Learning Heuristics for Basic Block Instruction Scheduling

Abid M. Malik, Tyrel Russell, Michael Chase, and Peter van Beek
School of Computer Science
University of Waterloo, Waterloo, Canada

Abstract

Instruction scheduling is an important step for improving the performance of object code produced by a compiler. A fundamental problem that arises in instruction scheduling is to find a minimum length schedule for a basic block—a straight-line sequence of code with a single entry point and a single exit point—subject to precedence, latency, and resource constraints. Solving the problem exactly is known to be difficult, and most compilers use a greedy list scheduling algorithm coupled with a heuristic. The heuristic is usually hand-crafted, a potentially time-consuming process. In contrast, we present a study on automatically learning good heuristics using techniques from machine learning. In our study, a recently proposed optimal basic block scheduler was used to generate the machine learning training data. A decision tree learning algorithm was then used to induce a simple heuristic from the training data. 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.

1 Introduction

Modern computer architectures are pipelined and can issue multiple instructions per time cycle. On such processors, the order that the instructions are scheduled can significantly impact performance. The basic block instruction scheduling problem is to find a minimum length schedule for a basic block—a straight-line sequence of code with a single entry point and a single exit point—subject to precedence, latency, and resource constraints[1]. Basic block scheduling is important in its own right and also as a building block for scheduling larger groups of instructions such as superblocks [2, 18].

Solving the basic block instruction scheduling problem exactly is known to be difficult, and most compilers use a greedy list scheduling algorithm together with a heuristic for choosing which

[1] See Section 2 for the necessary background on computer architectures and basic block instruction scheduling.

instruction to schedule next [4, 13]. Such a heuristic usually consists of a set of features and a priority or order in which to test the features. Many possible features and orderings have been proposed (see, for example, [1, 19]). The heuristic in a production compiler is usually hand-crafted by choosing and testing many different subsets of features and different possible orderings—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" [7, p. 112, emphasis added].

In this paper, we present a study on automatically learning a good heuristic using supervised machine learning techniques. In supervised learning, one learns from training examples which are labeled with the correct answers. More precisely, each training example consists of a vector of feature values and the correct classification or correct answer for that example. The success of a supervised learning approach depends heavily on the quality of the training data; i.e., if the examples are representative of what will be seen in practice and if the features recorded in each example are adequate to distinguish all of the different cases. Moss et al. [12] were the first to propose the use of supervised learning techniques in this context. Their idea was to design an optimal schedule and use their optimal scheduler to correctly label the data. However, their approach was hampered by the quality of their training data; their optimal scheduler could only optimally solve basic blocks with ten or fewer instructions and they recorded only five features in each training example[2].

McGovern et al. [11] used the same set of features as Moss et al., but proposed the use of rollouts and reinforcement learning to overcome the difficulty of obtaining training data on larger basic blocks. Rollouts and reinforcement learning are machine learning techniques which are non-supervised; i.e., they do not require the data to be correctly labeled. However, the efficiency of the instruction scheduler is critical in compilers, and McGovern et al.'s scheduler based on rollouts would be much too expensive to be used in a production compiler. Further, in McGovern et al.'s work using reinforcement learning, a known heuristic was used to guide the reinforcement learning. However, the resulting learned instruction scheduler was not better than the original heuristic used to guide the learning. Li and Olafsson [9], in recent work on learning heuristics for single machine job shop scheduling using supervised learning, use existing heuristics to label the training examples. In other words, in their approach some of the training examples will be labeled incorrectly. As a result, the heuristics that they learn are never better than the original heuristics used to label the data. In contrast to McGovern et al. and to Li and Olafsson, in our work we are able to learn heuristics that outperform existing heuristics.

In our work, we improved the quality of the training data in three ways. First, we overcame the limitation on basic block size by designing and using an optimal basic block scheduler based on constraint programming [10, 21] to generate the correctly labeled training data. Currently, this optimal scheduler is too time consuming to be used on a routine basis[3]. However, it can solve

[2]The heuristic that would be learned from their training data would be similar to the critical-path based heuristic, which we compare against below and in Section 4.

[3]Large software projects can contain tens of thousands of basic blocks and users demand fast compile times during the development phase of a project. Thus, the instruction scheduling component of the compiler must execute quickly during this phase. However, slower compile times may be acceptable during a production build and the constraint programming approach may be viable during this phase of a project.

all but a very few basic blocks and routinely solves blocks with 2500 or more instructions in a few minutes. (The speed of the optimal scheduler is not an issue when gathering training data, as this is an offline process.) Second, we improved the quality of the training data by performing an extensive and systematic study and ranking of previously proposed features (as surveyed in [19]). Third, we improved the quality of the training data by synthesizing and ranking novel features. One of these novel features is the best feature among all of the features that we studied, and is one of the major reasons behind the success of our approach.

Once the training data was gathered, a decision tree learning algorithm [16] was used to induce a heuristic from the training data. In a decision tree the internal nodes of the tree are labeled with features, the edges to the children of a node are labeled with the possible values of the feature, and the leaves of the tree are labeled with a classification. To classify a new example, one starts at the root and repeatedly tests the feature at a node and follows the appropriate branch until a leaf is reached. The label of the leaf is the predicted classification of the new example.

The usual criterion one wants to maximize when devising a heuristic is accuracy. In this study, we also had an additional criterion: that the learned heuristic be efficient. When compiling large software projects, the heuristic used by the list scheduler can be called hundreds of thousands or even millions of times and can represent a significant percentage of the overall compilation time. Since each additional feature used in a heuristic adds to the overall computation time, we want to learn a heuristic that is both simple and accurate. Fortunately, these are not necessarily conflicting goals since it is known that more complex decision trees often "overfit" the training data, and that simpler decision trees often generalize better and so perform better in practice [23]. In contrast to Cooper and Torczon [2], who note that no set of features and no order in which to test the features dominates the others, we found that a small set of features and orderings did dominate and that many features were irrelevant in that they did not improve the accuracy of a heuristic in a statistically significant way.

Once learned, the resulting decision tree heuristic was incorporated into both a forward list scheduler and a backward list scheduler and experimentally compared against a popular critical-path heuristic on the SPEC 2000 benchmarks, using four different architectural models. 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. As well, for every basic block where the critical-path heuristic found a better schedule than the decision tree heuristic, there were up to eight basic blocks where the decision tree heuristic found a better schedule than the critical-path heuristic. Finally, the decision tree heuristic improved performance guarantees in terms of the worst-case factor from optimality, a measure of the robustness of a heuristic.

2 Background

In this section, we define the instruction scheduling problem studied in this paper followed by a brief review of the list scheduling algorithm and the heuristics used in the algorithm (for more background on these topics see, for example, [4, 6, 13]).

We consider multiple-issue, pipelined processors. Multiple-issue and pipelining are two techniques for performing instructions in parallel and processors which use these techniques are now

standard in desktop and laptop machines. In such processors, there are multiple functional units and multiple instructions can be issued (begin execution) in each clock cycle. Examples of functional units include arithmetic-logic units (ALUs), floating-point units, memory or load/store units which perform address computations and accesses to the memory hierarchy, and branch units which execute branch and call instructions. The number of instructions that can be issued in each clock cycle is called the *issue width* of the processor. As well, in such processors functional units are pipelined. Pipelining is a standard implementation technique for overlapping the execution of instructions on a single functional unit. A helpful analogy is to a vehicle assembly line [6] where there are many steps to constructing the vehicle and each step operates in parallel with the other steps. An instruction is issued (begins execution) on a functional unit and associated with each instruction is a delay or *latency* between when the instruction is issued and when the instruction has completed and the result is available for other instructions which use the result. In this paper, we assume that all functional units are fully pipelined and that instructions are typed and execute on a functional unit of that type. Examples of types of instructions are integer, floating point, load/store, and branch instructions.

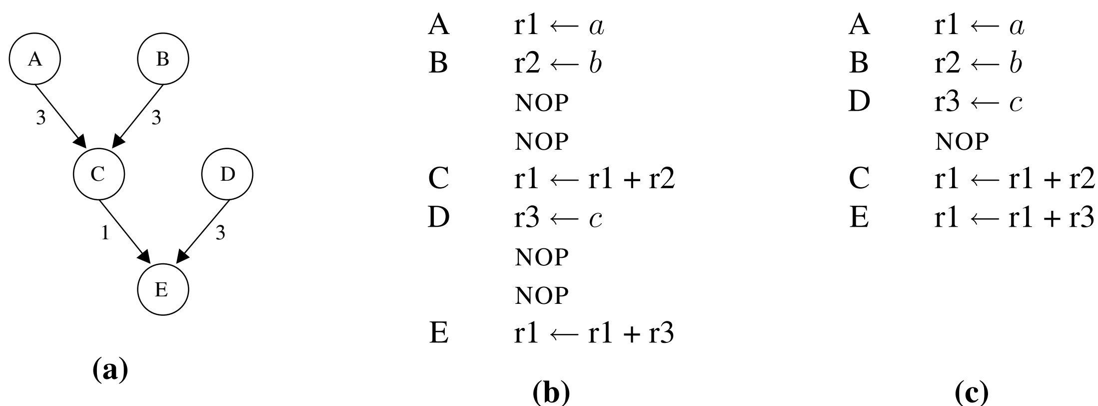


Figure 1: (a) Dependency DAG associated with the instructions to evaluate $(a+b)+c$ on a processor where loads from memory have a latency of 3 cycles and integer operations have a latency of 1 cycle; (b) a schedule; (c) a better schedule.

We use the standard labeled directed acyclic graph (DAG) representation of a basic block (see Figure 1(a)). Each node corresponds to an instruction and there is an edge from i to j labeled with a positive integer $l(i,j)$ if j must not be issued until i has executed for $l(i,j)$ cycles. The goal is to schedule the basic block on a target processor. The target processor will have a particular set of resources: the number and type of functional units available in the processor for executing instructions.

Given a labeled dependency DAG for a basic block and a target processor, a *schedule* for the processor specifies an issue or start time for each instruction or node such that the latency constraints are satisfied and the resource constraints are satisfied. The resource constraints ensure that the limits of the processor's resources are never exceeded; i.e., the resource constraints are satisfied if, at every time cycle, the number of instructions of each type issued at that cycle does

not exceed the number of functional units that can execute instructions of that type. The *length* of a schedule is the number of cycles needed for the schedule to complete; i.e., each instruction has been issued at its start time and, for each instruction with no successors, enough cycles have elapsed that the result for the instruction is available. The *basic block instruction scheduling problem* is to construct a schedule with minimum length.

Example 1 *Figure 1 shows a simple dependency DAG and two possible schedules for the DAG, assuming a single-issue processor that can execute all types of instructions. The first schedule (b) requires four* NOP *instructions (null operations) because the values loaded are used by the following instructions. The better schedule (c), the optimal or minimum length schedule, requires only one* NOP *and completes in three fewer cycles.*

Basic block instruction scheduling under the assumption that all functional units are fully pipelined is the special case of resource-constrained project scheduling (see, e.g., [3, 14]) where all of the activities have unit execution times and we seek a schedule which minimizes the makespan.

Instruction scheduling for basic blocks is known to be NP-complete for realistic architectures. The most popular method for scheduling basic blocks continues to be list scheduling [13]. A list scheduler takes a set of instructions as represented by a dependency DAG and builds a schedule using a best-first greedy heuristic. A list scheduler generates the schedule by determining all instructions that can be scheduled at that time step, called the ready list, and uses the heuristic to determine the best instruction on the list. The selected instruction is then added to the partial schedule and the scheduler determines if any new instructions can be added to the ready list. List schedulers work in one of two directions: forward from the roots to the leaves or backward from the leaves to the roots (see [2] for a discussion of forward versus backward list scheduling). Conceptually, a backward list scheduler can be obtained by running a forward list scheduler on a dependency DAG that has had the direction of each edge reversed. Both forward and backward list scheduling are used in production compilers. As examples, IBM's Tobey compiler uses forward list scheduling [1] and the GCC compiler uses backward list scheduling [20].

The heuristic in a list scheduler generally consists of a set of features and an order for testing the features. Some standard features are as follows. The *path length* from a node i to a node j in a DAG is the maximum number of edges along any path from i to j. The *critical-path distance* from a node i to a node j in a DAG is the maximum sum of the latencies along any path from i to j. Note that both the path length and the critical-path distance from a node i to itself is zero. A node j is a *descendant* of a node i if there is a directed path from i to j; if the path consists of a single edge, j is also called an *immediate successor* of i. The *earliest start time* of a node i is a lower bound on the earliest cycle in which the instruction i can be scheduled. The heuristic in a list scheduler is also known as a dispatching rule in the scheduling literature (see, e.g., [15]).

3 Learning a Heuristic

In this section, we describe the methodology we followed to automatically construct a list scheduling heuristic for scheduling basic blocks by applying techniques from supervised machine learning.

We explain the construction of the initial set of features (Section 3.1), the collection of the data (Section 3.2), the use of the data to filter and rank the features to find the most important features (Section 3.3), and the use of the data and the important features to learn a simple heuristic (Section 3.4).

3.1 Feature construction

To use supervised learning techniques to construct a list scheduling heuristic, the problem first has to be phrased as a classification problem. Moss et al. [12] note that to choose the best instruction on a ready list to schedule next, it is sufficient to be able to compare two instructions i and j and return true if i should be scheduled before j; and false otherwise. We thus have a binary classification problem.

The choice of distinguishing features is critical to successfully learning a classifier. In contrast to Moss et al. [12], who recorded only five features in each training example, we began with almost 100 features that we felt would be promising. The features can be categorized as either static or dynamic. The value of a static feature is determined before the execution of the list scheduler; the value of a dynamic feature is determined during the execution of the list scheduler. The 100 features included all of the classic features surveyed by Smotherman et al. [19], except for the features having to do with register pressure[4] and a few features that were irrelevant or whose roles were better performed by other features. These classic features all measure properties of an instruction on the ready list (see Table 2 for a summary of most of these features).

Also included were features that measured properties of the DAG to be scheduled, properties of the target architecture, and properties of a ready list itself. The DAG features were both static and dynamic. The static features included the source language and the number of instructions of each type. The dynamic features included the number of instructions of each type that still needed to be scheduled; the number of edges between instructions of the same type and of different types in the unscheduled part of the DAG; and the maximum, average, and standard deviation of the critical-path distances and the latencies on the edges in the unscheduled part of the DAG. The target architecture features were all static features. Examples of these features include the number of functional units of each type. The ready list features were all dynamic features. Examples of these features include the number of instructions of each type on the ready list and the maximum number of instructions of each type that could still be scheduled during the current time cycle.

Also included were features that measured properties of an instruction relative to all of the other instructions on a ready list. For these features, given an instruction i, a property of an instruction such as critical-path distance, and a ready list, we determine the property's maximum and minimum value over all of the instructions on the ready list. The value v of the property for instruction i is then compared against these maximum and minimum values and the value of the feature is gt if v is closer to the maximum, eq if it is midway between the maximum and the minimum, and lt if it is closer to the minimum.

[4] These features were omitted as our optimal basic block scheduler [10, 21], which was used to correctly label the data, does not currently handle register pressure. We are currently working on removing this limitation and once it is removed, it will be straightforward to incorporate these features as well.

A more accurate classifier can sometimes be achieved by synthesizing new features from existing basic features. We also included many such synthesized features. In our case, the feature synthesis was performed by hand. However, this phase only has to be performed once and then these new features can be used in the future for constructing heuristics for compilers targeted to new computer architectures. We constructed some of the novel features by applying simple functions to basic features. Examples include comparison of two features, maximum of two features, and the average of several features.

Table 1: Notation for the resource-based distance to leaf node feature.

k_t	The number of functional units that can execute instructions of type t.
$desc(i, t)$	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., $\mid desc(i, t) \mid / \; k_t$, 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 constructed, resource-based distance to the leaf node, turned out to be the best feature among all of the features that we studied. Consider the notation shown in Table 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 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. 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 $r_1(i, t) + r_3(i, t)$.

3.2 Collecting the training, validation, and testing data

In addition to the choice of distinguishing features (see Section 3.1 above), a second critical factor in the success of a supervised learning approach is whether the data is representative of what will be seen in practice. To adequately train and test our heuristic classifier, we collected all of the basic blocks in the jpeg and mpeg benchmarks from the MediaBench [8] benchmark suite and all of the basic blocks from the SPEC 2000 integer and floating point benchmarks [http://www.specbench.org]. The SPEC benchmarks are standard benchmarks used to evaluate new CPUs and compiler optimizations. The benchmarks were compiled using IBM's Tobey compiler [1] targeted towards the PowerPC processor [7], and the basic blocks were captured as they were passed to Tobey's instruction scheduler. The basic blocks contain four types of instructions: branch, load/store, integer, and floating point. 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). The Tobey compiler performs instruction scheduling before global register allocation and once again afterwards, and our test suite contains both kinds of basic blocks. The compilations were done using Tobey's highest level of optimization, which includes aggressive optimization techniques such as software pipelining and loop unrolling.

Following Moss et al. [12], a forward list scheduling algorithm was modified to generate the data. Recall that each instance in the data is a vector of feature values and the correct classification for that instance. Let $\text{better}(i, j, class)$ be a vector that is defined as follows,

$$\text{better}(i, j, class) = \langle f_1(i, j), \ldots, f_n(i, j), class \rangle,$$

where i and j are instructions, $f_k(i, j)$ is the kth feature that measures some property of i and j, and *class* is the correct classification. Given a partial schedule and a ready list during the execution of the list scheduler on a basic block, each instruction on the ready list was scheduled by an optimal scheduler [10, 21] to determine the length of an optimal schedule if that instruction were to be selected next. The optimal scheduler was targeted to a 4-issue processor, with one functional unit for each type of instruction. Then, for each pair of instructions i and j on the ready list, where i led to an optimal schedule and j did not, the instances $\text{better}(i, j, \text{true})$ and $\text{better}(j, i, \text{false})$ were added to the data set (see the equation above). Note that the goal of the heuristic that is learned from the data is to distinguish those instructions on a ready list that lead to optimal schedules from those instructions that lead to non-optimal schedules. Thus, pairs of instructions i and j in which both i and j led to an optimal schedule are ignored; i.e., they do not add any instances to the data set. Similarly, pairs of instructions in which both i and j led to a non-optimal schedule are also ignored. Once the data collection process was completed for a particular ready list, the partial schedule was then extended by randomly choosing an instruction from among the instructions on the ready list that led to an optimal schedule, the ready list was updated based on that choice of instruction, and the data collection process was repeated.

Example 2 *Consider once again the DAG and its schedules introduced in Example 1. Suppose that each instance in our learning data contains two features: $f_1(i, j)$ returns the size of the DAG*

and $f_2(i, j)$ returns lt, eq, or gt depending on whether the critical-path distance of i to the leaf node is less than, equal to, or greater than the critical-path distance of j to the leaf node. When the list scheduling algorithm is applied to the DAG, instructions A, B, and D are on the ready list at time cycle 1. Scheduling A first or B first both lead to an optimal schedule, whereas scheduling D first does not. Thus, the pair A, D would add the vectors $\langle 5, gt, \text{true} \rangle$ and $\langle 5, lt, \text{false} \rangle$ to the data, since the critical-path distance for A is 4 and for D is 3. Similarly, the pair B, D would add the vectors $\langle 5, gt, \text{true} \rangle$ and $\langle 5, lt, \text{false} \rangle$. At this point, one of A and B is randomly selected and scheduled at time cycle 1. The list scheduler then advances to time cycle 2, updates the ready list, and repeats the above process.

When lots of data is available, as in our study, a standard approach is to split the data into training, validation, and test sets [23, pp. 120-122]. The training set is used to come up with the classifier, the validation set is used to optimize the parameters of the learning algorithm and to select a particular classifier, and the test set is used to report the classification accuracy. Separating the training and the testing data in this way is important for getting a reliable estimate of how accurately the heuristic will perform in practice. We set aside all of the basic blocks from the SPEC 2000 benchmark for testing, and used the data generated from the jpeg basic blocks for training and the data generated from the mpeg basic blocks for validation. There were approximately 200,000 instances in the training set and 100,000 instances in the validation set. Many of these instances were from large basic blocks of up to 2600 instructions.

3.3 Feature filtering

An important next step, prior to learning the heuristic, is to filter the features. The goal of filtering is to select the most important features for constructing a good heuristic. Only the selected features are then passed to the learning algorithm and the features identified as irrelevant or redundant are deleted. There are two significant motivations for performing this preprocessing step: the efficiency of the learning process can be improved and the quality of the heuristic that is learned can be improved (many learning methods, decision tree learning included, do poorly in the presence of redundant or irrelevant features [23, pp. 231-232]).

Several feature filtering techniques have been developed (see, for example, [5] and the references therein). In our work, a feature was deleted if both: (i) the accuracy of a single feature decision tree classifier constructed from this feature was no better than random guessing on the validation set; and (ii) the accuracy of all two-featured decision tree classifiers constructed from this feature and each of the other features was no better than or a negligible improvement over random guessing on the validation set. The motivation behind case (ii) is that a feature may not improve classification accuracy by itself, but may be useful together with another feature. In both cases, the heuristic classifier was learned from the jpeg training data and evaluated on the mpeg validation set. Finally, a feature was also deleted if it was perfectly correlated with another feature.

Table 2 shows the 17 features that remained after filtering. 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 (see the use of the critical-path feature in Example 2). The features are shown ranked according

Table 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 (see Section 3.1).
3. Path length to leaf node.
4. Number of descendants of the instruction.
5. Critical-path distance to leaf node.
6. Slack—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.

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 [23] 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 [4, 13]), is ranked only in 5th 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.

3.4 Classifier selection

Given the features shown in Table 2, the next step is to learn the best heuristic from the training data; i.e., the best decision tree classifier.

In our context, the best decision tree classifier is one that is both accurate and efficient. Since each additional feature brings additional computational cost, we want the smallest subset of features such that a classifier learned using this subset still has acceptable accuracy. Several methods have been proposed for searching through the possible subsets of features. We chose forward selection with beam search, as it works well to minimize the number of features in the classifier [23, 5]. Forward selection with beam search begins at level 1 by examining all possible ways of constructing a decision tree from one feature. The search then progresses to level 2 by choosing the best of the classifiers from level 1 and extending them in all possible ways by adding one additional feature. In general, the search progresses to level $k + 1$ by extending the best classifiers at

level k by one additional feature. The search continues until some stopping criteria is met.

In our work, the search expanded up to a maximum of 30 of the best classifiers at each level. The value of 30 was chosen as it was found that around this point the quality of the classifiers had already deteriorated. Thus the value of 30 was chosen as a conservative value that avoided a brute-force test of all possible classifiers but with a low risk that we would cutoff and therefore miss a good classifier. For each subset of features at each level, a decision tree heuristic was learned from the jpeg training data and an estimate of the classification accuracy of the heuristic was determined by evaluating the heuristic on the mpeg validation set. The classification accuracy was used to decide which subsets to expand to the next level. We chose decision tree classifiers over other possible machine learning techniques because of their excellent fit with our goals of accuracy and efficiency. To learn a classifier, we used Quinlan's C4.5 decision tree software [16]. The software was run with the default parameter settings, as this consistently gave the best results on the validation set.

The following table shows for each level l (where l corresponds to the number of features in the decision tree), the accuracy of the best decision tree learned with l features and the size of the decision tree. The accuracy is stated as the percentage of instances in the validation set that were *incorrectly* classified. When there were ties for best accuracy at a level, the average size was recorded. The size of the decision tree is 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

We chose four features as the best trade-off between simplicity and accuracy. The decision tree with four features attains an accuracy of 3.72 while containing only 17 nodes. 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 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. In contrast to Moss et al. [12], 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 1.

It is interesting to note that all of the features in the final decision tree heuristic are comparative features with three discrete values; that is, the feature compares some property of two instructions i and j and the value of the feature is lt, eq, or gt depending on whether the value of the property for i is less than, equal to, or greater than the value of the property for j. Besides using three discrete values lt, eq, and gt, we also tried five discrete values: $2gt$, gt, eq, lt, and $2lt$. The idea was to more accurately capture *how* much smaller or bigger one value was than another. For example, if the value of the property for i is much greater than the value of the property for j, the value of the feature would be $2gt$ and if it was only slightly greater, the value of the feature would be just gt. Using five discrete levels, we were able to improve the accuracy to 3.50% with seven features. Unfortunately, it also increased the size of the tree to 250 nodes. As we already

Algorithm 1: Automatically constructed decision tree heuristic for list scheduler.

input : Instructions i and j
output : Return true if i should be scheduled before j; false otherwise

$i.max_distance \leftarrow \max(i.resource_based_dist_to_leaf, i.critical_path_dist_to_leaf)$;
$j.max_distance \leftarrow \max(j.resource_based_dist_to_leaf, j.critical_path_dist_to_leaf)$;
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 $i.latency \geq j.latency$ **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;

mentioned, besides good accuracy we are also interested in a simple heuristic which is easily understandable and efficient. A decision tree with 250 nodes gives a complex heuristic, which is against our aim.

4 Experimental Evaluation

In this section, we describe the experimental evaluation of the decision tree heuristic that was learned following the methodology given above (see Algorithm 1).

The decision tree heuristic was incorporated into a list scheduler and experimentally evaluated on all of the basic blocks from the SPEC 2000 benchmarks, using four different architectural

models:

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.

To begin, the decision tree heuristic (h_{dt}) was compared against previously proposed list scheduling heuristics (h_{est}, h_{cp}, and h_{s+p}) and against the schedules found by the optimal scheduler.

The h_{est} heuristic used updated earliest start time as the primary feature, critical-path distance as a tie-breaker if the updated earliest start times were equal, and order within the instruction stream as a tie-breaker if both the earliest start times and the critical-path distances were equal. The heuristic is similar to a heuristic proposed by Warren [22]. The minor differences are due to differences in architectural models and register pressure features. We refer to this heuristic as the earliest start time heuristic.

The h_{cp} heuristic used critical-path distance as the primary feature, updated earliest start time as a tie-breaker, and order within the instruction stream as the next tie-breaker. The h_{cp} heuristic is perhaps of the most interest in this comparison. This is true for several reasons: critical-path distance is one of the most popular features in the heuristics surveyed by Smotherman et al. [19]; the primary and secondary features are as recommended in a classic text by Muchnick [13]; h_{cp} is similar to the heuristic in the widely used GCC compiler [20]; and finally and most importantly, h_{cp} is the heuristic that would have been learned in the work of Moss et al. [12] when targeted towards our architectural models. We refer to this heuristic as the critical-path heuristic.

The h_{s+p} heuristic used critical-path distance as the primary feature, latency of the instruction as a tie-breaker, number of successors as the next tie-breaker, and order within the instruction stream as the final tie-breaker. The heuristic is similar to a heuristic proposed by Shieh and Papachristou [17]. The minor differences are due to further tie-breaking features in the original proposal. However, we found that these additional features slightly degraded performance. We refer to this heuristic as Shieh and Papachristou's heuristic.

One of the aims of the experiments is to measure the robustness of the decision tree heuristic across architectures and scheduling methods. Recall that the decision tree heuristic was discovered from training data that arose from applying forward list scheduling on a single target architecture (a 4-issue architecture). To measure robustness, the experiments evaluate the heuristics on multiple architectures and using both forward and backward list scheduling.

Table 3 shows the number of basic blocks in the SPEC 2000 benchmark suite that were *not* scheduled optimally by a forward list scheduling algorithm when using the various heuristics. Table 4 shows the results for a backward list scheduling algorithm. The automatically learned decision tree heuristic is *better* than all three of the previously proposed heuristics. To systematically study the scaling behavior of the heuristics, we report the results broken down by increasing size ranges of the basic blocks. For reference, the number of basic blocks in each size range is given in Table 5. For all of the heuristics, as the basic block size increases the accuracy of the list scheduler

decreases. For the largest basic blocks, up to 31% of the schedules are not optimal (see the 2-issue architecture). However, it can be seen that the decision tree heuristic is very effective for small and medium size basic blocks. Tables 3 & 4 show that the decision tree heuristic gave overall reductions in the number of basic blocks not optimally scheduled of between 28% and 55% when compared to the critical-path heuristic. The decision tree heuristic gave similar overall reductions when compared to the h_{s+p} heuristic and significantly greater reductions when compared to the h_{est} heuristic.

Table 3: *Forward list scheduler.* Number of basic blocks in the SPEC 2000 benchmark suite not scheduled optimally by the earliest start time heuristic (h_{est}), Shieh and Papachristou's heuristic (h_{s+p}), 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 percentage improvement given by the decision tree heuristic over the critical-path heuristic ($\% = 100 \times (h_{cp} - h_{dt})/h_{cp}$).

	1-issue					2-issue				
range	h_{est}	h_{s+p}	h_{cp}	h_{dt}	%	h_{est}	h_{s+p}	h_{cp}	h_{dt}	%
1–5	2,613	105	338	0	100.0	2,602	113	350	0	100.0
6–10	15,269	531	804	134	83.3	15,216	637	907	165	81.8
11–20	16,949	1,077	1,118	598	46.5	16,186	1,185	1,226	586	52.2
21–30	6,574	569	619	290	53.2	5,808	761	781	347	55.6
31–50	4,815	640	628	337	46.3	4,419	817	853	512	40.0
51–100	3,053	540	536	315	41.2	2,805	818	790	464	41.3
101–250	1,600	260	270	218	19.3	1,571	485	505	408	19.2
251–2600	264	64	72	68	5.6	273	103	111	111	0.0
Total	51,137	3,786	4,385	1,960	55.3	48,880	4,919	5,523	2,593	53.1

	4-issue					6-issue				
range	h_{est}	h_{s+p}	h_{cp}	h_{dt}	%	h_{est}	h_{s+p}	h_{cp}	h_{dt}	%
1–5	1,045	112	182	12	93.4	0	0	0	0	—
6–10	7,934	565	736	121	83.6	576	90	69	56	18.8
11–20	9,864	1,515	1,623	681	58.0	3,093	546	534	344	35.6
21–30	4,637	926	962	479	50.2	2,396	627	584	469	19.7
31–50	3,723	951	1,013	548	45.9	2,437	597	615	437	28.9
51–100	2,578	867	915	455	50.3	2,017	542	538	318	40.9
101–250	1,533	484	501	358	28.5	1,347	346	337	251	25.5
251–2600	260	108	117	101	13.7	230	93	96	91	5.2
Total	31,574	5,528	6,049	2,755	54.5	12,096	2,841	2,773	1,966	29.1

Table 4: *Backward list scheduler.* Number of basic blocks in the SPEC 2000 benchmark suite not scheduled optimally by the earliest start time heuristic (h_{est}), Shieh and Papachristou's heuristic (h_{s+p}), 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 percentage improvement given by the decision tree heuristic over the critical-path heuristic ($\% = 100 \times (h_{cp} - h_{dt})/h_{cp}$).

	1-issue					2-issue				
range	h_{est}	h_{s+p}	h_{cp}	h_{dt}	%	h_{est}	h_{s+p}	h_{cp}	h_{dt}	%
1–5	495	27	43	0	100.0	488	26	43	0	100.0
6–10	21,240	184	324	43	86.7	21,011	195	323	52	83.9
11–20	19,810	673	784	346	55.9	18,732	733	846	343	59.5
21–30	7,181	386	387	224	42.1	6,381	421	393	207	47.3
31–50	5,311	438	406	289	28.8	4,785	606	552	376	31.9
51–100	3,258	359	386	259	32.9	2,999	565	580	374	35.5
101–250	1,824	220	231	196	15.2	1,790	353	386	293	24.1
251–2600	303	54	54	53	1.9	303	79	77	71	7.8
Total	59,422	2,341	2,615	1,410	46.1	56,489	2,978	3,200	1,716	46.4

	4-issue					6-issue				
range	h_{est}	h_{s+p}	h_{cp}	h_{dt}	%	h_{est}	h_{s+p}	h_{cp}	h_{dt}	%
1–5	56	28	46	1	97.8	6	3	4	0	100.0
6–10	5,258	280	277	76	72.6	201	60	65	39	40.0
11–20	9,269	807	822	338	58.9	2,457	343	355	176	50.4
21–30	4,283	665	616	384	37.7	1,978	275	250	190	24.0
31–50	3,629	824	787	514	34.7	2,282	532	491	376	23.4
51–100	2,583	757	805	527	34.5	1,992	543	565	369	34.7
101–250	1,609	473	493	398	19.3	1,400	372	378	335	11.4
251–2600	268	106	104	106	-1.9	247	92	97	94	3.1
Total	26,955	3,940	3,950	2,344	40.7	10,563	2,220	2,205	1,579	28.4

Table 5: Number of basic blocks in the SPEC 2000 benchmark suite in each size range.

range	# blocks
1–5	324,352
6–10	94,066
11–20	46,502
21–30	13,911
31–50	9,760
51–100	5,669
101–250	2,789
251–2600	358
Total	497,407

In the remainder of the experimental evaluation, we give a detailed comparison between the decision tree heuristic h_{dt} and the critical-path heuristic h_{cp} and omit the Shieh and Papachristou heuristic h_{s+p} and the earliest start time heuristic h_{est} from further consideration. We focus on comparing against the critical-path heuristic for three reasons. First, as previously mentioned, the critical-path heuristic is the heuristic that would have been learned in the work of Moss et al. [12]. Second, the performance of the Shieh and Papachristou heuristic h_{s+p} is quantitatively and qualitatively similar to that of the critical-path heuristic; hence the relative performance of h_{s+p} can be inferred from the relative performance of h_{cp}. Third, the performance of the earliest start time heuristic h_{est} is completely dominated by the performance of the other heuristics; hence a detailed comparison against h_{est} is of lesser interest.

Table 6 shows a comparison of the number of basic blocks where one heuristic gave a better schedule than the other heuristic when coupled with a forward list scheduler. Table 7 shows the results for a backward list scheduling algorithm. On this performance measure, the decision tree heuristic is significantly better than the critical-path heuristic for small and medium size basic blocks. However, as the basic block size increases, the advantage of the decision tree heuristic decreases. Overall, for every basic block where the critical-path heuristic found a better schedule than the decision tree heuristic, there were between 2.9 and 7.8 basic blocks where the decision tree heuristic found a better schedule than the critical-path heuristic.

Table 6: *Forward list scheduler.* 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	1006	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
251–2600	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 7: *Backward list scheduler.* 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	43	—	0	43	—	0	45	—	0	4	—
6–10	19	300	15.8	23	294	12.8	13	214	16.5	0	26	—
11–20	120	561	4.7	109	610	5.6	54	544	10.1	7	188	26.9
21–30	64	247	3.9	51	251	4.9	65	318	4.9	28	88	3.1
31–50	106	228	2.2	122	297	2.4	126	406	3.2	79	211	2.7
51–100	83	271	3.3	92	360	3.9	96	460	4.8	69	303	4.4
101–250	71	119	1.7	88	216	2.5	80	243	3.0	62	158	2.5
251–2600	21	20	1.0	12	45	3.8	26	47	1.8	25	32	1.3
Total	484	1,789	3.7	497	2,116	4.3	460	2,277	5.0	270	1,010	3.7

We conclude the detailed comparison between the decision tree heuristic h_{dt} and the critical-path heuristic h_{cp} by examining how far from optimal are the non-optimal schedules found by the heuristics.

Table 8 shows performance guarantees in terms of the worst-case factor from optimality, a measure of the robustness of a heuristic, when the heuristics are coupled with a forward list scheduler. Table 9 shows the results for a backward list scheduling algorithm. For each basic block, we calculated the ratio of the length of the schedule found by the heuristic over the length of the optimal schedule. Each entry in the table is then the maximum over all ratios for basic blocks in that size range. Another way to read the entries in this table is that saying, for example, that the decision tree heuristic is within a factor of 1.25 of optimal is that the decision tree heuristic was always within 25% of the optimal value in that size range. For most size ranges and architectural models the decision tree heuristic gave much better worst-case guarantees than the critical-path heuristic. It is interesting to note that (i) the heuristics, especially the critical-path heuristic, can be quite far from optimal for relative large basic blocks (see, for example, the size range 51–100 and the 4-issue and 6-issue architectures), and (ii) the backward list scheduler, while optimal more often than the forward list scheduler, can make more costly mistakes when it returns a schedule that is non-optimal.

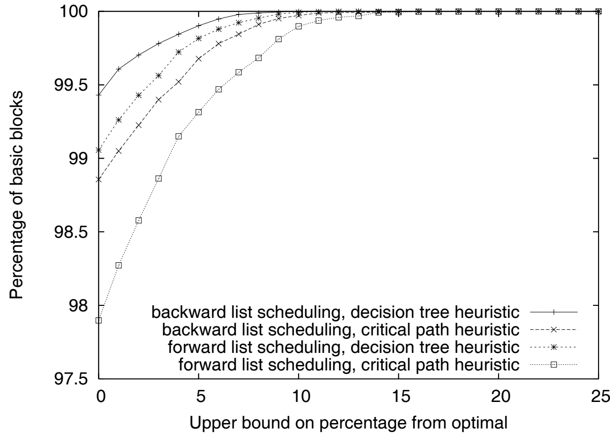
Finally, Figure 2 shows the percentage of all basic blocks in the size range 6–2600 where the list scheduler was within a given percentage of optimal. In constructing the graphs, we omitted the blocks in the size range 1–5. This was done to avoid inflating the percentages by including hundreds of thousands of basic blocks that are relatively easy for most any heuristic. The results for both the critical-path heuristic and the decision tree heuristic are shown. For example, the backward list scheduler using the decision tree heuristic is within 5% of optimal for 99.7% of all basic blocks on the 4-issue architecture. As another example, the forward list scheduler using the decision tree heuristic finds an optimal schedule (i.e., is with 0% of optimal) for 98.6% of all basic blocks on the 4-issue architecture. As is clear from the graphs, the decision tree heuristic dominates the critical path heuristic at all points for all architectures.

Table 8: *Forward list scheduler.* Performance guarantees in terms of worst-case factors from optimality for the critical-path heuristic (h_{cp}) and the decision tree heuristic (h_{dt}), for ranges of basic block sizes and various issue widths.

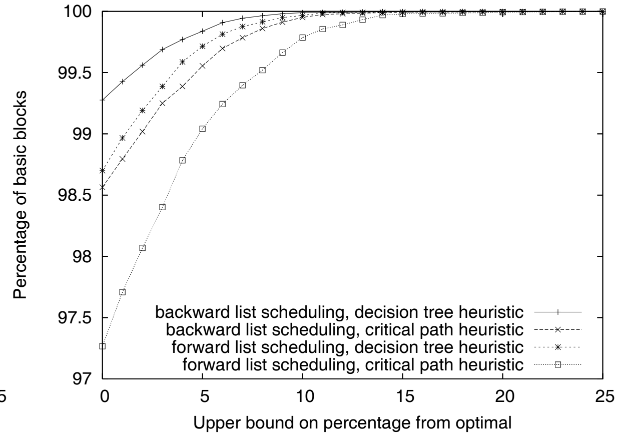
	1-issue		2-issue		4-issue		6-issue	
range	h_{cp}	h_{dt}	h_{cp}	h_{dt}	h_{cp}	h_{dt}	h_{cp}	h_{dt}
1–5	1.20	1.00	1.33	1.00	1.25	1.25	1.00	1.00
6–10	1.20	1.13	1.30	1.14	1.33	1.25	1.33	1.33
11–20	1.21	1.17	1.27	1.17	1.38	1.17	1.25	1.20
21–30	1.14	1.10	1.19	1.10	1.29	1.25	1.18	1.20
31–50	1.16	1.07	1.25	1.25	1.32	1.15	1.21	1.21
51–100	1.10	1.09	1.20	1.13	1.39	1.12	1.29	1.24
101–250	1.10	1.09	1.27	1.24	1.28	1.16	1.32	1.16
251–2600	1.01	1.02	1.24	1.26	1.16	1.13	1.05	1.06
Maximum	1.21	1.17	1.33	1.26	1.39	1.25	1.33	1.33

Table 9: *Backward list scheduler.* Performance guarantees in terms of worst-case factors from optimality for the critical-path heuristic (h_{cp}) and the decision tree heuristic (h_{dt}), for ranges of basic block sizes and various issue widths.

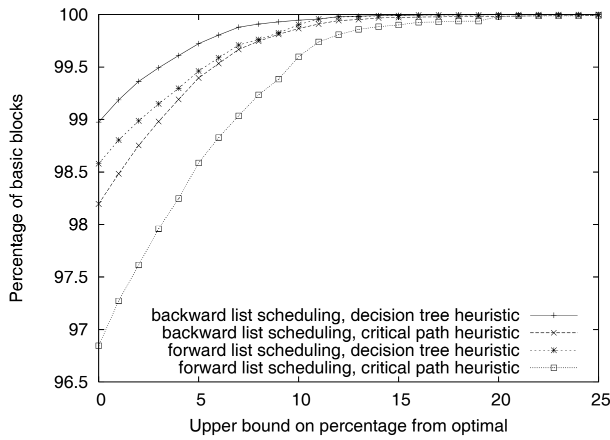
	1-issue		2-issue		4-issue		6-issue	
range	h_{cp}	h_{dt}	h_{cp}	h_{dt}	h_{cp}	h_{dt}	h_{cp}	h_{dt}
1–5	1.20	1.00	1.20	1.00	1.33	1.13	1.11	1.00
6–10	1.18	1.13	1.18	1.13	1.33	1.33	1.20	1.20
11–20	1.17	1.13	1.23	1.13	1.25	1.17	1.20	1.17
21–30	1.13	1.11	1.39	1.11	1.41	1.23	1.14	1.18
31–50	1.08	1.08	1.19	1.17	1.19	1.18	1.16	1.15
51–100	1.10	1.08	1.15	1.15	1.65	1.16	1.47	1.23
101–250	1.06	1.07	1.24	1.24	1.27	1.17	1.25	1.10
251–2600	1.01	1.02	1.32	1.28	1.23	1.24	1.22	1.22
Maximum	1.20	1.13	1.39	1.28	1.65	1.33	1.47	1.23



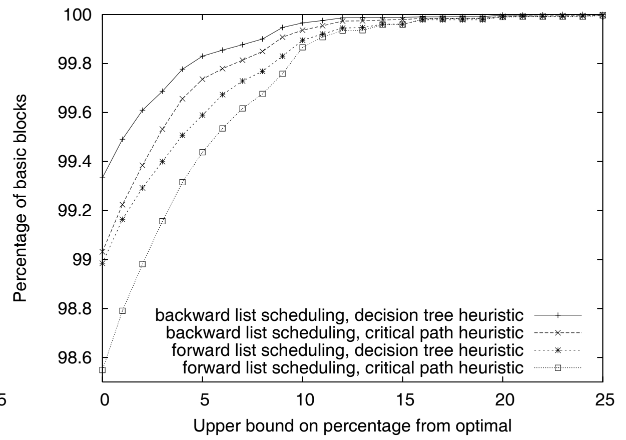
(a)



(b)



(c)



(d)

Figure 2: Performance guarantees in terms of worst-case factors from optimal for the critical-path heuristic and the decision tree heuristic, for all basic blocks in the size range 6–2600; (a) 1-issue architecture; (b) 2-issue architecture; (c) 4-issue architecture; (d) 6-issue architecture.

5 Discussion

The machine learning approach for constructing a good heuristic for basic block scheduling has several advantages over hand-crafted heuristics. The primary advantage is that of efficiency—hand-crafting a heuristic is potentially a time-consuming process. In contrast, in the machine learning approach feature construction needs to be done just once (as we have done in this paper) and all of the subsequent stages can be fully automated. This means that new heuristics can be easily generated for new architectures and for new programming languages and programming styles. A secondary advantage is that hand-crafted heuristics are prone to the well-known pitfall of over-fitting; that is, they work well on the training data to which they are tuned but not as well on data that has not been seen before. In contrast, we used techniques from machine learning which are designed to avoid over-fitting. In particular, we used feature filtering, feature selection, the use of a validation set, and most importantly, the complete separation of the data used to discover the heuristic from the data used to evaluate the heuristic. Further secondary advantages of the machine learning approach include: (i) it is possible to test many more possible combinations of features and orderings of features than in a hand-crafted approach, (ii) richer forms of the heuristics can be tested (the form of the decision tree heuristic is more complex than the form that is usually constructed by hand, which is a series of tie-breaking schemes), and (iii) the improved overall performance of the resulting heuristic.

6 Conclusion

We presented a study on automatically learning a good heuristic for basic block scheduling using supervised machine learning techniques. The novelty of our approach is in the quality of the training data—we obtained training instances from very large basic blocks and we performed an extensive and systematic analysis to identify the best features and to synthesize new features—and in our emphasis on learning a simple yet accurate heuristic. We performed an extensive evaluation of the heuristic that was automatically learned by comparing it against three previously proposed heuristics and against an optimal scheduler, using all of the basic blocks in the SPEC 2000 benchmarks. On this benchmark suite, the decision tree heuristic was better than all three of the previously proposed heuristics. In particular, the decision tree heuristic reduced the number of basic blocks that were not optimally scheduled by up to 55% compared to the popular critical-path heuristic, and gave improved performance guarantees in terms of the worst-case factor from optimality. Beyond heuristics for compiler optimization, our results also provide further evidence for the interest of machine learning techniques for discovering heuristics.

Acknowledgements

This research was supported by an IBM Center for Advanced Studies (CAS) Fellowship, an NSERC Postgraduate Scholarship, and an NSERC CRD Grant.

References

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

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

[3] U. Dorndorf. *Project Scheduling with Time Windows*. Physica-Verlag, 2002.

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

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

[6] J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, third edition, 2003.

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

[8] C. Lee, M. Potkonjak, and W. Manginoe-Smith. MediaBench: A tool for evaluating and synthesizing multimedia and communications. In *Proceedings of the 30th Annual IEEE/ACM International Symposium on Microarchitecture (Micro-30)*, pages 330–335, Research Triangle Park, North Carolina, 1997.

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

[10] 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.

[11] 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.

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

[13] S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[14] K. Neumann, C. Schwindt, and J. Zimmermann. *Project Scheduling with Time Windows and Scarce Resources*. Springer, second edition, 2003.

[15] M. Pinedo. *Scheduling: Theory, Algorithms, and Systems*. Prentice-Hall, 1995.

[16] J. R. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann, 1993. The C4.5 software is available at: http://www.cse.unsw.edu.au/~quinlan/.

[17] J.-J. Shieh and C. Papachristou. On reordering instruction streams for pipelined computers. *SIGMICRO Newsl.*, 20(3):199–206, 1989.

[18] G. Shobaki and K. Wilken. Optimal superblock scheduling using enumeration. In *Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture (Micro-37)*, pages 283–293, Portland, Oregon, 2004.

[19] 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 IEEE/ACM International Symposium on Microarchitecture (Micro-24)*, pages 93–102, Albuquerque, New Mexico, 1991.

[20] M. D. Tiemann. The GNU instruction scheduler, 1989.

[21] P. van Beek and K. Wilken. Fast optimal instruction scheduling for single-issue processors with arbitrary latencies. In *Proceedings of the Seventh International Conference on Principles and Practice of Constraint Programming*, pages 625–639, Paphos, Cyprus, 2001.

[22] H. S. Warren Jr. Instruction scheduling for the IBM RISC System/6000 processor. *IBM J. Res. Develop.*, 34(1):85–92, 1990.

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