Learning Basic Block Scheduling Heuristics from Optimal Data

Tyrel Russell, Abid M. Malik, 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. 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. 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.

Copyright © 2005 Tyrel Russell, Abid M. Malik, Michael Chase, and Peter van Beek. Permission to copy is hereby granted provided the original copyright notice is reproduced in copies made.

1 Introduction

Modern 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. 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, 12].

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 instruction to schedule next [3, 10]. 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, 13]). 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.

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. 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. [9] were the first to propose the use of supervised learning techniques in this context. Their idea was to use an optimal scheduler to correctly label the data. However, their approach was hampered by the quality of their training data; they could only optimally solve basic blocks of size ten or less and they recorded only five features in each training example[1]. McGovern et al. [8] used the same set of features, but proposed the use of reinforcement learning to overcome the difficulty of obtaining training data on larger basic blocks. However, reinforcement learning is a more difficult problem, and supervised learning, as used in this paper, is preferred whenever it is possible.

In our work, we improved the quality of the training data in three ways. First, we overcame the limitation on basic block size by using a recently proposed optimal basic block scheduler based on constraint programming [7, 14] to generate the correctly labeled training data. Currently, this optimal scheduler is too time consuming to be used on a routine basis. However, it can solve 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 [13]). 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.

[1]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.

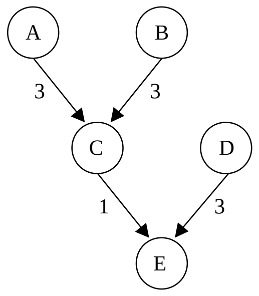
Once the training data was gathered, a decision tree learning algorithm [11] was used to induce a heuristic from the training data. The usual criteria one wants to maximize when devising a heuristic is accuracy. In this study, we also had additional criteria: that the learned heuristic be both understandable—so as to increase confidence in and acceptance of the heuristic—and efficient, and so we placed a strong emphasis on learning a simple heuristic. 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 a 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 almost 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 first define the instruction scheduling problem studied in this paper followed by a brief review of the needed background from machine learning (for more background on these topics see, for example, [3, 10, 15]).

We consider multiple-issue pipelined processors. On such processors, there are multiple functional units and multiple instructions can



(a)

A	r1 $\leftarrow a$
B	r2 $\leftarrow b$
	NOP
	NOP
C	r1 $\leftarrow$ r1 + r2
D	r3 $\leftarrow c$
	NOP
	NOP
E	r1 $\leftarrow$ r1 + r3

(b)

A	r1 $\leftarrow a$
B	r2 $\leftarrow b$
D	r3 $\leftarrow c$
	NOP
C	r1 $\leftarrow$ r1 + r2
E	r1 $\leftarrow$ r1 + r3

(c)

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.

be issued (begin execution) each clock cycle. Associated with each instruction is a delay or *latency* between when the instruction is issued and when 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. Examples of types of instructions are load/store, integer, floating point, and branch instructions.

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.

Given a labeled dependency DAG for a basic block, a *schedule* for a multiple-issue 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 latter 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.*

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 [10]. 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.

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.

In supervised learning of a classifier from examples, one is given a training set of instances, where each instance is a vector of feature values and the correct classification for that instance, and is to induce a classifier from the instances. The classifier is then used to predict the class of instances that it has not seen before. Many algorithms have been proposed for supervised learning. One of the most widely used is decision tree learning. 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.

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. [9] 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. We began with almost 60 features that we felt would be promising. These included all of the classic features surveyed by Smotherman et al. [13], except for the features having to do with register pressure 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. We also included features that measured properties of the DAG to be scheduled (such as the source language and the number of instructions of each type) and of the architecture (such as the number of functional units of each type).

A more accurate classifier can sometimes be achieved by synthesizing new features from existing basic features. We also included many such synthesized features. Some of the novel features were constructed by applying simple functions to basic features. Examples include comparison of two features, maximum of two features, and the average of several features.

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 $r_1(i,t) + r_3(i,t)$.

Table 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.

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 SPEC 2000 integer and floating point benchmarks [http://www.specbench.org] and the jpeg and mpeg benchmarks from the MediaBench [6] benchmark suite. The benchmarks were compiled using IBM's Tobey compiler [1] targeted towards the PowerPC processor [5], and the basic blocks were captured as they were passed to Tobey's instruction scheduler. 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. The basic blocks contain four types of instructions: branch, load/store, integer, and floating point.

Following Moss et al. [9], a 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 k^{th} 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 [7, 14] to determine the length of an optimal schedule if that instruction were 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). 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, and the 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 [15, 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

As mentioned above, we began with 60 features that we felt would be promising for learning a good heuristic. We then gathered a data set where each instance in the set was a vector of these features and the correct classification for that instance.

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 [15, pp.231-232]).

Several feature filtering techniques have been developed (see, for example, [4] 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 by 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 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 [15] 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-

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—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.

ranked features in any heuristic. A surprise is that critical-path distance to the leaf node, commonly used as the primary feature [3, 10]), is ranked relatively lowly. 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. 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. The usual criteria one wants to maximize when selecting a classifier is classification accuracy. In this study, we also had additional criteria: that the learned classifier be both understandable—so as to increase confidence in and acceptance of the heuristic—and efficient, and so we placed a strong emphasis on learning a simple classifier.

Ideally, 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 [4, 15]. 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. 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, user-interpretability, and efficiency. To learn a classifier, we used Quinlan's C4.5 decision tree software [11]. The software was run with the default parameter settings, as this consistently gave the best results on the validation set.

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;
```

The following table 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.

level	1	2	3	4
size	4	7	14	17
accuracy	4.05	3.82	3.76	3.72

level	5	6	7
size	30	48	43
accuracy	3.71	3.71	3.70

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 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 decision tree heuristic constructed is shown in Algorithm 1.

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.

The decision tree heuristic was compared against a previously proposed list scheduling heuristic and against the schedules found

Table 3: 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
251–2600	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
251–2600	358	117	101	13.7	96	91	5.2
Total	497,407	6,049	2,755	54.5	2,773	1,966	29.1

by our optimal scheduler. Following Muchnick [10], the list scheduling heuristic we compared against used critical-path distance as the primary feature, earliest start time as a tie-breaker if the critical-path distances were equal, and order within the instruction stream as a tie-breaker if both the critical-path and earliest start times were equal. We refer to this heuristic as the critical-path heuristic. We chose this heuristic to compare against as it is similar to the heuristic that would have been learned in the work of Moss et al. [9], and because critical-path distance is the most popular feature in the heuristics surveyed by Smotherman et al. [13].

Table 3 shows the number of basic blocks in the SPEC 2000 benchmark suite that were *not* scheduled optimally by the critical-path heuristic and the decision tree heuristic. 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 also given. For both 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, and that overall reductions in the number of basic blocks not optimally scheduled of between 29% and 55% were achieved compared to the critical-path heuristic.

Recall that the Tobey compiler performs instruction scheduling before global register allocation and once again afterwards, and that our test suite contains both kinds of basic blocks. It

Table 4: 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

is interesting to note that both heuristics were more effective for the basic blocks from before register allocation and less effective for those after; approximately 2/3 of the basic blocks that were not scheduled optimally were from after register allocation had been performed.

Table 4 shows a comparison of the number of basic blocks where one heuristic gave a better schedule than the other heuristic. On this performance measure, the decision tree heuristic is significantly better than the 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 5 shows performance guarantees in terms of the worst-case factor from optimality, a measure of the robustness of a heuristic. 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.

5 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 a popular critical-path heuristic and against an optimal scheduler, using all of the basic blocks in 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.

For future work, we intend to apply the same

Table 5: 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	1.200	1.000	1.333	1.000	1.250	1.250	1.000	1.000
6–10	1.200	1.125	1.300	1.143	1.333	1.250	1.333	1.333
11–20	1.214	1.167	1.273	1.167	1.375	1.167	1.250	1.200
21–30	1.136	1.097	1.190	1.100	1.286	1.250	1.176	1.200
31–50	1.156	1.073	1.250	1.250	1.320	1.150	1.214	1.214
51–100	1.103	1.086	1.200	1.125	1.389	1.123	1.286	1.241
101–250	1.097	1.088	1.273	1.240	1.275	1.163	1.324	1.160
251–2600	1.009	1.023	1.241	1.258	1.163	1.126	1.049	1.060
Maximum	1.214	1.167	1.333	1.258	1.389	1.250	1.333	1.333

methodology to the scheduling of superblocks. In contrast to a basic block which has a single entry point and a single exit point, a superblock has a single entry point but multiple exit points. Superblocks are larger units of code than basic blocks and are constructed from multiple basic blocks. To automatically learn scheduling heuristics for superblocks from optimal training data, the next step will be to develop an optimal scheduler for superblocks.

Acknowledgements

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

About the Authors

Tyrel Russell is a Master's student in the School of Computer Science at the University of Waterloo. He completed his undergraduate degree at the University of Northern British Columbia and is the recipient of an NSERC Postgraduate Scholarship. His email address is tcrussel@cs.uwaterloo.ca.

Abid M. Malik is a PhD student in the School of Computer Science at the University of Waterloo. He completed his Master's degree at the University of Guelph and is the recipient of an IBM Center for Advanced Studies (CAS) Fellowship. His email address is ammalik@cs.uwaterloo.ca.

Michael Chase is a Master's student in the School of Computer Science at the University of Waterloo. He completed his undergraduate degree at the University of Waterloo. His email address is mjchase@cs.uwaterloo.ca.

Peter van Beek is a Professor in the School of Computer Science at the University of Waterloo. His research interests are in constraint programming and its application to scheduling and other optimization tasks. He has coauthored four papers which have won awards; is Editor-in-Chief of the Constraints Journal, a journal devoted to the area of constraint programming; and is serving as program chair for CP 2005, the premier conference in constraint programming. His email address is vanbeek@uwaterloo.ca.

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

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

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

[6] 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.

[7] A. M. Malik, J. McInnes, C.-G. Quimper, 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.

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

[9] 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.

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

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

[12] 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.

[13] 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, Albuquerque, New Mexico, 1991.

[14] 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.

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