$rb(i) = \max_t \{r_1(i,t) + r_2(i,t) + r_3(i,t)\},$$

where we are finding the maximum over all instruction types t.

Example 4.1 *Consider the small basic block shown in Figure 4.1(a). All instructions within this basic block are integer instructions, consequently we will not maximize the distance over different types, and, for this example, we will use a single issue architecture. For the sake of clarity, we show the calculation of resource-based distance in its entirety for only the first instruction and only present the results for the other instructions. The set of descendants for the root node of the graph, A, includes all of the remaining instructions of the graph. Therefore,* $r_1(A,t) = 1$ *because both immediate successors of the root node have a latency of one. Seven nodes are descendants of node A and the issue width of the example architecture is one, so* $r_2(A,t) = 7/1 = 7$*. Given that the leaf node is in the set of descendants, the minimum distance from a descendant to the leaf node is zero and* $r_3(A,t) = 0$*. Therefore, the resource-based distance for the root node is eight. The complete values along with comparative critical path distances can be seen in Figure 4.1(b).*

The distance was sometimes improved by "removing" a small number of nodes (between one and three nodes) from $desc(i,t)$. This was done whenever removing these nodes led to an increase in the value of $rb(i)$; i.e., the decrease in $r_2(i,t)$ was more than offset by the increase in $r_1(i,t) + r_3(i,t)$. To find a more accurate approximation of the distance, we find the maximum of the resource based distance and critical path distance to sink.

For the second type of novel features, we generated several features used to determine when scheduling an instruction will allow other instructions to be scheduled either in the current or next clock cycle. This occurs when the latency between successor instructions is either zero or one. If the latency is zero then the successor instruction becomes available to be scheduled in the current time slot. If the latency is one then the successor instruction becomes available in the next time slot.

4.3 Collecting Training, Validation and Testing Data

To perform supervised learning of any classifier, a large set of instances with correctly classified results is required. In our context, this requires the generation of features for each of the instructions, a comparison of those features and the correct classification. The correct classification of an instance comparing two instructions i and j would be true if i is better than j and false otherwise. This means that the cost of the schedule when i is scheduled next leads to an optimal schedule whereas scheduling j next does not. Whether scheduling an instruction next leads to an optimal schedule is determined by using an expensive optimal scheduler to finish scheduling the partial schedule created by the list scheduler with the current instruction scheduled in the next clock cycle. We used the optimal scheduler by Malik et al. [37] and modified the scheduler so that it could schedule partial schedules. We used two different sets of benchmarks to create training, validation and testing data for basic blocks. The training and validation data was taken from the MediaBench benchmarks Jpeg and Mpeg [31], respectively. The testing data included all of the benchmarks from the SPEC floating point and integer cpu benchmarks [11]. The SPEC benchmarks are discussed in further detail in Section 4.6.

The process of data collection is relatively straight forward. For every basic block in the benchmark suites, a modified list scheduler is used to generate data similar to the method used by Moss et al. [43]. Our method differs from the technique used by Moss et al. [43] because our optimal scheduler can solve large blocks exactly. We extract the ready lists from the list scheduler and find an

optimal schedule length when scheduling each instruction. Once these ready lists were extracted, we merged the list into a set of instances where optimal instructions are matched up to non-optimal instructions, their feature values are compared and the correct classification is appended to each instance.

The instances can be characterized by a set of feature value comparisons and a classification. Each feature value comparison can be expressed as $f_k(i, j)$, where f_k is a feature comparison function for feature k of instructions i and j. Therefore, an instance is a vector of the form,

$$instance(i, j, class) = \langle f_1(i, j), \ldots, f_n(i, j), class \rangle. \tag{4.1}$$

where *class* is the correct classification of the instance; i.e., whether instruction i is better than instruction j. Note that there is an exception to this scheme. The DAG related and ready list related features are presented in the instances as values and not comparisons because the values are always the same for every pair of instructions. However, none of these features survive the feature selection process described in Section 4.4 and therefore all instance values are comparisons.

The simplest way to compare instructions would be to use a greater than, equal to or less than scheme where each feature of an instruction is compared and given a value one, zero and negative one, respectively. This comparison function is useful because it is both simple to understand and is inexpensive to compute for feature value pairs. However, with this method, it is clear that some information is lost. It is possible to create scenarios where a feature means very little when it is only slightly greater than or less than a given comparison value but is very informative when the value varies wildly with the comparison value. Therefore, a more fine grained distinction of the feature values may lead to better classifiers. However for this project, we found that the finer grained distinction only increased the size of our classifier without making any significant gains in instruction scheduling accuracy.

Example 4.2 *Consider the small basic block shown in Figure 4.1(a). Suppose that all instructions within the DAG are integer instructions and assume we are collecting data for a single issue processor. For this example, we are collecting data for two features: critical path distance to root and critical path to sink. The modified list scheduling algorithm performs a data collection step for each ready list constructed. Since a single element on a ready list must be the optimal instruction, any ready list with a singleton instruction will not generate an instance. The first ready list with more than a single instruction would be* $[B, C]$. *Using an optimal scheduler, we find that selecting instruction* B *leads to a schedule of length ten while selecting instruction* C *leads to a schedule of length eleven. Therefore, we mark instruction*

B as the optimal instruction. We calculate the features for the two instructions to be $cp(A, B) = 1$ and $cp(A, C) = 1$ for critical path distance to root and $cp(C, H) = 3$ and $cp(B, H) = 6$ for critical path distance to sink. With this data, we can generate two instances: one positive for the optimal instruction, B, and one negative for the non-optimal instruction, C. The first instance instance$(B, C, 1)$, calculated using Equation 4.1 and the simple greater than, less than or equal comparison function, is $\langle 0, 1, 1\rangle$ and the second instance instance$(C, B, 0)$, calculated in the same manner, is $\langle 0, -1, 0\rangle$. Once we have generated every possible pair of instances from the ready list, in this case a single pair, we select an instruction from the optimal instructions to continue. For this example, we select instruction B as it is the only optimal instruction. This process would continue for the remainder of the ready lists. However for this example, every instruction on the remainder of the ready lists leads to an optimal schedule and, therefore, the algorithm generates no more instances.

4.4 Feature Selection

Once a set of instances have been created, the next step is to prune certain features from the feature space primarily due to the cost of generating each feature but also to remove redundant or irrelevant features. The reason for removing redundant or irrelevant features is to improve the quality of the heuristic. There are many learning algorithms, including decision tree learning, that perform poorly in the presence of redundant or irrelevant features [59].

One method for removing the features is to use a wrapper method involving the actual learning results using a training and a validation set. Wrapper methods are explained in more detail in Section 2.2. In our work, we only remove those features that have an accuracy on the validation set that is only a negligible improvement over a random guess. However, those features are not immediately discarded. This second stage is motivated by the observation that some features can combine with other features to gain more than an additive effect in terms of accuracy. To identify these features, we use the same wrapper method but allow for two feature trees where one feature in the tree must be a candidate for removal. If there is a non-additive improvement in any of the two featured trees for a given removal candidate, then that candidate is kept in the set of features.

On top of these two selection criteria, we remove one feature from any pair of features that are perfectly correlated. An example of two correlated features would be latest start time of a node and the critical path distance from the sink. These

highly correlated features sometimes arise when features are calculated using the other feature or, in the case of the above example, the same concept is calculated in two different ways.

Table 4.2 shows the set of features that remains after feature selection is applied to the complete list of basic block features shown in Appendix A. For succinctness, each feature is stated as being a property of one instruction. When used in a heuristic to compare two instructions i and j, we actually compare the value of the feature for i with the value of the feature for j. The features are shown ranked according to their overall value in classifying the data. The overall rank of a feature was determined by averaging the rankings given by three feature ranking methods: the single feature decision tree classifier, information gain, and information gain ratio (see [59] for background and details on the calculations). The feature ranking methods all agreed on the top seven features. The ranking can be used as a guide for hand-crafted heuristics and also for our automated machine learning approach, as we expect to see at least one of the top-ranked features in any heuristic. A surprise is that critical-path distance to the leaf node, commonly used as the primary feature [18, 44]), is ranked only in fifth place. Also somewhat surprising is that the lowest ranked features, features 14–17, are dynamic features. All of the rest of the features are static features.

4.5 Classifier Selection and Beam Search

Once the set of possibly useful features has been determined from the set of all features, the next step is to induce a heuristic from the instances generated from the benchmark suites. Given the large number of features, it is impractical to search the entire space of features to determine the exact tree that produces the best result on the validation set. On top of searching the entire space of possible classifiers using the features, there is the added cost of using a heuristic with a large number of features. Each added feature increases the cost of the heuristic and if the cost of features becomes large enough, the heuristic becomes impractical to use within a production compiler.

We used Quinlan's C4.5 algorithm [45] to learn a classifier from the instances and a forward selection technique with beam search [59, 19] to search through a likely set of heuristic approximations. The principle of beam search is relatively simple. Initially, we start with all single feature trees and then we keep the best k trees from the first round. The best k trees is defined to be the k trees having the highest classification accuracy on the validation set. Once the single feature trees

Table 4.2: Features remaining after filtering, ordered from highest ranking to lowest ranking.

1. Maximum of feature 2 and feature 5.
2. Resource-based distance to leaf node
3. Path length to leaf node.
4. Number of descendants of the instruction.
5. Critical-path distance to leaf node.
6. Slack—the difference between the earliest and latest start times.
7. Order of the instruction in the original instruction stream.
8. Number of immediate successors of the instruction.
9. Earliest start time of the instruction.
10. Critical-path distance from root.
11. Latency of the instruction.
12. Path length from root node.
13. Sum of latencies to all immediate successors of the instruction.
14. Updated earliest start time.
15. Number of instructions of type load/ store that would be added to the ready list for the next time cycle if the instruction was scheduled.
16. Number of instructions of type integer that would be added to the ready list for the current time cycle if the instruction was scheduled.
17. Number of instructions of type load/ store that would be added to the ready list for the current time cycle if the instruction was scheduled.

have been generated and pruned, the search looks a level deeper and adds to those k single feature trees every feature not previously used in the current tree. This makes a new set of trees, k times the number of features less the level of the current search to be more precise, and from this set of trees we keep only the best k trees. We iterate in this manner keeping the best k trees from each of the expansions. An illustration of this process can be seen in Figure 4.2. We can stop the search at any time but a good measure would be to stop when classification accuracy does not increase.

Table 4.3 shows for each level l the accuracy of the best decision tree learned with l features (stated as the percentage incorrectly classified), and the size of the decision tree (the number of nodes in the tree). When there were ties for best accuracy at a level, the average size was recorded.

We chose four features as the best trade-off between simplicity and accuracy. Increasing the number of features gives only a slight improvement in accuracy, but a relatively large increase in the size of the tree (1.8 – 2.8 times). Since there were

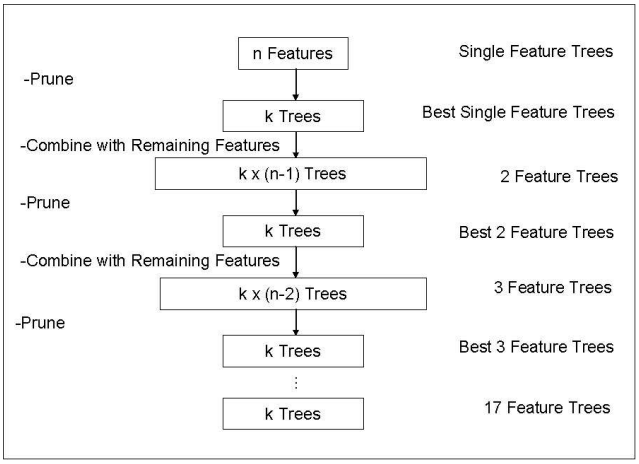


Figure 4.2: Combination and pruning steps for a beam search of width k on n features. The end result is k trees of depth n.

Table 4.3: For each level l, the accuracy of the best decision tree learned with l features (stated as the percentage incorrectly classified) and the size of the decision tree (the number of nodes in the tree).

level	1	2	3	4	5	6	7
accuracy	4.05	3.82	3.76	3.72	3.71	3.71	3.70
size	4	7	14	17	30	48	43

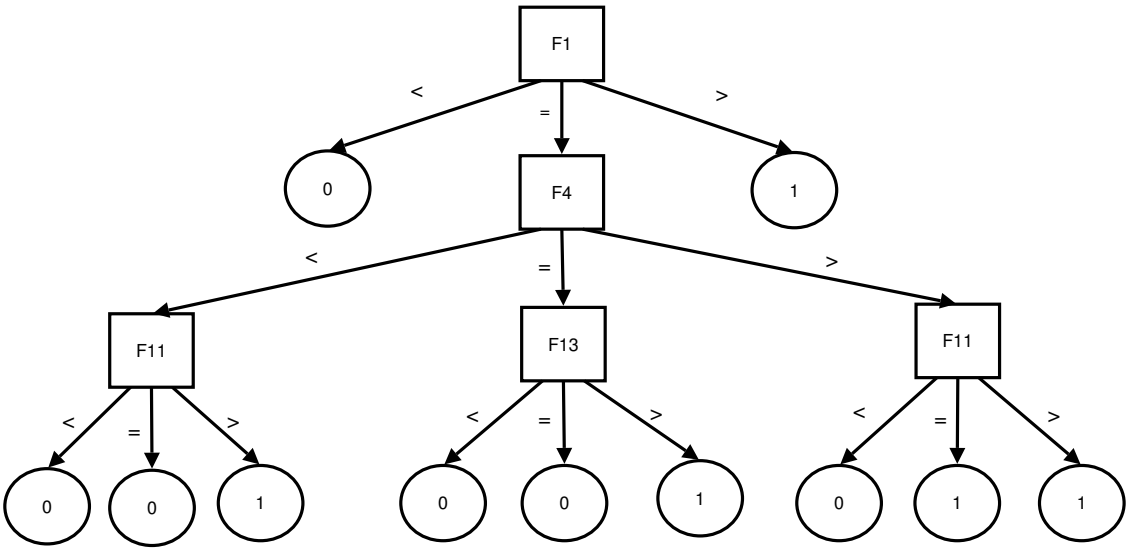


Figure 4.3: The final decision tree generated by the beam search. The labelled features f1, f4, f11 and f13 are the maximum distance, the number of descendants, the latency of an instruction and the sum of the latencies of each child, respectively.

ties for the best choice of four features, decision trees for the subsets of four features tied for best were learned over again, this time using all of the data (the validation set was added into the training set, a standard procedure once the best subset of features has been chosen). The smallest tree was then chosen as the final tree. The final tree is shown in Figure 4.3. In contrast to Moss et al. [43], who did not perform feature filtering and used all of the features in the training data at once to learn a classifier, our use of forward selection with beam search led to a smaller yet more accurate heuristic. The final decision tree heuristic constructed is shown in Algorithm 4.1.

4.6 Experimental Evaluation

In order to develop and test compiler designs and computer architectures, a standard set of benchmark programs have been developed by the Standard Performance Evaluation Corporation (SPEC)[http://www.spec.org]. The benchmarks, called the CPU Integer and Floating point benchmarks, consist of a suite of programs designed to cover a wide range of processor capabilities and application areas. In this work, the 2000 release of the benchmark was used and will be referred to as the

Algorithm 4.1: Automatically constructed decision tree heuristic for a basic block list scheduler.

```
input  : Instructions i and j
output: Return true if i should be scheduled before j; false otherwise
i.max_distance ←
max(i.resource_based_dist_to_leaf, i.critical_path_dist_to_leaf);
j.max_distance ←
max(j.resource_based_dist_to_leaf, j.critical_path_dist_to_leaf);
if i.max_distance > j.max_distance then
    return true;
else if i.max_distance < j.max_distance then
    return false;
else
    if i.descendants > j.descendants then
        if i.latency ≥ j.latency then return true;
        else return false;
    else if i.descendants < j.descendants then
        if i.latency > j.latency then return true;
        else return false;
    else
        if i.sum_of_latencies > j.sum_of_latencies then return true;
        else return false;
```

SPEC 2000 benchmarks. The programs consist of a variety of C/C++ and Fortran programs that range is usage from scientific computation to Unix tools.

We compiled the SPEC 2000 benchmarks using IBM's Tobey Compiler [4] and extracted 497,907 basic blocks. The basic blocks generated for this project were targeted for the PowerPC [22] assembly instruction set. The various instructions can be separated into four types that correspond to functional units within the processor. The types are integer instructions, floating point instructions, load/store instructions and branch instructions. Each of these types contains many different assembly instructions. For example, the branch unit includes both branch instructions and call instructions.

While there are four different types of instructions, and thus four types of functional units, different processor architectures have different numbers and arrangements of these functional units. In addition, it is possible for the processor to contain general purpose functional units that can execute two or more of the differ-

ent types within the same functional unit. We look at four hypothetical processor architectures on which to learn and test our scheduling heuristics. These four architectures were proposed by Shobaki and Wilken and are intended to cover a wide variety of general architecture models [50].

1-issue	processor executes all types of instructions.
2-issue	processor with one floating point functional unit and one functional unit that can execute integer, load/store, and branch instructions.
4-issue	processor with one functional unit for each type of instruction.
6-issue	processor with the following functional units: two integer, one floating point, two load/store, and one branch.

The latencies between the instructions in the DAG are generated for the IBM PowerPC architecture. The range of the latencies is: all 1 for branch instructions, 1–12 for load/store instructions (the largest value is for a store-multiple instruction, which stores to memory the values in a sequence of registers), 1–37 for integer instructions (the largest value is for division), and 1–38 for floating point instructions (the largest value is for square root).

Once a heuristic had been generated by the decision tree classifier, we tested the heuristic against several different basic block heuristics. These heuristics included a standard critical path heuristic, a heuristic using earliest start time (also known as critical path to root) as the primary feature and the Shieh and Papachristou [48] heuristic. These heuristics will be referred to as h_{cp}, h_{est} and h_{s+p}, respectively. The decision tree heuristic developed will be referred to as h_{dt}. The heuristic h_{est} is similar to the heuristic proposed by Warren [27]. The critical path heuristic was selected because of its similarity, in terms of features, to heuristics proposed by Muchnick [44] and Tiemann [55]. Also, it should be noted that h_{cp} would have been the heuristic learned by Moss et al. [43] when targeted towards our architectural model. Each of the heuristics is described in more detail in Section 3.1.

Since one of the goals of this research was to develop accurate heuristics automatically, we measure the accuracy of the heuristic by determining the number of times the heuristic fails to find an optimal schedule. Table 4.4 shows the number of non-optimal schedules created by each of the four heuristics being tested. It can be seen from the table that the decision tree heuristic performs better in terms of optimality as fewer schedules are found to be non-optimal. The results show that the critical path and Shieh & Papachristou heuristics are very similar in terms of optimality, while the earliest start time heuristic performs rather poorly. The decision tree heuristic improves the number of non-optimally scheduled blocks on

every architecture compared to every other heuristic.

Table 4.4: Number of basic blocks in the SPEC 2000 benchmark suite not scheduled optimally by the earliest start time heuristic (h_{est}), the critical-path heuristic (h_{cp}), Shieh and Papachristou's heuristic (h_{s+p}), and the decision tree heuristic (h_{dt}).

	1-issue				2-issue			
# blocks	h_{est}	h_{cp}	h_{s+p}	h_{dt}	h_{est}	h_{cp}	h_{s+p}	h_{dt}
497,407	51,137	4,385	3,786	1,960	44,880	5,523	4,919	2,593

	4-issue				6-issue			
# blocks	h_{est}	h_{cp}	h_{s+p}	h_{dt}	h_{est}	h_{cp}	h_{s+p}	h_{dt}
497,407	31,574	6,049	5,528	2,755	12,096	2,773	2,841	1,966

Recall that the critical path heuristic is the heuristic that would have been learned in the work by Moss et al. [8] and closely approximates the heuristic used in the GCC compiler [55]. We chose to compare the critical path heuristic against the decision tree heuristic on a comparison of non-optimally scheduled blocks broken down by the size of the blocks. The results of this experiment can be seen in Table 4.5. The percentage improvement ranges from 29.1% to 55.3% over the different architectures. It can be observed that the new decision tree heuristic optimally scheduled more blocks from the smaller and medium size ranges but offers less improvement for larger size blocks.

In terms of the overall heuristic quality, it is necessary to look at not only optimality but also the quality of the heuristic when schedules are non-optimal. We compared the critical path heuristic with the decision tree heuristic and recorded the number of times that each heuristic generated better, or in this case shorter, schedules than the other. Table 4.6 shows the result of this experiment. Overall, the decision tree heuristic generated 4.8, 6.0, 7.8 and 2.9 times more shorter schedules than the critical path heuristic for the different respective architectures. Only in the 251–2600 case of the single issue processor does the critical path heuristic marginally out perform the decision tree heuristic.

In order to look at the worst case performance of the heuristic, we looked at the maximal ratio from optimal by any schedule created by the heuristic. Table 4.7 shows the results of this experiment. The decision tree heuristic can vary from the optimal schedule by as much as 33% while the critical path heuristic can vary by as much as 39%. Interestingly, there seems to be no correlation with size to the

maximum distance from optimal as the numbers vary significantly as the size of the blocks grow.

4.7 Summary

This chapter described the proposed method for automated generation of basic block heuristics. The features of the problem were highlighted and methods were presented for identifying the useful features. A new feature integrating resource constraints into the problem of critical path distances was described in detail. The collection method, using the pairing of list scheduler and optimal scheduler, was also described. The decision tree heuristic, learned using the beam search to limit the size of the tree and increase generalization, was tested against the major basic block scheduling heuristics and the new heuristic outperformed all of the previous heuristics across all of the test measures.

The decision tree heuristic was tested against a critical path heuristic and found to have improvements in terms of optimality and worst case behaviour. Between 55% and 29% less basic blocks were found to have non-optimal schedules across all architectures. When comparing non-optimal schedules, the decision tree heuristic had between 2.9 and 7.8 times more improved blocks across all architectures. It was also found that the the worst case behaviour of the heuristic ranged from 17% and 33% from optimal while the critical path heuristic varied from optimal by between 21% and 39% across all architectures.

The next chapter extends this method for learning basic block heuristics to the global scheduling domain and proposes a method for the automated learning of super block heuristics. Many of the techniques described in this chapter will be used in the next chapter.

Table 4.5: Number of basic blocks in the SPEC 2000 benchmark suite not scheduled optimally by the critical-path heuristic (h_{cp}) and the decision tree heuristic (h_{dt}), for ranges of basic block sizes and various issue widths. Also shown is the total number of basic blocks in each size range and the percentage improvement given by the decision tree heuristic ($\% = 100 \times (h_{cp} - h_{dt})/h_{cp}$).

		1-issue			2-issue		
range	# blocks	h_{cp}	h_{dt}	%	h_{cp}	h_{dt}	%
1–5	324,352	338	0	100.0	350	0	100.0
6–10	94,066	804	134	83.3	907	165	81.8
11–20	46,502	1,118	598	46.5	1,226	586	52.2
21–30	13,911	619	290	53.2	781	347	55.6
31–50	9,760	628	337	46.3	853	512	40.0
51–100	5,669	536	315	41.2	790	464	41.3
101–250	2,789	270	218	19.3	505	408	19.2
251–2,600	358	72	68	5.6	111	111	0.0
Total	497,407	4,385	1,960	55.3	5,523	2,593	53.1

		4-issue			6-issue		
range	# blocks	h_{cp}	h_{dt}	%	h_{cp}	h_{dt}	%
1–5	324,352	182	12	93.4	0	0	—
6–10	94,066	736	121	83.6	69	56	18.8
11–20	46,502	1,623	681	58.0	534	344	35.6
21–30	13,911	962	479	50.2	584	469	19.7
31–50	9,760	1,013	548	45.9	615	437	28.9
51–100	5,669	915	455	50.3	538	318	40.9
101–250	2,789	501	358	28.5	337	251	25.5
251–2,600	358	117	101	13.7	96	91	5.2
Total	497,407	6,049	2,755	54.5	2,773	1,966	29.1

Table 4.6: Number of basic blocks in the SPEC 2000 benchmark suite where the critical-path heuristic gave a better schedule (h_{cp}) and where the decision tree heuristic gave a better schedule (h_{dt}), for ranges of basic block sizes and various issue widths. Also shown is the ratio of the number of improvements ($r = h_{dt}/h_{cp}$).

	1-issue			2-issue			4-issue			6-issue		
range	h_{cp}	h_{dt}	r	h_{cp}	h_{dt}	r	h_{cp}	h_{dt}	r	h_{cp}	h_{dt}	r
1–5	0	338	—	0	350	—	0	170	—	0	0	—
6–10	33	708	21.5	32	791	24.7	7	625	89.3	5	18	3.6
11–20	145	677	4.7	127	785	6.2	63	1,006	16.0	70	260	3.7
21–30	90	423	4.7	83	544	6.6	89	603	6.8	128	233	1.8
31–50	115	422	3.7	144	553	3.8	128	669	5.2	80	288	3.6
51–100	113	355	3.1	104	545	5.2	115	644	5.6	86	341	4.0
101–250	103	155	1.5	116	272	2.3	83	301	3.6	78	184	2.4
251–2,600	46	39	0.8	47	51	1.1	36	60	1.7	29	36	1.2
Total	645	3,117	4.8	653	3,891	6.0	521	4,078	7.8	476	1,360	2.9

Table 4.7: Maximum percentage difference from optimal for the critical-path heuristic (h_{cp}) and the decision tree heuristic (h_{dt}), for ranges of basic block sizes and various issue widths.

	1-issue		2-issue		4-issue		6-issue	
range	h_{cp}	h_{dt}	h_{cp}	h_{dt}	h_{cp}	h_{dt}	h_{cp}	h_{dt}
1–5	20	0	33	0	25	25	0	0
6–10	20	13	30	14	33	25	33	33
11–20	21	17	27	17	38	17	25	20
21–30	14	10	19	10	29	25	18	20
31–50	16	7	25	25	32	15	21	21
51–100	10	9	20	13	39	12	29	24
101–250	10	9	27	24	28	16	32	16
251–2,600	1	2	24	26	16	13	5	6
Maximum	21	17	33	26	39	25	33	33

Chapter 5

Learning Super Block Heuristics

In contrast to basic block scheduling, super block scheduling is complicated by a weighted evaluation function. Super blocks are made from several different basic blocks and contain multiple exit points and thus create an interesting scheduling problem that requires the minimization of a cost function dependent on the weight of the side exits and the position of the side exits in the final schedule. This difference suggests that some of the features used in the scheduling heuristic should include profile information about the exits within the super blocks.

There has been no work proposing automated techniques for learning super block scheduling heuristics. We show that, using the machine learning methodology explained in the previous chapter, we can automatically generate a super block heuristic. As in the basic block scheduling technique, an effort was made to identify a set of new features that may produce good results. We develop a heuristic that, by selecting the appropriate features, improves the scheduling accuracy over current list scheduling heuristics for super blocks.

5.1 Feature Construction

Since super blocks are generalizations of basic blocks, all of the features used in basic block scheduling can be considered in the super block scheduling problem. In addition to these features, it is important to determine what other features have been useful in super block scheduling. Many of the features [9, 13, 14] used by the previously proposed, hand-crafted super block scheduling heuristics are derived from the fact there exist multiple exits within the super block. Another obvious addition is the profile information given about the multiple exits. This leads to

new features like weighted estimates, helped weight and helped count. Table 5.1 and Appendix B show all of the features considered for super block scheduling heuristics. The overall rank of a feature was again determined by averaging the rankings given by three feature ranking methods: the single feature decision tree classifier, information gain, and information gain ratio (see [59] for background and details on the calculations). The DAG related and ready list related features were removed because they were found to have very little impact on scheduling decisions during the basic block feature selection.

For super block scheduling, we created many different features using a combination of features from basic blocks along with the weights of the exits. One new feature that was added was the feature called Helped Cost (Table 5.1, feature 24). This feature is an extension of the features developed by Deitrich and Hwu in their paper on Speculative Hedge [13]. In their paper, they create two unique features called Helped Weight and Helped Count. The idea is to identify those branches which are helped by scheduling an instruction (see Section 3.1). These branches are referred to as the helped branches of an instruction i ($HB(i)$). The Helped Cost feature still uses the same branch identification scheme that allows Speculative Hedge to find helpful branches but, instead of just looking at the weight of the branch, the Helped Cost feature keeps track of the weighted critical path to the identified branch. More formally, the Helped Cost for an instruction i and the helped branches is,

$$helped\_cost(i) = \sum_{b \in HB(i)} cp(i,b)w(b), \tag{5.1}$$

where $cp(i,b)$ is the critical path distance from i to b and $w(b)$ is the exit probability of branch b.

Another new feature we added to this scheme is weighted measurement of the critical path distances to branches within the schedule (Table 5.1, feature 8). More formally, the weighted critical path of an instruction i is,

$$weighted\_cp(i) = \sum_{b \in B(i)} cp(i,b)w(b),$$

where $B(i)$ is the set of all branches which are descendants of instruction i.

One of the most important sets of features to be described in this section are the weighted estimates. As in the basic block features, we can more closely approximate

the distance to an instruction by finding a resource based distance and using the maximum of that distance and the simple critical path distance. We use these same techniques to approximate the distances between an instruction and the branches within the DAG. We estimate the total weighted distance in two ways. The first way is to simply take the estimates to every successor branch and weight them by their exit probability. Second, we use the concept of dependence height and speculative yield developed by Bringmann [6] and substitute our new estimates into the equation (see Equation 3.1 for details about the DHASY calculation).

First, we can define the weighted estimates as the following,

$$weighted\_estimate(i) = \sum_{b \in B(i)} est(i,b)w(b),$$

where $est(i,b)$ can be replaced by either the resource based distance $rb(i,b)$ (Table 5.1, feature 4) or the maximum distance $maxd(i,b)$ (Table 5.1, feature 3).

Second, we define the estimate and speculative yield features as,

$$estimate\&speculative\_yield(i) = \sum_{b \in B(i)} w(b)(est(1,n) + 1 - (est(1,b) - est(i,b))), \tag{5.2}$$

where $est(i,j)$ can again be replaced by either estimate (Table 5.1, features 1 and 2). The idea is to replace the critical path distances in the DHASY equation with our more accurate estimates of distance.

The quality of an instruction can often be determined by measuring a variety of tie breaking features where slight differences between instructions highlight the overall difference between an instruction that leads to a good schedule and a bad schedule. Therefore, it is important to highlight these differences especially in the maximal and minimal ranges of values. The features that we created for this purpose are described below.

The maximum (or minimum) distance to branch for an instruction i (Table 5.1, feature 15) is given by,

$$max\_db(i) = \max_{b \in B(i)} \{(cp(i,b))\}.$$

The weighted maximum (or minimum) distance to branch for an instruction i (Table 5.1, feature 11) is given by,

$$max\_wdb(i) = \max_{b \in B(i)} \{(w(b)cp(i,b)\}), \tag{5.3}$$

where $wdb(i)$ denotes the weighted distance to a branch.

When evaluating features within the basic block scheduling domain, we found that the slack of an instruction often showed information about instructions with equal distances to the sink node. Similarly when evaluating super block features, we need to determine the minimum slack for any instruction to any branch (Table 5.1, feature 17). The smaller this value, the less chance it can be delayed without delaying a branch instruction. The minimum slack to a branch is given by,

$$min\_slack(i) = \min_{b \in B(i)} \{(cp(1,b) - cp(i,b) - cp(1,i))\}.$$

As with all super block features, it is necessary to analyze the features in terms of its weighted value. The weighted minimum slack (Table 5.1, feature 12) is given by,

$$min\_wslack(i)) = \min_{b \in B(i)} \{(cp(1,b) - cp(i,b) - cp(1,i))w(b)\}.$$

Other features were added to the feature block scheduling to determine estimates of when the instruction will schedule within the graph. These features include resource delay, sink to root ratio, projected slot, projected schedule and estimated release time.

The resource delay of an instruction i is a dynamic feature that measures the number of remaining slots available to schedule all of the descendants of the current instruction and then finds delay attributed to this measurement (Table 5.1, feature 20). The resource delay can be formally defined as,

$$resource\_delay(i) = (maxDelay - currentTime) - (desc(i)/issueWidth),$$

where $maxDelay$ is the maximum critical path distance within the graph, $currentTime$ is the current time slot being scheduled, $desc(i)$ is the number of descendants for instruction i and $issueWidth$ is the number of functional units available for instruction scheduling.

As each of the instructions within the DAG are delayed from their original earliest start time, each node acquires an accumulated delay with respect to each

of the branches. This delay can be measured by finding the difference between the current scheduling slot and initial estimate of the earliest starting time (Table 5.1, feature 22). This cumulative delay can be formally defined as,

$$cumulative\_delay(i) = \sum_{b \in B(i)} cp(i,b) + (currentTime - cp(1,i)). \tag{5.4}$$

Again, as with other super block features, we extend the cumulative delay to find the cumulative cost by weighting each delay by the exit probability (Table 5.1, feature 23). The cumulative cost can be defined as,

$$cumulative\_cost(i) = \sum_{b \in B(i)} (cp(i,b) + currentTime - cp(1,i))w(b). \tag{5.5}$$

5.2 Collecting Training, Validation and Testing Data

As with basic blocks, we collected instance data from the super blocks generated from the Tobey compiler [4]. We used the same collection technique to acquire a set of instances from the blocks. We used the optimal super block scheduler developed by Malik et al. [36] and modified this scheduler so that it solves the partial schedules given to it from the list scheduling algorithm. Once we could schedule partial super block schedules, we collected all of the relevant ready lists from the SPEC benchmarks [11], where relevant is defined to be those ready lists where there exists a mix of optimal and non-optimal choices. We converted the ready lists into instances by comparing each feature of every pair of optimal and non-optimal instructions for every ready list. Unfortunately, the Jpeg and Mpeg super blocks from the MediaBench benchmarks were not available so we divided the SPEC benchmarks into training, validation and testing data. We chose galgel and gap to be our testing and validation sets for two reasons. First, the galgel and gap instances make up approximately ten percent (9.2%) of the super block instances and, second, the gap and galgel blocks provide a good cross section of the data as this includes both integer and floating point blocks as well as both C and Fortran blocks.

Table 5.1: The top 25 features within the super block domain ordered from highest ranking to lowest ranking.

1. Resource-based distance and speculative yield (see Equation 5.2)
2. Maximum distance and speculative yield (see Equation 5.2)
3. Sum of feature 6 weighted by the exit probability for every branch
4. Weighted resource-based distance to every branch
5. DHASY—Brinmann's priority function (see Section 3.1)
6. Maximum distance—maximum of feature 7 and feature 13
7. Resource-based distance to leaf node
8. Weighted critical-path distance to each branch
9. Maximum possible cost (see Equation 5.3)
10. Minimum possible cost (see Equation 5.3)
11. Maximum of weighted critical-path distances to each branch
12. Minimum of the slack for each branch weighted by the exit probabilities
13. Critical-path distance to leaf node
14. Latest start time of the instruction
15. Maximum of critical-path distances to each branch
16. Sum of path lengths to each branch weighted by the exit probabilities
17. Minimum slack to each branch
18. Path length to leaf node
19. Number of descendants of the instructions
20. Resource delay—number of descendants that cannot be scheduled between current time and the maximum critical path distance
21. Slack—the difference between earliest and latest start times
22. Cumulative delay (see Equation 5.4)
23. Cost of cumulative delay (see Equation 5.5)
24. Helped cost—sum of costs to each helped branch (see Equation 5.1)
25. Minimum of weighted critical-path distances to each branch

5.3 Feature Selection

The technique to remove irrelevant features was identical to the technique used in basic block scheduling. We tested for irrelevant instructions that do not easily combine to create useful features. To this end, we removed any instruction that was nearly random, i.e. had an error of 49.5% or worse for a single featured tree. As with basic block scheduling, removal candidates are not removed immediately. We first determine if they have non-additive improvements when combined with a feature that is not a removal candidate. With this technique, we removed the uncovered next for all types, uncovered current float and rank. On top of irrelevant features, we removed perfectly correlated features like latest start time and earliest start time, which are correlated with critical path length to sink and critical path length to root, respectively. We also removed perfectly correlated features 48, 50, 52 and 57 which were correlated with the uncovered current features.

5.4 Classifier Selection

Once we generated the instances and pruned the irrelevant features, we implemented a beam search to generate an appropriate classifier using a beam width of thirty and searched to a depth of seven. The results of this search can be seen in Table 5.2. One can see that we achieve the best accuracy at depth of four without unnecessarily increasing the size of the tree. The final tree was constructed by combining both the training and validation set using the features identified by the beam search. The final tree can be seen in Figure 5.1. The tree shows that the primary feature is the Maximum Distance and Speculative Yield, which is one of the synthesized estimates of weighted distance described in Section 5.1. The secondary features are from two different sources. Helped Cost is also described in Section 5.1 and is an extension of the primary features of the heuristic developed by Deitrich and Hwu [13]. Path length from root and number of descendants are DAG properties carried over from basic block scheduling. We translate the decision tree into the algorithm shown in Algorithm 5.1.

5.5 Experimental Evaluation

We compared the classifier generated to four other heuristics including the critical path heuristic from basic block scheduling, dependence height and speculative yield

level	1	2	3	4	5	6	7
accuracy	1.4	1.2	1.2	1.1	1.1	1.1	1.1
size	4.0	7.0	7.6	49.0	58.6	68.8	78.6

Table 5.2: For each level l, the accuracy of the best decision tree learned with l features (stated as the percentage incorrectly classified) and the size of the decision tree (the number of nodes in the tree).

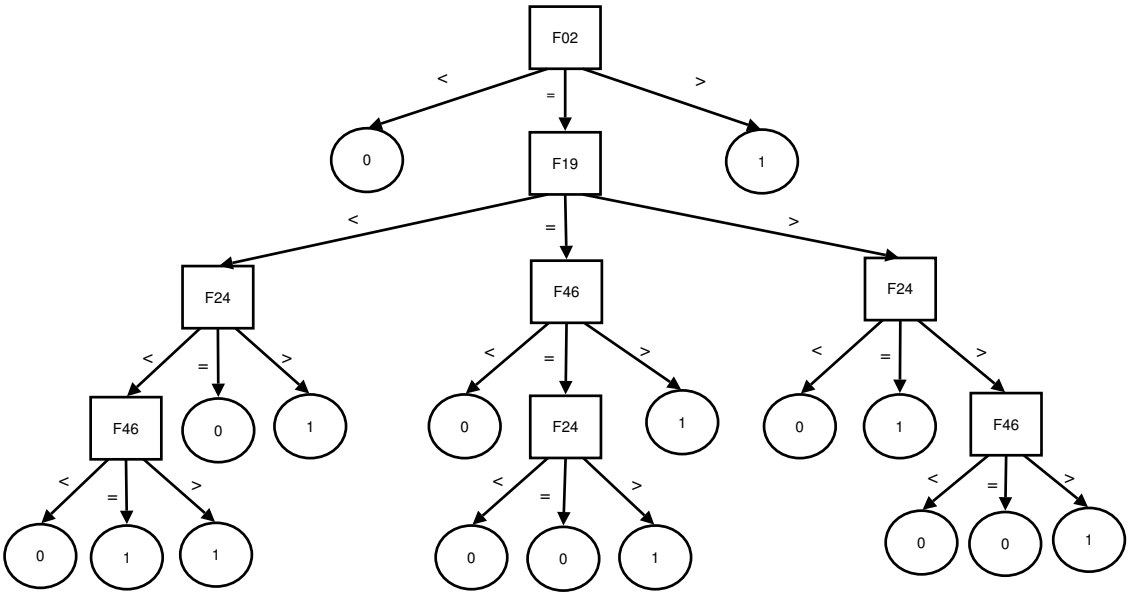


Figure 5.1: The final decision tree generated by the beam search. The labelled features f02, f19, f24 and f46 are maximum distance and speculative yield, number of descendants, helped cost and path length from root, respectively.

[6], G* [9] and Speculative Hedge [13]. The decision tree heuristic was tested on the remainder of the SPEC benchmarks against each of the heuristics. We examine the results in terms of non-optimal schedules generated, heuristic difference, and maximum percentage difference from optimal. We examine each of the heuristics under the same four architectures described in Section 4.6.

The critical path heuristic (h_{cp}) is the same heuristic used in basic block scheduling and is kept for super block scheduling as a comparison between profile independent and profile dependent heuristics. The dependence height and speculative yield heuristic (h_{dhasy}) was developed by Bringmann [6] and is an extension of Fisher's work on Trace Scheduling [17]. Bringmann's heuristic attempts to weight the path lengths to each branch while accounting for the maximum delay in the graph. The G* heuristic was developed by Chekuri et al. [9] (h_{g*}) and uses a profile independent scheduler and a ranking method to schedule super blocks. The last heuristic, Speculative Hedge [13] (h_{spec}), determines and weights for each instruction only the branches which are determined to be useful. More detailed descriptions of these algorithms can be found in Section 3.1.

We compare each of the schedules generated by the heuristics to the optimal schedule and determine the number of schedules generated that are more expensive than the optimal schedule. Tables 5.3–5.6 show the number of non-optimal schedules for each architectural model broken down by size[1]. We found that the decision tree heuristic reduced the number of non-optimally scheduled blocks by as much as 38% and by at least 16%. The h_{dhasy} performed second best while h_{spec}, h_{cp} and h_{g*} performed significantly worse. Surprisingly, h_{g*} actually creates more non-optimal schedules than the h_{cp} heuristic.

We compared the two best heuristics, h_{dt} and h_{dhasy}, in terms of schedule cost in order to determine how many times each heuristic out performs the other heuristic. Table 5.7 shows the results of this experiment. The decision tree heuristic outperforms Bringmann's heuristic by 4.2, 4.4, 3.6 and 2.6 times respective to each architecture. The heuristic h_{dhasy} outperforms h_{dt} only once and, in this case, only by three blocks for size 6–10 on the 6-issue architecture.

Once we know how many of the schedules are non-optimal, it is important to know the worst case behaviour of the heuristic. We measured the maximum percentage difference from the optimal schedule cost for each of the heuristic costs. Table 5.8 shows the maximum difference of each heuristic schedule from optimal broken down by size. We found that the h_{dt} heuristic performs slightly better

[1]Note that the number of super blocks compared varies across the architectures because the number of time outs by the optimal scheduler varies with respect to the architecture.

than the other heuristics for all but the 4-issue and 6-issue architectures and, more surprisingly, h_{g*} provided the next best heuristic for maximum difference from optimal except on the 4-issue architecture. For the 4-issue architecture, h_{dhasy} outperforms h_{g*} by 2.68 percent over optimal. Another interesting note about these results is that, while h_{g*} is often outperformed by h_{cp} and h_{spec} in terms of optimality, h_{g*} is more stable and has a maximum percentage increase of only 43% across all architectures.

Overall, we show that our heuristic does better than other heuristics on the SPEC 2000 benchmarks. We also show that the DHASY heuristic does well in terms of optimal blocks and the G* heuristic performs rather poorly. These results agree with previous results (see Table 3.1). In contrast to previous results, we observed that the Speculative Hedge heuristic performs well in terms of optimal results but does not compete with the DHASY heuristic. We also noted that the worst case behaviour of the Speculative Hedge heuristic is far worse than either the G* or DHASY heuristics when these heuristics generate non-optimal blocks.

5.6 Summary

Through the application of the techniques proposed in Chapter 4, we developed a heuristic that reduces the number of non-optimal blocks by at least 16% and up to 38%, depending on the architecture. The heuristic was found to outperform the next best heuristic by at least 2.6 times in terms of schedule cost. The new features developed for this technique were usefully incorporated into the super block decision tree heuristic. The cost of the heuristic was kept within reasonable limits as only four features were needed to obtain good results.

The last chapter of thesis briefly summarizes the work in this thesis and outlines the results obtained during this research.

Algorithm 5.1: Automatically constructed decision tree heuristic for a super block list scheduler.

```
input  : Instructions i and j
output: Return true if i should be scheduled before j; false otherwise
if i.max_dist_spec_yield > j.max_dist_spec_yield then
    return true;
else if i.max_dist_spec_yield < j.max_dist_spec_yield then
    return false;
else
    if i.descendants > j.descendants then
        if i.helped_cost < j.helped_cost then
            return false;
        else if i.helped_cost > j.helped_cost then
            if i.pl_root > j.pl_root then return true else return false
        else
            return true
    else if i.descendants < j.descendants then
        if i.pl_root < j.pl_root then
            return false
        else if i.pl_source > j.pl_source then
            return true
        else
            if i.helped_cost ≤ j.helped_cost then return false else return
            true
    else
        if i.helped_cost > j.helped_cost then
            return true;
        else if i.helped_cost < j.helped_cost then
            if i.pl_root ≥ j.pl_root then return true else return false
        else
            return false
```

Table 5.3: Number of super blocks in the SPEC 2000 benchmark suite not scheduled optimally by the decision tree heuristic (h_{dt}), the critical-path heuristic (h_{cp}), Bringmann's heuristic (h_{dhasy}), the G* heuristic (h_{g*}) and the speculative hedge heuristic (h_{spec}) for 1-issue architecture.

Range	# Blocks[1]	h_{dt}	h_{cp}	h_{dhasy}	h_{g*}	h_{spec}
3–5	30,371	104	143	124	129	143
6–10	46,615	437	3,095	720	2,551	1,935
11–15	33,687	869	4,058	1,555	3,902	2,939
16–20	23,511	718	3,363	1,469	3,835	2,639
21–30	22,845	1,120	4,333	1,802	5,224	3,273
31–50	17,713	1,302	4,168	1,954	4,966	3,299
51–100	9,389	881	2,482	1,321	2,960	2,012
101–250	2,633	379	795	491	861	676
251–2,600	322	68	131	80	131	101
Total	187,086	5,878	22,568	9,516	24,559	17,017

Table 5.4: Number of super blocks in the SPEC 2000 benchmark suite not scheduled optimally by the decision tree heuristic (h_{dt}), the critical-path heuristic (h_{cp}), Bringmann's heuristic (h_{dhasy}), the G* heuristic (h_{g*}) and the speculative hedge heuristic (h_{spec}) for 2-issue architecture.

Range	# Blocks[1]	h_{dt}	h_{cp}	h_{dhasy}	h_{g*}	h_{spec}
3–5	30,371	103	126	123	128	126
6–10	46,615	432	3,097	717	2,510	1,938
11–15	33,687	883	4,027	1,557	3,804	2,918
16–20	23,511	731	3,353	1,479	3,786	2,626
21–30	22,845	1,145	4,350	1,829	5,169	3,272
31–50	17,716	1,302	4,214	2,015	4,985	3,322
51–100	9,393	1,054	2,680	1,502	3,102	2,149
101–250	2,642	453	909	599	931	756
251–2,600	323	99	152	103	148	114
Total	187,103	6,202	22,908	9,924	24,563	17,221

Table 5.5: Number of super blocks in the SPEC 2000 benchmark suite not scheduled optimally by the decision tree heuristic (h_{dt}), the critical-path heuristic (h_{cp}), Bringmann's heuristic (h_{dhasy}), the G* heuristic (h_{g*}) and the speculative hedge heuristic (h_{spec}) for 4-issue architecture.

Range	# Blocks[1]	h_{dt}	h_{cp}	h_{dhasy}	h_{g*}	h_{spec}
3–5	30,371	0	6	1	6	6
6–10	46,615	185	1,009	228	1,075	978
11–15	33,687	494	1,894	788	2,045	1,662
16–20	23,512	517	1,694	759	1,905	1,504
21–30	22,858	876	2,774	1,250	3,166	2,310
31–50	17,765	916	2,737	1,444	3,219	2,233
51–100	9,479	947	2,051	1,385	2,404	1,740
101–250	2,671	458	726	603	770	680
251–2,600	333	111	129	119	142	118
Total	187,291	4,504	13,020	6,577	14,732	11,231

Table 5.6: Number of super blocks in the SPEC 2000 benchmark suite not scheduled optimally by the decision tree heuristic (h_{dt}), the critical-path heuristic (h_{cp}), Bringmann's heuristic (h_{dhasy}), the G* heuristic (h_{g*}) and the speculative hedge heuristic (h_{spec}) for 6-issue architecture.

Range	# Blocks[1]	h_{dt}	h_{cp}	h_{dhasy}	h_{g*}	h_{spec}
3–5	30,371	0	0	0	0	0
6–10	46,615	118	159	115	204	120
11–15	33,687	248	470	255	634	398
16–20	23,512	251	497	306	641	394
21–30	22,858	488	912	566	1,202	747
31–50	17,760	609	1,092	773	1,470	908
51–100	9,476	641	984	787	1,228	866
101–250	2,667	327	444	400	510	419
251–2,600	321	61	79	74	80	77
Total	187,267	2,743	4,637	3,276	5,969	3,929

Table 5.7: Number of super block blocks in the SPEC 2000 benchmark suite where Bringmann's heuristic gave a better schedule (h_{dhasy}) and where the decision tree heuristic gave a better schedule (h_{dt}), for ranges of super block sizes and various issue widths. Also shown is the ratio of the number of improvements ($r = h_{dt}/h_{dhasy}$).

	1-issue			2-issue			4-issue			6-issue		
range	h_{dhasy}	h_{dt}	r	h_{dhasy}	h_{dt}	r	h_{dhasy}	h_{dt}	r	h_{dhasy}	h_{dt}	r
1–5	1	21	21.0	1	21	21.0	0	1	–	0	0	–
6–10	88	385	4.4	88	387	4.4	76	121	1.6	4	1	0.2
11–15	186	909	4.9	182	894	4.9	104	397	3.8	28	36	1.3
15–20	166	968	5.8	164	963	5.9	121	368	3.0	34	81	2.4
21–30	256	1,025	4.0	257	1,035	4.0	161	560	3.5	47	121	2.6
31–50	245	1,020	4.2	235	1,055	4.5	150	750	5.0	75	246	3.3
51–100	167	679	4.1	173	741	4.3	152	637	4.2	126	307	2.4
101–250	135	266	2.0	108	315	2.9	100	307	3.1	74	192	2.6
251–2,600	24	41	1.7	26	46	1.8	23	56	2.4	11	34	3.1
Total	1,268	5,314	4.2	1,234	5,457	4.4	887	3,197	3.6	399	1,018	2.6

Table 5.8: Maximum percentage from optimal for the decision tree heuristic (h_{dt}), the critical-path heuristic (h_{cp}), Bringmann's heuristic (h_{dhasy}), the G* heuristic (h_{g*}) and the Speculative Hedge heuristic (h_{spec}) for ranges of super block sizes and various issue widths.

	1-issue					2-issue				
Range	h_{dt}	h_{cp}	h_{dhasy}	h_{g*}	h_{spec}	h_{dt}	h_{cp}	h_{dhasy}	h_{g*}	h_{spec}
3–5	26.9	26.9	26.9	26.9	26.9	26.9	26.9	26.9	26.9	26.9
6–10	37.5	65.8	37.5	37.5	44.7	37.5	65.8	37.5	37.5	44.7
11–15	25.6	82.0	25.6	25.6	65.7	26.7	82.0	26.7	25.6	65.7
16–20	24.4	98.2	24.7	26.9	98.2	24.4	98.2	24.7	26.9	98.2
21–30	16.7	159.3	28.7	27.9	155.6	16.7	159.3	28.7	27.9	155.6
31–50	17.6	192.2	35.2	31.1	143.7	17.6	192.2	35.2	31.1	143.7
51–100	12.1	246.0	40.9	37.7	246.0	12.1	246.0	40.9	37.7	246.0
101–250	13.8	170.6	13.8	20.5	133.5	13.8	170.6	13.8	20.5	133.5
251–2,600	5.8	86.4	5.8	6.8	24.4	7.3	86.4	7.3	11.6	15.2
Maximum	37.5	246.0	40.9	37.7	246.0	37.5	246.0	40.9	37.7	246.0

	4-issue					6-issue				
Range	h_{dt}	h_{cp}	h_{dhasy}	h_{g*}	h_{spec}	h_{dt}	h_{cp}	h_{dhasy}	h_{g*}	h_{spec}
3–5	0.0	22.2	20.0	22.2	22.2	0.0	0.0	0.0	0.0	0.0
6–10	40.0	41.1	40.0	41.1	40.0	22.2	28.6	22.2	28.6	25.8
11–15	21.1	47.4	33.3	41.1	47.4	22.2	31.4	22.2	22.2	31.4
16–20	21.9	73.7	22.2	34.6	73.7	21.9	52.7	21.9	23.8	52.7
21–30	19.1	152.9	37.8	42.7	152.9	21.1	80.4	21.1	22.1	80.4
31–50	28.6	136.0	28.6	35.0	129.7	29.4	61.5	29.4	29.4	61.5
51–100	11.7	136.5	34.2	39.1	133.0	14.9	106.0	29.3	18.4	48.9
101–250	10.8	556.1	14.5	16.8	556.0	12.8	285.3	9.6	9.6	285.3
251–2,600	4.5	962.1	7.6	19.8	962.1	3.2	478.1	3.3	6.4	478.1
Maximum	40.0	962.1	40.0	42.7	962.1	29.4	478.1	29.4	29.4	478.1

Chapter 6

Conclusion

Instruction scheduling is important for improving the efficiency of code in the presence of instruction level parallelism and pipelining. The increase of hardware complexity brings an increase in the time needed to design, test and revise heuristics used to schedule the code executed on the architectures. Automated techniques have been proposed for generating instruction scheduling heuristics and we show that these techniques can be successfully applied to the problems of basic block and super block scheduling. The work on basic block scheduling extends the work of Moss et al. [43, 8] by using a more robust data set including large basic blocks and by increasing the size of the features set. We applied automated techniques, for the first time, to the problem of learning super block scheduling heuristics. The larger feature set allowed us to apply machine learning techniques to identify useful and irrelevant features prior to learning a heuristic. The feature set was derived from features found in the literature and expanded to include many novel features that were found to be useful for instruction scheduling. Two of these novel features, maximum distance and maximum distance and speculative yield, became the primary features of the basic block and super block scheduling heuristics, respectively.

In terms of results, we found that both heuristics reduced the number of non-optimal blocks over the next best heuristic. For basic blocks, we reduced the number of non-optimal blocks by at least 30% and, for super blocks, we reduced the number of non-optimal blocks by at least 16%. For certain architectures, we reduced the number of non-optimal blocks by 55% for basic blocks and 38% for super blocks. We also found that when comparing to standard scheduling heuristics we obtained at least 2.9 and 2.6 times more schedules with reduced cost for basic blocks and super blocks, respectively. Again for certain architectures, we find that we can obtain 7.8 and 4.4 times more schedules with reduced cost for basic blocks and super blocks,

respectively. We also found that the basic block and super block heuristics were never worse and often better in terms of worst case behaviour.

In the future, this model for developing scheduling heuristics could be expanded to remove the assumptions that we made in the model. Specifically, we could remove the assumption of fully pipelined functional units to include instructions that lock a functional unit for more than a single time cycle. Also, we need to consider the effects of registers and register pressure within the scheduling context. The inclusion of registers changes not only the heuristics that are being used but the optimal solution to the problem. With these modifications, this heuristic could be implemented within a production compiler. This technique can also be extended into any scheduling domain where an inefficient optimal scheduler exists to improve heuristics. Job shop scheduling and time tabling are two problems that could benefit from this technique.

This thesis shows that automated techniques for learning instruction scheduling heuristics, both for basic blocks and super blocks, can be successful. We vastly increased the number of features considered by our automated technique and we used filtering, selection and ranking to determine useful features. The features that we synthesized had a significant effect within the scheduling domain. The features used in instruction scheduling can be easily implemented and analyzed in terms of data collected about the scheduling problem. Most importantly, it shows that, if given the appropriate features, the automatically generated heuristics can outperform the techniques generated by hand.

Appendix A

Basic Block Features

This appendix contains the tables for all of the basic block features that were considered before filtering and feature selection was applied. The three tables of features are separated into DAG related features, ready list features and instruction level features.

Table A.1: The set of features related to DAG

1. Number of integer instructions
2. Number of floating point instructions
3. Number of Load/Store Instructions
4. Number of branch instructions
5. Total number of instructions
6. Number of edges between instructions of different types
7. Number of edges between instructions of similar types
8. Maximum critical path distance
9. Average critical path distance
10. Standard deviation of critical path distances
11. Maximum ILP
12. Average ILP
13. Standard deviation of ILPs
14. Maximum latency for any instruction
15. Average latency of instructions
16. Initial number of excess integer instructions given the possible number of instructions that could be scheduled given the maximum critical path and the issue width
17. Initial number of excess floating point instructions given the possible number of instructions that could be scheduled given the maximum critical path and the issue width
18. Initial number of excess load/store instructions given the possible number of instructions that could be scheduled given the maximum critical path and the issue width
19. Initial number of excess branch instructions given the possible number of instructions that could be scheduled given the maximum critical path and the issue width
20. Feature 16 divided by the issue width
21. Feature 17 divided by the issue width
22. Feature 18 divided by the issue width
23. Feature 19 divided by the issue width

Table A.2: The set of features related to the ready list

1. Number of unscheduled integer instructions
2. Number of unscheduled floating point instructions
3. Number of unscheduled load/store instructions
4. Number of unscheduled branch instructions
5. Total number of unscheduled instructions
6. Current time
7. Number of integer instructions ready to be scheduled
8. Number of floating point instructions ready to be scheduled
9. Number of load/store instructions ready to be scheduled
10. Number of branch instructions ready to be scheduled
11. Number of integer instructions that could still be scheduled at the current time
12. Number of floating point instructions that could still be scheduled at the current time
13. Number of load/store instructions that could still be scheduled at the current time
14. Number of branch instructions that could still be scheduled at the current time
15. Number of excess integer instructions given the possible number of instructions that could be scheduled given the maximum critical path and the issue width
16. Number of excess floating point instructions given the possible number of instructions that could be scheduled given the maximum critical path and the issue width
17. Number of excess load/store instructions given the possible number of instructions that could be scheduled given the maximum critical path and the issue width
18. Number of excess branch instructions given the possible number of instructions that could be scheduled given the maximum critical path and the issue width
19. Feature 15 divided by issue width
20. Feature 16 divided by issue width
21. Feature 17 divided by issue width
22. Feature 18 divided by issue width

Table A.3: The set of instruction level basic block features

1. Order of instructions in the original instruction stream
2. Number of immediate successors of the instruction
3. Number of descendants of the instruction
4. Resource-based distance to leaf node
5. Sum of latencies to all immediate successors of the latencies
6. Critical-path distance to leaf node
7. Critical-path distance from root
8. Path length to leaf node
9. Path length from root
10. Slack–the difference between earliest and latest start time
11. Earliest start time updated by current time and partial schedule
12. Number of instructions of type integer that would be added to the ready list for the current time cycle if the instruction was scheduled
13. Number of instructions of type floating point that would be added to the ready list for the current time cycle if the instruction was scheduled
14. Number of instructions of type load/store that would be added to the ready list for the current time cycle if the instruction was scheduled
15. Number of instructions of type branch that would be added to the ready list for the current time cycle if the instruction was scheduled
16. Number of instructions of type integer that would be added to the ready list for the next time cycle if the instruction was scheduled
17. Number of instructions of type floating point that would be added to the ready list for the next time cycle if the instruction was scheduled
18. Number of instructions of type load/store that would be added to the ready list for the next time cycle if the instruction was scheduled
19. Number of instructions of type branch that would be added to the ready list for the next time cycle if the instruction was scheduled
20. Maximum of feature 4 and feature 6

Appendix B

Super Block Features

This appendix contains the remainder of the super block features along with the explanations of any of the new features. The remainder of the super block features can be viewed in Table B.1.

The maximum (or minimum) possible cost for an instruction i finds what percentage of the maximum cost that the weighted maximum distance to branch accounts for within the DAG. This feature can be formalized as,

$$max\_cost(i) = \frac{max\_wdb(i)}{max\_db(i) \sum_{b \in B} w(b)}. \tag{B.1}$$

Instead of finding the ratio of weighted maximum to total maximum, for the next feature, we reverse the ratio and find what percentage the weighted minimum is of the total maximum cost and vice versa. This allows us to determine the likelihood that the remaining branches are closer to the maximum distance to branch or minimum distance to branch. These features can be described formally as,

$$max\_ratio(i) = \frac{min\_wdb(i)}{max\_db(i) \sum_{b \in B} w(b)}. \tag{B.2}$$

The sink to root ratio is a simple ratio of the distance to the sink node of the DAG and the root node of the DAG. This ratio gives an estimate of the position of the node within the graph. This feature can be formally stated as,

$$sink/root = cp(i, n)/cp(1, i). \tag{B.3}$$

The projected slot is a simple measurement of the number of slots in the graph and the slot of the instruction based on the original order. This feature can be formalized as,

$$pslot = order(i)/num\_instructions, \tag{B.4}$$

where $order(i)$ is the original position of the instruction generated by the code generator and $num\_instructions$ is the number of instructions in the graph.

The projected schedule is a dynamic measurement of how far the original maximum distance has been delayed by resource constraints. This feature can be denoted as,

$$psched(i) = maxDelay(currentTime - cp(1, i)). \tag{B.5}$$

The projected schedule and projected slot of an instruction can be used to identify a possible release time for the instruction. The initial estimate of release combined with the newly recalculated schedule length provides a simple method of calculating a new release time. The estimated release time can be formally defined as,

$$ert(i) = pslot(i)psched(i). \tag{B.6}$$

Table B.1: Features Within the Domain of Super block Scheduling

26. Helped weight—sum of weights of each helped branch
27. Helped count—count of helped branches
28. Minimum of critical-path distances to each branch
29. Weight—sum of the exit probabilities of every descendant branch
30. Estimated release time (see Equation B.6)
31. Order of instruction in the original instruction stream
32. Projected slot (see Equation B.4)
33. Ratio of minimum distance to maximum possible cost (see Equation B.2)
34. Probability of minimum possible cost (see Equation B.1)
35. Number of immediate successors of the instructions
36. Minimum latest start time with respect to any branch
37. Execution time of the instruction
38. Sum of the latencies to all immediate successors of the instruction
39. Sink to root ratio (see Equation B.3)
40. Ratio of maximum distance to minimum possible cost (see Equation B.2)
41. Critical-path distance from the root
42. Earliest start time of the instruction
43. Projected schedule (see Equation B.5)
44. Updated earliest start time
45. Probability of maximum possible cost (see Equation B.1)
46. Path length from the root
47. Number of instructions of type branch that would be added to the ready list for the current time cycle if the instruction was scheduled
48. Feature 47 minus the number of slots currently available for scheduling branch instructions
49. Number of instructions of type integer that would be added to the ready list for the current time cycle if the instruction was scheduled
50. Feature 49 minus the number of slots currently available for scheduling integer instructions
51. Number of instructions of type load/store that would be added to the ready list for the current time cycle if the instruction was scheduled

Table B.2: Features Within the Domain of Super block Scheduling

52. Feature 51 minus the number of slots currently available for scheduling load/store instructions
53. Number of instructions of type integer that would be added to the ready list for the next time cycle if the instruction was scheduled
54. Number of instructions of type load/store that would be added to the ready list for the next time cycle if the instruction was scheduled
55. Rank—the earliest start time of the instruction divided by the exit probability of the instruction. Only defined for branches
56. Number of instructions of type floating point that would be added to the ready list for the current time cycle if the instruction was scheduled
57. Feature 56 minus the number of slots currently available for scheduling floating point instructions
58. Type of instruction
59. Number of instructions of type floating point that would be added to the ready list for the next time cycle if the instruction was scheduled
60. Number of instructions of type branch that would be added to the ready list for the next time cycle if the instruction was scheduled

Bibliography

[1] S. G. Abraham, W. M. Meleis, and I. D. Baev. Efficient backtracking instruction schedulers. In *Proceedings of the 2000 International Conference on Parallel Architectures and Compilation Techniques (PACT '00)*, pages 301–308. IEEE Computer Society, 2000.

[2] S. Beaty, S. Colcord, and P. Sweany. Using genetic algorithms to fine-tune instruction scheduling heuristics. In *Proceedings of the IEEE International Conference on Massively Parallel Computing Systems (MPCS '96)*, 1996.

[3] D. Bernstein and M. Rodeh. Global instruction scheduling for superscalar machines. In *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation (PLDI '91)*, pages 241–255. ACM Press, 1991.

[4] R. J. Blainey. Instruction scheduling in the TOBEY compiler. *IBM J. Res. Develop.*, 38(5):577–593, 1994.

[5] A. L. Blum and P. Langley. Selection of relevant features and examples in machine learning. *Artificial Intelligence*, 97(1-2):245–271, 1997.

[6] R. A. Bringmann. *Enhancing Instruction Level Parallelism through Compiler-Controlled Speculation.* PhD thesis, University of Illinois at Urbana-Champaign, 1995.

[7] B. Calder, D. Grunwald, M. Jones, D. Lindsay, J. Martin, M. Mozer, and B. Zorn. Evidence-based static branch prediction using machine learning. *ACM Transactions on Programming Language Systems*, 19(1):188–222, 1997.

[8] J. Cavazos. *Automatically Constructing Compiler Optimization Heuristics Using Supervised Learning.* PhD thesis, University of Massachusetts Amherst, 2004.

[9] C. Chekuri, R. Johnson, R. Motwani, B. Natarajan, B. R. Rau, and M. Schlansker. Profile-driven instruction level parallel scheduling with application to super blocks. In *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture (MICRO 29)*, pages 58–67. IEEE Computer Society, 1996.

[10] K. D. Cooper and L. Torczon. *Engineering a Compiler*. Morgan Kaufmann, 2004.

[11] SPEC Corporation. SPEC 2000 cpu benchmarks, 2000.

[12] R.C. Correa, A. Ferreira, and P. Rebreyend. Scheduling multiprocessor tasks with genetic algorithms. *IEEE Transactions on Parallel and Distributed Systems*, 10(8):825–837, 1999.

[13] B. L. Deitrich and W. W. Hwu. Speculative hedge: Regulating compile-time speculation against profile variations. In *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture (MICRO 29)*, pages 70–79. IEEE Computer Society, 1996.

[14] A. E. Eichenberger and W. M. Meleis. Balance scheduling: Weighting branch tradeoffs in superblocks. In *Proceedings of the 32nd Annual ACM/IEEE International Symposium on Microarchitecture (MICRO 32)*, pages 272–283. IEEE Computer Society, 1999.

[15] P. Faraboschi, J.A. Fisher, and C. Young. Instruction scheduling for instruction level parallel processors. *Proceedings of the IEEE*, 89(11):1638–1659, 2001.

[16] J. A. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Transactions on Computers*, c-30(7):478–490, 1981.

[17] J. A. Fisher. Global code generation for instruction-level parallelism: Trace scheduling-2. Technical Report HPL-93-43, Hewlett Packard, July 1993.

[18] R. Govindarajan. Instruction scheduling. In Y. N. Srikant and P. Shankar, editors, *The Compiler Design Handbook*, pages 631–687. CRC Press, 2003.

[19] I. Guyon and A. Elisseeff. An introduction to variable and feature selection. *Journal of Machine Learning Research*, 3:1157–1182, 2003.

[20] W. Havanki, S. Banerjia, and T. Conte. Treegion scheduling for wide issue processors. In *Proceedings of the 4th International Symposium on High-Performance Computer Architecture (HPCA '98)*, pages 266–276. IEEE Computer Society, 1998.

[21] J. L. Hennessy and T. R. Gross. Code generation and reorganization in the presence of pipeline constraints. In *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '82)*, pages 120–127. ACM Press, 1982.

[22] S. Hoxey, F. Karim, B. Hay, and H. Warren. *The PowerPC Compiler Writer's Guide*. Warthman Associates, 1996.

[23] T. C. Hu. Parallel sequencing and assembly line problems. *Operations Research*, 9(6):841–848, 1961.

[24] W. W. Hwu, S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Warter, R. A. Bringmann, R. G. Ouellette, R. E. Hank, T. Kiyohara, G. E. Haab, J. G. Holm, and D. M. Lavery. The superblock: An effective technique for VLIW and superscalar compilation. *Journal of Supercomputing*, 7(1-2):229–248, 1993.

[25] Rocket Software Inc. The rocket compiler, 2006.

[26] D. Jimenez and C. Lin. Perceptron learning for predicting the behavior of conditional branches. In *Proceedings of the International Joint Conference on Neural Networks (IJCNN01)*, pages 2122–2126, 2001.

[27] H. S. Warren Jr. Instruction scheduling for the IBM RISC system/6000 processor. *IBM Journal of Research and Development*, 34(1):85–92, 1990.

[28] R. Kohavi and F. Provost. Glossary of terms. *Machine Learning*, 30:271–274, 1998.

[29] S. M. Krishamurthy. A brief survey of papers on scheduling for pipelined processors. *SIGPLAN Notices*, 25(7):97–106, 1990.

[30] M. Langevin and E. Cerny. A recursive technique for computing lower-bound performance of schedules. *ACM Transactions on Design Automation of Electronic Systems*, 1(4):443–455, 1996.

[31] C. Lee, M. Potkonjak, and W. H. Mangione-Smith. Mediabench: a tool for evaluating and synthesizing multimedia and communicatons systems. In *Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture (MICRO 30)*, pages 330–335. IEEE Computer Society, 1997.

[32] C.-Y. Lee, S. Piramuthu, and Y.-K. Tsai. Job shop scheduling with a genetic algorithm and machine learning. *International Journal of Production Research*, 35(4):1171–1191, 1997.

[33] X. Li and S. Olafsson. Discovering dispatching rules using data mining. *Journal of Scheduling*, 8:515–527, 2005.

[34] S. Long and M. O'Boyle. Adaptive java optimisation using instance-based learning. In *Proceedings of the 18th Annual International Conference on Supercomputing (ICS '04)*, pages 237–246. ACM Press, 2004.

[35] S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, and R. A. Bringmann. Effective compiler support for predicated execution using the hyperblock. In *Proceedings of the 25th Annual International Symposium on Microarchitecture (MICRO 25)*, pages 45–54. IEEE Computer Society Press, 1992.

[36] A. Malik, M. Chase, T. Russell, and P. van Beek. Optimal superblock instruction scheduling for multi-issue processors using constraint programming. Technical report, School of Computer Science, University of Waterloo, 2006.

[37] A. M. Malik, J. McInnes, and P. van Beek. Optimal basic block instruction scheduling for multiple-issue processors using constraint programming. Technical Report CS-2005-19, School of Computer Science, University of Waterloo, 2005.

[38] A. McGovern, J. E. B. Moss, and A. G. Barto. Building a basic block instruction scheduler using reinforcement learning and rollouts. *Machine Learning*, 49(2/3):141–160, 2002.

[39] W. M. Meleis, A. E. Eichenberger, and I. D. Baev. Scheduling superblocks with bound-based branch trade-offs. *IEEE Transactions on Computers*, 50(8):784–797, 2001.

[40] G. De Micheli. *Synthesis and Optimization of Digital Circuits*. McGraw Hill, 1994.

[41] T. M. Mitchell. *Machine Learning*. McGraw Hill, 1997.

[42] A. Monsifrot, F. Bodin, and R. Quiniou. A machine learning approach to automatic production of compiler heuristics. In *Proceedings of the 10th International Conference on Artificial Intelligence: Methodology, Systems, and Applications (AIMSA '02)*, pages 41–50. Springer-Verlag, 2002.

[43] J. E. B. Moss, P. E. Utgoff, J. Cavazos, D. Precup, D. Stefanovic, C. E. Brodley, and D. T. Scheeff. Learning to schedule straight-line code. In *Proceedings of the 10th Conference on Advances in Neural Information Processing Systems (NIPS)*, pages 929–935, 1997.

[44] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers Inc., 1997.

[45] J. R. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann Publishers Inc., 1993.

[46] T. Russell, A. M. Malik, M. Chase, and P. van Beek. Learning basic block scheduling heuristics from optimal data. In *Proceedings of the 15th CASCON*, Toronto, 2005.

[47] M. Schlansker. Compilation for VLIW and superscalar processors. In *ASPLOS-IV Tutorial*, pages 1–74, April 1991.

[48] J.-J. Shieh and C. Papachristou. On reordering instruction streams for pipelined computers. In *Proceedings of the 22nd Annual Workshop on Microprogramming and Microarchitecture (MICRO 22)*, pages 199–206. ACM Press, 1989.

[49] J.-J. Shieh and C. A. Papachristou. An instruction reoderer for pipelined computers. In *Proceedings of the 23rd Annual Workshop and Symposium on Microprogramming and Microarchitecture (MICRO 23)*, pages 135–142. IEEE Computer Society Press, 1990.

[50] G. Shobaki and K. Wilken. Optimal superblock scheduling using enumeration. In *Proceedings of the 37th Annual International Symposium on Microarchitecture (MICRO 37)*, pages 283–293. IEEE Computer Society, 2004.

[51] R. Sites. Alpha architecture reference manual. Technical report, Digital Equip. Corp., 1992.

[52] M. Smotherman, S. Krishnamurthy, P. S. Aravind, and D. Hunnicutt. Efficient dag construction and heuristic calculation for instruction scheduling. In *Proceedings of the 24th Annual International Symposium on Microarchitecture (MICRO 24)*, pages 93–102. ACM Press, 1991.

[53] M. Stephenson, S. Amarasinghe, M. Martin, and U.-M. O'Reilly. Genetic programming applied to compiler heuristic optimisation. In *The 6th European Conference on Genetic Programming*, 2003.

[54] M. Stephenson, S. Amarasinghe, M Martin, and U.-M. O'Reilly. Meta optimization: Improving compiler heuristics with machine learning. *SIGPLAN Notices*, 38(5):77–90, 2003.

[55] M.D. Tiemann. The GNU instruction scheduler. Cs343 course report, Stanford University, Jun. 1989.

[56] P. van Beek and K. Wilken. Fast optimal instruction scheduling for single-issue processors with arbitrary latencies. In *Proceedings of the 7th International Conference on Principles and Practice of Constraint Programming (CP '01)*, pages 625–639, 2001.

[57] S. Weiss and J. E. Smith. A study of scalar compilation techniques for pipelined supercomputers. In *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 105–109, 1987.

[58] K. Wilken, J. Liu, and M. Heffernan. Optimal instruction scheduling using integer programming. In *Proceedings of the SIGPLAN 2000 Conference on Programming Language Design and Implementation (PLDI)*, pages 121–133, Vancouver, 2000.

[59] I.H. Witten and E. Frank. *Data Mining.* Morgan Kaufmann, 2000.