Using Machine Learning to Build Flexible Instruction Schedulers

John Cavazos
Department of Computer Science
University of Massachusetts, Amherst
Amherst, MA 01003 U.S.A.
cavazos@cs.umass.edu

April 9, 1997

Abstract

Hardware implementations of a given computer architecture proliferate quickly; every few months new generations of chips are announced. However compilers do not keep this pace, because of the time consuming process of tailoring optimization and code generation for specific architectures. We want to automate parts of the compiler development process using adaptive techniques, such as machine learning. In particular, we consider local instruction scheduling — the ordering of instructions of a basic block for efficient execution on a modern superscalar processor. This is an important problem in compiler optimization, because it is easy for programs to suffer very bad performance (twice the time or more of good schedulers) with naive scheduling algorithms. Choosing the optimal schedule sequence requires looking at all legal (subject to data-dependence constraints) permutations — in practice the number of legal permutations is very large though lower than the worst case, $n!$. Heuristics may find close to optimal solutions, but due to the requirement that scheduling be done in linear time, the heuristics applied must be kept simple. Hand-crafted heuristics are difficult to devise, and it is not easy to adapt them to new hardware implementations. In this paper, we describe a novel approach to instruction scheduling. We incorporate a machine learning component, such as a neural network or a decision tree into an instruction scheduler, in order to construct a scheduler that is flexible. We found that the technique is effective, in that the schedules produced are exceptionally close to the hand-crafted schedulers of the commercial compilers (and by extension to the optimum). Given a set of good features, the best of these adaptive instruction schedulers produces code with execution cost no worse than 1% slower than a manufacturer supplied scheduler provided by Digital.

1 Introduction

As computer architectures become increasingly more complex, more sophisticated compiler optimizations are required to take advantage of new features of the architecture, especially in instruc-

tion scheduling. However, the fast paced development of new technology precludes spending a large amount time handcrafting complicated instruction schedulers. Hand-crafting the instruction scheduler also means the compiler writer has less time to spend on other parts of the compiler, which can further increase the potential of performance gains. Also, computer engineers desire the ability to experiment with many different designs of an architecture to measure gains of one architecture over another. This not only requires being able to quickly prototype the architectures in question (either by using Field Programmable Gate Arrays or simulators), but also requires the quick prototyping of optimizing compilers for those architectures. Supporting quick prototyping of optimizing compilers precludes hand tuning any part of the compiler, including the instruction scheduler. We propose using a machine learning component, such as a neural network or decision tree, to automate generation of the heuristics used to schedule instructions.

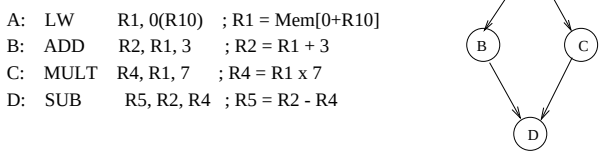
So we ask the following questions:

1. Is it possible to apply a generic greedy algorithm guided by the evaluation of certain features of the instructions being scheduled?

2. Can the decision-making of such a greedy scheduling algorithm be constructed by a machine learning component (MLC), such as a neural network or a decision tree?

We extracted basic blocks from Fortran programs in the SPEC95 benchmark suite, compiled on the Digital Alpha architecture for the 21064 chip implementation[1]. We computed a set of features that served as input to a MLC, either a comparator artificial neural network (CANN) or a decision tree. The MLCs were trained to choose the better of two candidate instructions (when either could legally follow). We performed cross-validation: we trained the MLC on all benchmarks save one, then tested it on that one. The testing phase consisted of integrating the trained network or induced decision tree into an instruction scheduler. Basic blocks were scheduled using this instruction scheduler and then run on a simulator of the target architecture to evaluate the costs of the schedules. We found that the technique is effective, in that the schedules produced were reasonably close to the hand-crafted heuristics of a public domain Digital scheduler (and by extension to the optimum), within .09%-1.43%, and .54% on the average for one experiment. This paper presents preliminary results of using neural networks and decision trees to combine and derive the importance of features in an instruction scheduler.

In Section 2, the problem of local instruction scheduling is described. Our experimental framework is presented in Section 3. This section elaborates on the process of building and evaluating a MLC scheduler, from training to scheduling. In Section 4, we describe the kinds of features found in our features sets. In Section 5, we formulate the local scheduling problem as a machine learning problem and describe the two MLCs that were used to solve this problem. The methods used to evaluate the MLC schedulers are presented in Section 6. In this section, we also describe the different feature sets and benchmarks used in our experiments. The results of our experiments are presented in Section 7. The idea of using machine learning in instruction scheduling is not new, but previous techniques were inadequate for use with modern hardware and commercial compilers. In Section 8, we present previous work in this area. In Section 9, we discuss future areas of work for this research; Section 10 concludes.



(a) Instruction Sequence to be Scheduled

(b) Dependence Dag of Instructions

1) A B C D

2) A C B D

(c) All Possible Schedules

Instructions	Cycles
LW (A)	1
ADD (B)	1
MULT (C)	2
SUB (D)	1

(d) Latency of Instructions

Expected Execute Cycle	1	2	3	4	5
Bad Ordering (5 cycles)	A	B	C	stall	D
Good Ordering (4 cycles)	A	C	B	D	

(e) Schedule Timings

Figure 1: On some architecture implementation.

2 The Local Instruction Scheduling Problem

A basic block consists of a list of instructions with a single entry and single exit point. Local instruction scheduling pertains to scheduling the instructions of a single basic block without regard to the effect that schedule may have on other blocks that follow. Global instruction scheduling techniques, such as trace scheduling [2] and percolation scheduling [3], pertain to methods of considering multiple basic blocks when scheduling. Global instruction scheduling may be an interesting machine learning problem and can lead to more benefits than local scheduling alone; however, these techniques do not preclude the need for good local scheduling and therefore our research is an essential first step. The rest of the paper pertains to local instruction scheduling, unless otherwise specified.

At any point during scheduling of a basic block, there is sequence of instructions that have been scheduled, the *partial schedule*, and a directed acyclic graph, called a *data dependence dag* (DDD), of instructions remaining to be scheduled. The DDD represents the data dependencies of the instructions in the basic block. Therefore, before scheduling can begin, a DDD of instructions must be built for the basic block to be scheduled. A final schedule consists of a configuration in which all instructions have been removed from the DDD and placed in the sequence of scheduled instructions. The cost of a schedule is the number of cycles the machine takes to execute the scheduled sequence. Figure 1 shows a sequence of instructions (*a*) that make up a basic block and its corresponding DDD (*b*). The figure also shows all the possible schedules for the DDD (*c*), the latencies of the instructions in the sequence (*d*), and estimated scheduled times (*e*) for some architecture implementation. As can be seen in the figure, some schedules are better than others.

Our method of scheduling follows the traditional approach to instruction scheduling : instructions are scheduled from the roots of the DDD to the leaves, and the DDD is maintained along the way to determine which instructions are candidates for scheduling. We illustrate this approach in Figure 2, by showing the steps required to arrive at the schedule *ACBD* for the DDD in Figure 1.

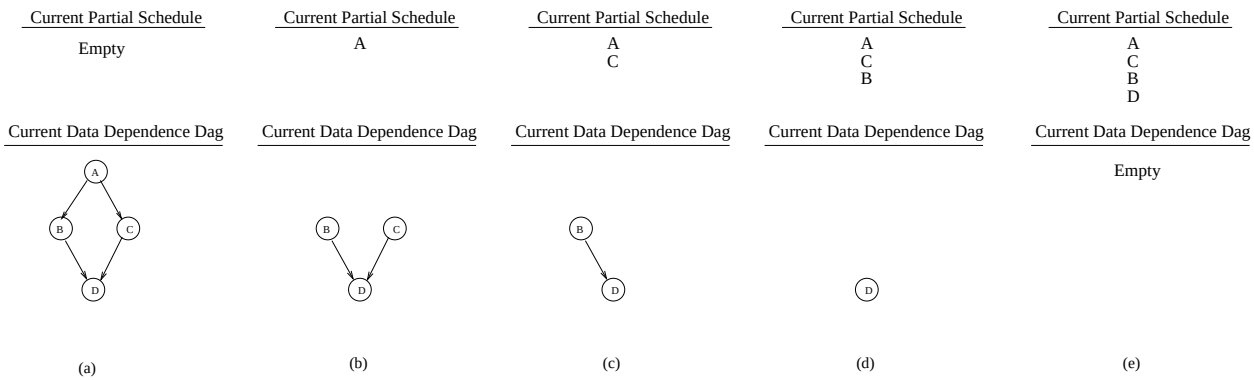


Figure 2: The sequence of scheduling steps to schedule DDD in Figure 1 to obtain schedule *ACBD*.

In Step (*a*), instruction *A* is the only instruction that has no dependencies on other instructions in the basic block (this makes the instruction a root) and therefore must be scheduled first (at any scheduling point, only the roots in the DDD can be scheduled). As we schedule instruction *A* in Step (*b*), we remove its corresponding node and its outgoing dependence edges from the DDD. Removing dependencies can make other instructions roots, causing these instructions to be available for scheduling. After we schedule instruction *A*, we can schedule either instruction *B* or instruction *C*, since neither of them have predecessors in the DDD. In Step (*c*), the scheduler chooses instruction *C* to schedule next. Since instruction *D* depends on *B*, *D* does not become a root of the newly formed DDD and cannot be scheduled. Step (*d*) corresponds to the scheduling of instruction *B* and its node and outgoing dependence edge being removed. In Step (*e*), we reach a final legal schedule by scheduling instruction *D*. The last node from the DDD is removed and the schedule contains all the instructions from the original DDD.

Instruction scheduling requires the use of a specific algorithm to construct schedules from basic blocks. The algorithm must not be exhaustive and if backtracking (i.e., the unscheduling of instructions to investigate alternatives) is used it should be done in a restricted amount. Our system supports backtracking, but we have not yet used it in the scheduling algorithm. A DDD induces a partial order to the basic block, therefore the number of final legal schedules for the basic block corresponds to the number of total orders for the DDD. Brightwell and Winkler [4] showed that determining the actual number of total orders in a dag, given a partial ordering, is #*P*-complete, thus exhaustive scheduling is infeasible. We concentrate on the greedy scheduling algorithm, which schedules the best instruction at the decision points, that is at the decision points we schedule the instruction that adds the least cost when added to the set of instructions already scheduled. "Best" is derived from the heuristics embedded in the scheduler.

We postulate that any heuristics used by a greedy scheduling algorithm can be calculated from a set of features that describe the present state of the scheduling process and the characteristics of the candidate instructions, provided such a set of features is well chosen . The present state of the scheduling process embodies the processor state and some aspects of the remaining DDD structure. The characteristics of the candidate instructions embody properties of the DDD structure as seen from their point of view. The results presented in Section 7 verify that heuristics can be calculated from a set of features.

Every schedule, whether partial or complete with respect to the basic block, has a well-defined cost associated with it, the number of cycles that the schedule will take to execute the schedule's

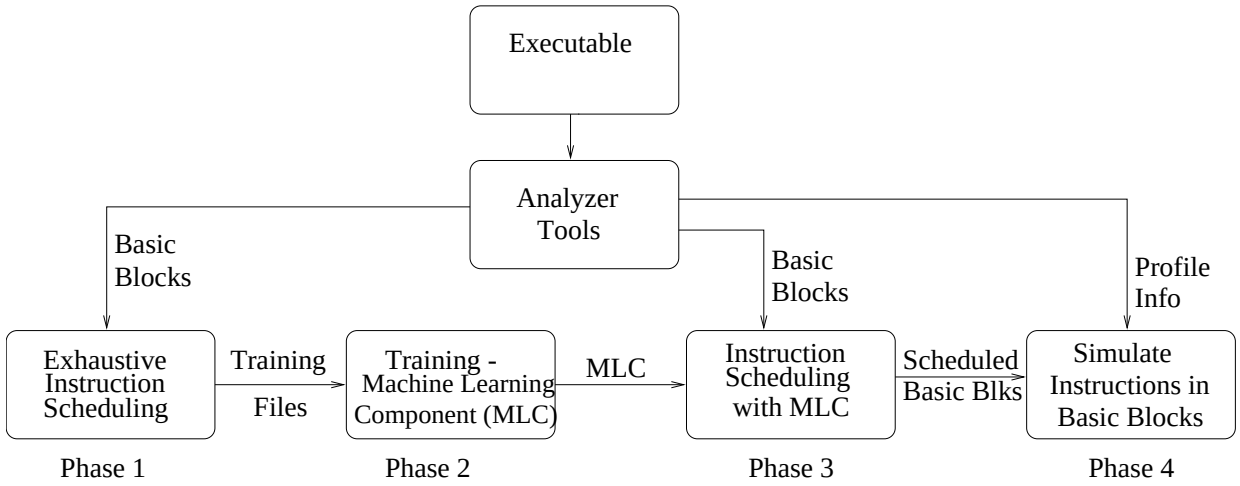


Figure 3: Adaptive Instruction Scheduler Framework

instructions. We obtain the cost by simulating each basic block using a simulator provided by Digital. This simulator models the Alpha 21064 architecture implementation. It simulates memory and chip resource constraints, including functional units pipelines, register usage, and latency of instructions. Another technique for simulating the cost of basic blocks called self-simulation is proposed in [5]. Baker there proposes a method, which, with minor modifications to chip implementations, would allow a compiler to execute individual basic blocks directly on the hardware. This is an intriguing concept since it would allow a MLC to learn intricate details of the hardware automatically, instead of someone having to provide them by hand.

3 Experimental Framework

Figure 3 depicts the different phases that are completed during the construction and evaluation of our MLC instruction schedulers. Before these phases begin, analyzer routines are used to extract basic blocks from our corpus of executable benchmarks. Other analyzer routines generate profile information, which is needed for estimating the performance of our MLC scheduler and for performing weighted training in our neural networks (explained in Section 5.1.1). The first phase of our experimental framework, uses the basic blocks created by our analyzer routines for generating training files. During this phase, a scheduler is given a certain amount of time in which to perform exhaustive scheduling. Enough time is allotted so that, for most blocks, the scheduler is able to find optimal schedules. For many large blocks, however, exhaustive scheduling is impossible and scheduling is therefore terminated after the time allotted. After we exhaustively schedule a block, we output features for that block into a file which will be used for training our MLCs.

Features are generated for the instructions at each decision point. A decision point is a place in the schedule where there is more than one instruction that can be scheduled. Thus a decision has to be made. At each of these decision points, we output a line of features (described in Section 4) for each candidate instruction together with three additional features: the cost of the best possible schedule that can be achieved by scheduling this instruction, the cost of the best possible schedule for the best candidate at this decision point, and the best possible schedule cost for that block. If

we were not able to exhaustively schedule a block, we use the best cost that was found before scheduling was terminated.

There are currently over three hundred and fifty features that can be generated for each line of features. This number far exceeds the capability of most learning techniques for size of data sets we were training over (tens of thousands), thus we assume a subset of features deemed interesting would be selected for learning. The second phase is training the machine learning component with filtered data sets. Once the set of features to be experimented with is chosen, the original feature files are filtered to create the data sets with which to train the MLC. Given we know the best possible schedule cost of each candidate instruction if it were to be scheduled at a certain decision point, we know which candidate is best to schedule and we can train a machine learning component, to pick this best candidate in a supervised setting. We elaborate on how we phrase the problem as a machine learning problem in Section 5. The third phase is the integration of the trained MLC into a scheduler. During this phase, learning is turned off, otherwise we would slow scheduling to an unacceptable level. In this phase, we also schedule blocks with other schedulers we wish to compare against (e.g., a random scheduler). In order to evaluate the performance of our MLC schedulers, we schedule basic blocks from our benchmarks and measure their schedule cost with the simulator. Thus in the fourth and final phase, we measure the cost of the basic blocks scheduled with each of the different schedulers. In our experiments, we scheduled with our MLC schedulers and several different schedulers described in Section 6.4. As mentioned, the scheduler was implemented through simulation — it was not actually implemented in the back end of a compiler — therefore a simulator was required. We used an instruction scheduler and an Alpha 21064 simulator both provided to the public domain by Digital. A large amount of code was built around these two components, including an interface to the simulator and scheduler and code for feature generation. All code relys heavily on ATOM, an instrumentation tool for Digital Alpha workstations. With this set of tools, we were able to integrate our trained MLCs into a scheduler quite effectively. The simulator was developed by computer architects at Digital, therefore we feel it is a good indication of actual costs incurred by basic blocks at run-time on this particular chip implementation.

4 Features

The features we use can be classified into three categories depending on the amount of knowledge about the architecture required to compute the features. The categories are *black-box* (no internal knowledge of the architecture needed), *gray-box* (a limited amount of knowledge of the architecture needed), and *white-box* (deep internal knowledge of architecture required). Black box features use the original DDD, the partial DDD, and the partial schedule for their calculation, but do not make any assumptions about computer architecture in doing so. Gray and white box features, also use these structures, but in addition they take advantage of characteristics of the machine for which we are building a scheduler. Because of the accurate information these features provide, we postulate that white and gray box features are essential in building schedulers that are competitive with those found in commericial compilers. Results presented in Section 7.1 and in Section 7.3.3 seem to support this hypothesis.

These categories are also a good indication of which features must be modified when considering a new architecture. For new architectures the computation of black box features should remain

the same, some of the gray box feature calculation may need to be modified, and most, if not all, of the algorithms used to compute white box features will have to be rewritten or at least adapted a fair amount. The selection of features we implemented was influenced by our understanding of the kinds of information needed by the various heuristics described in the literature. Some of the features can take parameters, which allows us to vary the amount of information or the level of detail returned by the feature. For instance, one feature, Scheduling Pressure of an Instruction Class, can take a parameter specifying how far back in the schedule we need to look for pressure of a certain instruction class. We also provide several representations of same feature when appropriate. This allows us to use input representations that better suit the learning technique being used. For example, the feature Instruction Class is provided as a number between 0 and 18. This should be fine for decision trees, but given that this feature represents unordered categorical data, we also provide it as a 1-of-n binary encoding, which may be more appropriate for neural networks. It also helps to categorize the features based on the information they carry. Features can carry information about the basic block as a whole, the current partial schedule, a candidate instruction, or a combination of both a candidate instruction and the current partial schedule. That is, we can classify features based on whether they are independent or dependent of the candidate instruction waiting to get scheduled. Features that are independent of any candidate instruction describe the current partial schedule or the DDD, without regard to any candidate instruction waiting to be scheduled. When a new instruction is scheduled, these features typically need to be recomputed. Features of the candidate instruction provide information about the candidate instruction or the candidate instruction and some other aspect of the scheduling state, such as the DDD. The "combined" features describes information about the candidate instruction as it pertains to current partial schedule. For instance, the feature Last Pred Issued Long Ago is calculated by finding the instruction last scheduled which the candidate instruction is dependent on. Therefore, this features requires (and is therefore dependent on) a candidate instruction to be calculated.

5 Scheduling as a Machine Learning Problem

One of our main concerns was formulating the problem as a machine learning problem. For experiments presented in this paper we settled on a method of preference learning, where we present a MLC with two instructions and train it to prefer the instruction resulting in the lowest schedule cost. Figure 4 depicts what a typical training pattern for a neural network looks like (the training patterns for DTs differed only in format). Three sets of features are presented to the MLC. The first set are independent of the two candidate instructions that are being compared. These features pertain to the state of the schedule at the current decision point. The second and third set of features correspond to one of the candidate instructions to be scheduled. Each line, therefore, pertain to some state information and the features of two instructions we are comparing. The target values (or class labels) to be learned are zero or one. A target value of zero corresponds to first instruction being preferred, and a target value of one pertains to the second instruction being preferred.

5.1 Scheduling using Neural Networks

We required a MLC that could learn preferences of instructions by categorizing a "pattern" or feature vector that carried information about the instructions which were candidates to be scheduled

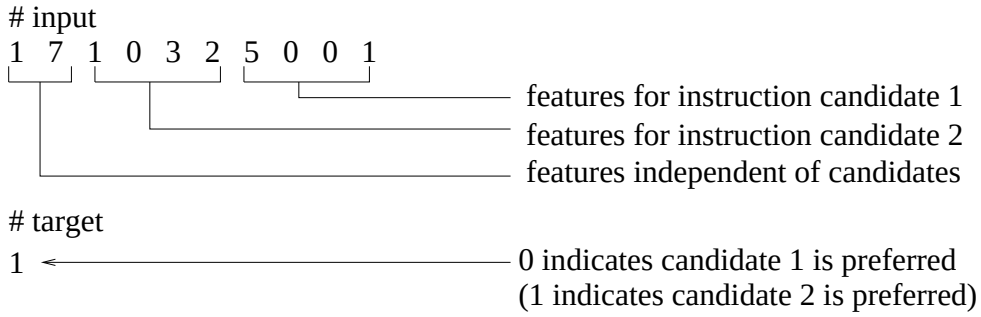


Figure 4: A typical training pattern.

and the current state of the scheduling process. One such MLC is a feed-forward neural network [6]. A feed-forward neural network has a topology consisting of groups of neurons or units forming layers. There are some number of input units forming an input layer, some number of hidden units arranged in zero to several hidden layers, and some number of outputs units forming an output layer. In our network topology, each layer i is fully to connect to each layer $i+1$, and these connections become either excitatory (positive weights), inhibitory (negative weights), or irrelevant (weights close to zero) during the training of the network. Input values from training data are fed simultaneously to the input layer, and the outputs of these units are simultaneously fed to the first hidden layer. This process continues until the outputs of the last hidden unit are fed as inputs to the output layer, and the network final output (the instruction preference) is produced. A neural network's capacity to learn largely depends on the number hidden units and the input representation. We train our neural networks in a supervised setting, that is after input values are given to the network, the correct response for these inputs is given to the network, which is used in the backpropagation algorithm for adjusting the weights and thresholds of the units. Backpropagation is an algorithm that iteratively computes slight changes to all of the weights and thresholds in the network in the direction of the fastest decrease to the sum of squared errors. Generalization is a highly desirable quality of NNs. Generalization refers to the neural networks ability to perform well (classify) on new data different from the data seen in training. It is important that the neural network does not memorize, or overfit, the training data. To alleviate this problem, we retain some test data different from the data used for training the network. As the network is learning, its performance on this test set is monitored. The training process is terminated when the network's performance does not improve for the test data. Several algorithms can be used to train a network, many of which are variants of the standard backpropagation.

Two algorithms we experimented with include Backpropagation with Momentum (BPM) and Scaled Conjugate Gradient (SCG). BPM uses a momentum term in the standard backpropagation algorithm. The momentum term introduces the old weight change as a parameter in the computation of the new weight change. The effect of momentum is that flat spots of the error are traversed rapidly with a few quick steps, thus increasing the learning speed significantly. SCG performs a special kind of gradient descent, which uses the gradient vector (first order partial derivatives) and the Hessian matrix (second-order partial derivatives) of the error function. Unlike standard backpropagation, which always proceeds down the gradient of the error function, conjugate gradient methods perform a gradient descent in $d-$dimensional space, by finding d directions along which the system is successively transformed. The largest possible step to take that decreases the error

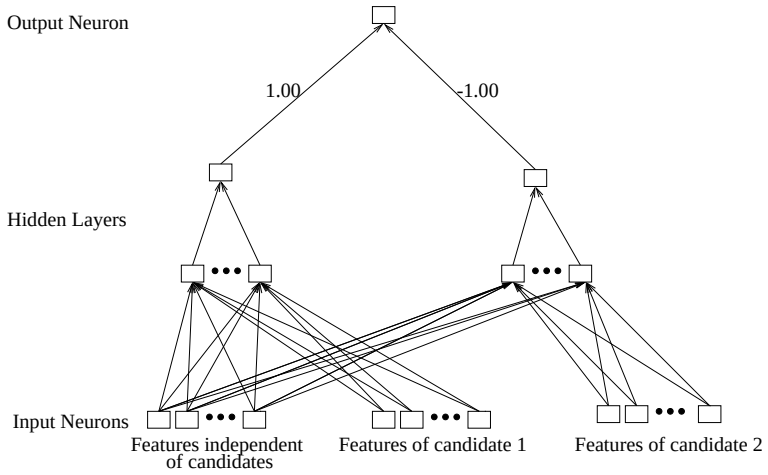


Figure 5: A schematic of a CANN.

function in each direction is taken, where the $(i+1)$th direction is chosen to be conjugate, with respect to the Hessian matrix, to the previous i directions. After several self-validation experiments (explained in Section 7.2.1), we chose SCG as the algorithm to update our networks. The results of these experiments are presented in Section 7.2.1. We found no significant difference between the two algorithms and SCG had one obvious advantage over BPM that made the choice clear: it took BPM many iterations to converge, typically between 2000-5000, while SCG normally converged in less than 50 iterations. SCG was thus much less computationally expensive than BPM.

The type of neural network used depends heavily on the way the scheduling problem is phrased. Given that we phrased our problem as learning of preferences of which instruction to schedule at a decision point, we chose to experiment with a comparator neural network structure. Figure 5 depicts the schematic of a comparator artificial neural network (CANN). CANNs were invented by Tesauro [7] for learning to compare alternative moves for the game of backgammon. A good description of CANNs and a discussion of their application to load balancing can be found in [8]. Since we are learning to choose the best of n alternatives, as in [8], we noticed the applicability of this network to our problem. We implemented CANNs using feed-forward neural networks [6]. Our CANN consisted of three sets of inputs. Each of the two sets on inputs receives features dependent on one of the candidate instructions involved in the comparison. The third set of inputs receives features independent of the instructions being compared. A discussion of features can be found in Section 4.

5.1.1 Weighted versus Unweighted Training

We also considered weighted versus unweighted training. Weighted training allows the learning algorithm to account for the importance of the basic blocks to program performance. During weighted training, each training pattern has an importance factor associated with it, which pertains to the offensiveness of the basic block (i.e., the contribution of the basic block to the total runtime of the benchmark containing it) that generated the pattern. The weight of a training pattern is multiplied by η, the step size taken by the gradient descent, before it is used in the backpropagation algorithm. Thus, training patterns generated from important basic blocks have a greater effect on

the MLC than training patterns from less important blocks.

It is imperative that important basic blocks be scheduled well, therefore treating them differently may increase our scheduling performance. In one experiment using CANNs, we noticed that bad scheduling of just one important basic block introduced over a 30% performance degradation to the overall execution cost of the benchmark. Important basic blocks tend to be larger more frequently executed blocks, such as blocks typically found in inner loops. A block's importance, and therefore the weights of the training patterns from that block, are generated from profile information. The results from training over the first feature set we used suggested that CANNs trained with weighted patterns schedule blocks better than those trained with unweighted patterns. This suggested that important basic blocks had properties inherently different from those of smaller blocks, and that these properties were important to the learner. In training our CANNs, the weights were used to increase or decrease (depending on the weight of the pattern) the step size of the gradient descent and thus we were biasing learning to the important basic blocks. Interestingly, weighted training with a better set of features did not help in the learning. Training using weighted patterns was not been integrated into a decision tree package, and is left as future research.

5.2 Scheduling using Decision Trees

We formulated the problem of instruction scheduling into one of classifying patterns. Given a set of input features respresenting present state and two instruction candidates, we train a MLC to label the pattern, either with a 0, corresponding to the first instruction being preferred over the second, or a 1 corresponding to the second instruction being preferred over the first. Formulating the problem in this manner, allowed us to use a technique of building classifiers from training data known as decision trees (DT). A DT is a structure consisting of either a leaf, indicating a class, or a decision node that specifies a test, corresponding to the value of a single attribute. Each outcome of the test has a branch and a subtree associated with it. There are effective and efficient algorithms for DT induction directly from a corpus of training data consisting of pattern instances with their assigned classification [9]. As with neural networks, it is possible for a decision tree to overfit to training data. We address this problem by pruning the resultant trees induced by the decision algorithm. Pruning consists of discarding one or more subtrees and replacing them with leaves. The DT induction experiments reported here were performed using the ITI system [10]. For experiments comparing CANN and DT schedulers, we use the same training data for both MLCs (with the exception of slight formatting changes). One advantage of decision trees over neural networks is that the generated trees can be directly interpreted by experts in the domain of interest. Figure 6 depicts a decision tree built with ITI with the Digital Predicate features explained in Section 7.2 (in the figure the features have been converted from numerical values to categorical labels for facillitate the interpretation of the tree).

6 Evaluation Methods

Learning algorithms are typically measured by their accuracy over an unseen test set. Some experiments proved this was an inadequate method to evaluate our trained MLCs. A basic block may have several decision points and at each decision point there may be several candidate instructions from which to choose. The decisions in our training files correspond to intermediate decisions and

are not a direct measure of how well we can schedule instructions. In fact, we found that high accuracy on test data did not imply one had a good scheduler. We therefore created schedulers that used trained MLCs. The quality of a MLC was measured by the total estimated cost of the benchmark it scheduled. The estimated cost was derived by accumulating the simulated cost of each basic block multiplied by the number of times it was executed, as reported in the profile information.

6.1 Self-Validation and Cross-Validation

We trained our CANNs using *self-validation*, that is training on data from a particular benchmark and then scheduling that benchmark using the trained network. This is beneficial for a couple of reasons. First, it gives us a upper bound on the quality of schedules that we can expect from a CANN scheduler. We are training the network with the exact patterns it will see at the time of scheduling. Therefore, we are not relying on the network's ability to generalize, but are instead interested only in the amount of information that can be captured by the network and whether the task is in fact amenable to a neural network configuration. These experiments were also used as an existence proof; that is, we wanted to prove there was a set of features, a set of training patterns, and a neural network which, when trained on those training patterns, could schedule instructions well (reasonably close to the quality of a manufacturer provided heuristic scheduler). Since it isn't reasonable to expect to achieve better results than those returned with the self-validation technique, experimenting with this method allowed us to tune several neural network parameters, such as the number of hidden units, the best training algorithm to use, etc. There are few parameters to tune with decision trees. Therefore, we used self-validation with decision trees only to get a upper bound on their performance and not for tuning parameters.

It is unknown at the time we are training our MLC, which programs a compiler with our instruction scheduler might be used to compile. Therefore, a *cross-validation study* is a true test of how well our scheduler can be expected do in practice. Our method of cross-validation, "leave-one-out" cross-validation, consisted of taking the training data from all of the programs, except one, and feeding the corpus of training data into the CANN or DT. We then used the trained CANN or DT to schedule the program not included in the corpus. We performed cross-validation on benchmarks for the same language only; These benchmarks include the ten Fortran benchmarks from Spec95 (described in Section 6.2). Therefore, during our cross-validation study, we were training a MLC with training data from nine of the Fortran benchmarks. The tenth benchmark that was left out was then used for scheduling. Each cross-validation experiment resulted in ten trained MLCs which were used for scheduling the ten Fortran benchmarks. Cross-validation tests the MLC's ability to generalize. Preliminary results with CANNs seemed to indicate that a MLC performed better if it was trained using only training patterns from a corpus of benchmarks of the same language as the benchmark, as opposed to using a corpus of different languages from which to train and schedule from. This result seemed reasonable, given that different languages and different compilers could cause basic blocks to have inherently different characteristics, which would require different heuristics for scheduling them. This assumption was invalidated by results from decision trees (see Table 8), which showed that indeed, we could train on a corpus of benchmarks from one language and schedule benchmarks from a completely different language.

6.2 Benchmarks

We trained our MLC schedulers on the 10 SPEC95 Fortran benchmarks. These benchmarks varied in size, total number of basic blocks, and average basic block length. Table 1 lists the benchmarks we used and their attributes. We derive the total cost of the benchmark by summing the costs of its basic blocks. The cost of each basic block is the number of times the basic block is executed (as obtained from the profile information) multiplied by the cost of the schedule of the basic block (obtained from the scheduler).

Benchmark	Description	Source lines	Number of blocks	Number of instructions	Average size of block (instructions)
FORTRAN programs					
110.applu	Parabolic and elliptic partial differential equations	3 817	25 475	129 853	5.097
141.apsi	Solves for the mesoscale and synoptic variations of potential temperature, wind, velocity, and distribution of pollutants	4 211	29 077	159 482	5.485
145.fpppp	Quantum chemistry	2 122	25 693	132 139	5.143
104.hydro2d	Astrophysics: hydrodynamical Navier-Stokes equations are solved to compute galactical jets	2 522	26 789	129 568	4.837
107.mgrid	Multi-grid solver in a 3D potential field	368	25 555	121 750	4.764
103.su2cor	Quantum physics: Monte Carlo calculation of elementary particle masses	1 614	26 972	135 837	5.036
102.swim	Shallow water model with 512×512 grid	259	25 109	119 333	4.753
101.tomcatv	A mesh-generation program	107	23 856	117 515	4.926
125.turb3d	Simulates isotropic, homogeneous turbulence in a cube	1 280	26 285	127 884	4.865
146.wave5	Plasma physics: solves Maxwell's equations and particle equations of motion on a Cartesian mesh with a variety of field and particle boundary conditions	6 430	28 932	152 655	5.276
Total		22 730	263 743	1 326 016	5.028

Table 1: Properties of SPEC95 Fortran benchmarks.

6.3 Experimentation with Different Feature Sets

For our first experiments, we decided to choose an initial set of thirty-four black, gray, and white box features, as shown in Table 2. We used this set of features to experiment with different learning parameters of neural networks, such as the algorithm used to update the network (we experimented with BackPropagation with momentum and Scaled Conjugate Gradient), the step size, the topology of the network, and weighted versus unweighted training. After several experiments varying these parameters we realized we would not get the desired performance from this feature set. Some results of these experiments are presented in Section 7.1. Instead of continue the experiments of different sets of features, we opted for another approach.

Iterating over different sets of features, until a good feature set was found, although possible, would have been extremely time-consuming. We decided, instead, to study the Digital scheduler to identify the kinds of features that might be required to build a good scheduler. A set of features (presented in Table 3) and a predicate on those features was developed by a colleague in the Object

Feature Name	Description	Hardware Dependence
Features Independent of Candidate Instructions		
Cost So Far	The number of cycles that would be taken by the current partial schedule.	white
Available Fu Integer Pipeline	Is the Integer functional unit available?	white
Available Fu Floating Pipeline	Is the floating point functional unit available?	white
Available Fu Memory Pipeline	Is the memory functional unit available?	white
NumberScheduled	Number of instructions already scheduled.	black
Number Remaining	Number of instructions remaining to be scheduled.	black
Features Dependent on Candidate Instructions		
Can Dual Issue With Last	Can the proposed instruction be issued as the second instruction of a pair, where the first instruction of the pair is the last scheduled instruction?	white
Needs To Wait	If the proposed instruction follows the current schedule, will it have to wait to issue?	white
Result Avail Delay Min	Result availability delay – max and min (the actual delay depends on the instruction which uses the result)	white
Instruction Class	The class of proposed instruction	grey
Class Of Prior Instruction	The class of the previously scheduled instruction.	grey
Class Of Most Recent Pred	The class of the most recently issued (immediate) predecessor	grey
Num Of Imm Succs	The number of directly dependent instructions.	black
Num Of Trans Succs	The number of directly or indirectly dependent instructions.	black
Num Of Imm Preds	Number of (immediate) instructions this instruction is dependent on.	black
Num Of Trans Preds	Number of (not only immediate) instructions this instruction is dependent on.	black
Critical Path	The height of the instruction in the DAG (the length of the longest chain of instructions dependent on this one) (edges with unit weights)	black
Dist From Last Pred Issued	How far back is the closest instruction on which this instruction depends?	black
Last Pred Issued Long Ago	Was the last scheduled predecessor issued longer ago than it's estimated latency?	black
Num Dominated Imm Succs	Number of dominated nodes among the immediate successors	black

Table 2: Features from the first feature set choosen.

Systems Laboratory, which exactly expressed the Digital scheduling algorithm. As verification that these features and the predicate were expressing the same information in the Digital scheduler, all the benchmarks were scheduled with the new predicate, and an exact agreement to the Digital scheduler was made for all benchmarks. These features were thus verified to be extremely predictive features. Any results returned by a scheduler using a MLC trained over these features would represent a lower bound on the performance we could expect. Both neural networks and decision trees did extremely well scheduling instructions using these features, with decision trees achieving the best performance.

6.4 Comparison of Different Schedulers

We measured the cost of the schedules produced by MLC schedulers along with the costs schedules produced by three additional schedulers: the original scheduler integrated in the compiler which compiled the benchmark (Original), a pubilc domain scheduler provided by Digital (PDS), which represents an excellent, but expensive, scheduler, and a random scheduler, that chose among instruction candidates at random, representing the extreme of a naive scheduler. The PDS provided by Digital usually outperforms all the other schedulers (even the scheduler they provide in their

Heuristic Name	Heuristic Description	Intuition for Use
Odd Partial	Is the current number of instructions scheduled odd or even?	If Odd Partial is TRUE, we're interested in scheduling instructions which can dual issue with last instruction.
Any Actual Dual	The number of instructions that can dual issue with the previously scheduled instruction	If Any Actual Dual > 0 and Odd Partial is true, we would like to schedule the next the instruction that can dual issue.
Max Rank	The instructionś rank based on its critical path and Digitalś heuristic weighting function.	Instructions with larger critical paths should be scheduled first. These instructions affect the lower bound of the schedule cost
Actual Dual	Can the instruction dual issue with last schedule instruction	If Odd Partial is true, it is important that we find an instruction that can issue in the same cycle (if there is one) with the last scheduled instruction.
Max Fu Data Delay	The earliest cycle when instruction can be issued without having to wait on its input data and its functional unit: relative to current cycle	We want to schedule instructions that will have their data and funtional unit available earliest.
Max Fu Data Delay Adv	The earliest cycle when instruction can be issued without having to wait on its input data and its functional unit: relative to current cycle advanced by one	We want to schedule instructions that will have their data available and functional unit earliest.

Table 3: Digital predicate features used for learning.

commercial compiler[1]) used in our experiments, therefore all scheduling costs presented in Section 7 are percentages of scheduling cost, worse than this scheduler.

7 Results

We now present results of several experiments of schedulers using the heuristics generated by CANNs (a CANN scheduler) and DT (a DT scheduler) trained on different features sets. Given that DT schedulers outperform NN schedulers, we continued our experimentation with DT schedulers and present results from these experiments. We present the results of an experiment which show that a DT scheduler is able to achieve good scheduling results with a small training set size (1% of the original training data). We also present results of a DT scheduler trained with small sets of features. To conclude this section, we present results of an experiment using only black and gray features. These results seem to indicate that any "good" feature set will require some white box features.

7.1 Results with First Set of Features

Table 4 shows the results of scheduling with MLCs trained on our first set of features. This table presents costs of scheduling with the scheduler in the compiler that originally compiled the benchmarks (Original), a CANN scheduler using both a weighted (W) and unweighted (U) training scheme, a decision tree scheduler using an unwieghted training scheme (DT), and a random scheduler (Random). The results presented are percentage differences from the scheduling cost of

[1]The PDS provided by Digital uses expensive simulation during its scheduling of instruction, therefore it is not surprising that it can outperform a scheduler in a compiler which uses less expensive heuristics (and therefore less accurate) to be extremely efficient.

Benchmark	Original	CANN (W)	CANN (U)	DT	Random
110.applu	2.01%	10.46%	16.84%	26.63%	24.79%
141.apsi	0.55%	8.20%	13.88%	19.43%	12.82%
145.fpppp	7.33%	28.92%	43.14%	54.97%	33.95%
104.hydro2d	0.19%	8.51%	10.74%	21.08%	15.22%
107.mgrid	0.23%	6.62%	88.98%	29.21%	45.06%
103.su2cor	1.23%	22.68%	5.90%	27.70%	24.75%
102.swim	0.72%	36.54%	10.06%	68.07%	41.57%
101.tomcatv	-0.07%	55.52%	22.30%	60.96%	30.56%
125.turb3d	0.45%	21.76%	32.65%	47.38%	30.45%
146.wave5	4.04%	14.36%	4.04%	24.95%	27.52%
Overall Avg	1.67%	21.36%	24.85%	38.04%	28.67%

Table 4: Results for benchmarks using Original, DT, CANN, and Random Schedulers on the first set of features. The results presented correspond to the percentage difference from the PDS.

the PDS. The first attempt to do instruction scheduling using this set of features failed to produce schedules that would satisfy a compiler-writer. We considered two main reasons for this:

- There are inherent limitations of the MLC used, which make it incapable of learning to make the right decisions.
- The information presented to the MLC is not sufficient; that is, the features do not contain enough information and/or the set of training patterns is too small or too sparse to allow generalization.

We suspected that the main problem was due to an inadequate feature set. We, therefore, set out to identify a "near-perfect" feature set before experimenting with other feature sets.

7.2 Results with Digital Predicate Features

A perfect set of features is one that allows a deterministic decision to be made, so as to produce an optimal schedule. Given the nature of the problem (it is *#P*-complete) this is infeasible. A near-perfect set, however, would allow producing a near-optimal schedule. Some experimentation [11] showed that the schedules produces by the PDS were near-optimal, and therefore the heuristics used by this scheduler are good. We examined the algorithm of the PDS and derived a set of features, we called Digital predicate features (listed in Table 3), all of which were white-box features. These features, when used by a deterministic predicate (see Algorithm 1), produced exactly the same schedules as the PDS. We then investigated whether the workings of this predicate could be approximated by a MLC. As was mentioned previously, the Digital predicate features were used to find an optimal configuration for our CANN with which to experiment. The results of these experiments are presented in Section 7.2.1. Cross-validation was then performed on this feature set using both decision trees and our optimal CANN configuration. These results are presented in Section 7.2.2.

7.2.1 Self-Validation Results with Digital Predicate Features

Considering the number of parameters from which to choose from when using neural networks, using all possible configurations for experimentation was impossible. We decided to perform experiments with our Digital predicate features using self-validation (as described in Section 6.1)

to identify a good configuration for our CANN. This configuration would then be used for further experiments. We experimented with two learning algorithms (SCG and BPM), three different numbers of hidden units (10, 20, and 30), and with weighted and unweighted training patterns. Because of the time required to run BPM and given the fact we did not observe an increase in performance over SCG, we did not run any experiments with weighted BPM, nor did we experiment with using BPM on a network of 30 hidden units. Table 5 shows the results for the self-validation experiments that we ran. The table shows that a CANN network using the SCG algorithm, with a configuration of 20 hidden units, trained on unweighted inputs outperforms all other configurations. We used this CANN configuration for further studies.

Benchmark	BPM H10 U	BPM H20 U	SCG H10 U	SCG H20 U	SCG H10 W	SCG H20 W	SCG H30 W
110.applu	20.83%	1.89%	4.74%	1.57%	0.46%	0.96%	3.10%
141.apsi	2.94%	0.22%	2.37%	0.21%	17.90%	2.89%	1.35%
145.fpppp	14.67%	6.60%	2.89%	1.68%	43.03%	5.66%	3.93%
104.hydro2d	2.47%	1.64%	0.86%	0.90%	4.87%	3.21%	2.32%
107.mgrid	1.14%	0.69%	2.47%	1.31%	3.82%	12.09%	1.97%
103.su2cor	0.59%	0.58%	0.25%	-0.01%	0.83%	22.96%	15.00%
102.swim	5.02%	1.43%	8.60%	6.44%	5.74%	0.72%	2.88%
101.tomcatv	1.04%	2.96%	5.51%	3.13%	50.55%	4.32%	69.76%
125.turb3d	4.66%	8.32%	4.00%	4.13%	17.24%	0.45%	23.31%
146.wave5	3.60%	4.02%	1.67%	1.59%	21.05%	5.52%	12.82%
Overall Avg	5.70%	2.84%	3.34%	2.10%	16.55%	5.88%	13.65%

Table 5: Self Validation Results for different CANN Schedulers. Each percentage is the percentage worse than the PDS from Digital. (H# = Number of Hidden units; U = Unweighted; W = Weighted)

7.2.2 Cross-Validation Results with Digital Predicate Features

The results in Table 6 were gathered from our cross-validation experiments, in order to obtain a realistic estimate of how MLC schedulers would perform in practice. Our cross-validation experiments used the ten Fortran programs from SPEC95 benchmarks, training on nine of the benchmarks and using the trained component to schedule the tenth. We experimented with decision trees and the best CANN configuration from our self-validation experiments. The results show that DT schedulers have a significant advantage over CANN schedulers. First, it seems that the CANN does not do well in the presence of filtering. It remains to be seen whether the CANNs can perform better, if more of the original data is used for training. However, there are close to 700 000 training instances in the original data set, therefore some amount of filtering is required. Given that the interaction of the heuristics in the public domain Digital scheduler can be expressed as a series of if-then-else tests (see Algorithm 1 in the Appendix), it is not hard reason why decision trees are able to outperform CANNs in this task. Decision trees are good at classification problems were the data can be differentiated with boolean tests, which is inherently how the DEC predicate works. If our problem required modeling a more complex function, a neural network configuration would be more adequate.

7.3 Further Decision Tree Experiments

We now present experiments on the reduction and combination of features, the effect of training with benchmarks from one language and scheduling a benchmark from another language, and the

Benchmark	Original	DT (1%)	DT (15%)	CANN (1%)	CANN (15%)	Random
110.applu	0.05%	1.85%	0.66%	4.76%	2.96%	14.30%
141.apsi	0.55%	0.87%	0.16%	36.99%	3.50%	12.82%
145.fpppp	7.33%	2.65%	1.10%	23.96%	6.97%	33.95%
104.hydro2d	0.19%	1.21%	0.09%	12.37%	2.31%	15.22%
107.mgrid	0.23%	4.75%	0.73%	47.86%	1.56%	45.06%
103.su2cor	1.23%	0.64%	0.22%	15.57%	4.63%	24.75%
102.swim	0.72%	1.43%	1.43%	68.09%	4.31%	41.57%
101.tomcatv	-0.07%	2.52%	0.15%	3.49%	2.45%	30.56%
125.turb3d	0.45%	1.46%	0.38%	26.31%	15.06%	30.45%
146.wave5	4.04%	1.90%	0.45%	5.26%	68.56%	27.52%
Overall Avg	1.47%	1.93%	0.54%	35.70%	11.23%	27.62%

Table 6: Results for benchmarks using Original, DTs, CANNs, and Random Schedulers. The percent in the parenthesis indicates the amount of data from the original data files. The CANN has 20 hidden units, is trained using unweighted instances, and uses the SCG algorithm.

performance of scheduling using only black and gray features.

7.3.1 Results with Scheduling C benchmarks

We decided to verify early results which indicated that good scheduling required training a MLC with a corpus of benchmarks of the same language as the benchmark we were scheduling. We scheduled the 8 SPEC95 C benchmarks using a MLC trained with the corpus of Fortran benchmarks. Table 7 and Table 1show that Fortran and C benchmarks have different characteristics. Also, the C and Fortran benchmarks were compiled with different compilers which creates further differences between the languages. Given these differences, we initially believed that scheduling and training would have to be done on benchmarks of the same language. However, the results presented in Table 8 indicate that this assumption was incorrect. We hypothesize that given a good feature set, a corpus made up of benchmarks of language X can be used to train a MLC for scheduling any other language Y, where X and Y can be any compiled language. Further experimentation remains to prove whether this hypothesis holds.

7.3.2 Results with Reducing and Combining Features

In order to identify a minimal set of features required by a good MLC scheduler for the Alpha 21064, we decided to experiment with small subsets of the Digital-specific feature set. We experimented with the following features (Max Rank, Actual Dual, Max Fu Delay Delay). This resulted in a feature vector of six features, three features for each instruction. To further reduce this feature set we combined the like features of the two instructions being compared (i.e., we combine two features by subtracting one feature from another feature) to obtain a feature vector of three features. The results of scheduling with decision trees using both feature sets are presented in Table 9 (columns DT(6) and DT(3)). The results show that decision trees trained on both features sets performed extremely well. The feature set with the combined features (DT(3)) performed slightly better than the feature set with uncombined features (DT(6)). When combining features, we are explicitly making comparisons between features where the comparison is relevant, instead of hav-

Benchmark	Description	Source lines	Number of blocks	Number of instructions	Average size of block (instructions)
C programs					
129.compress	Reduces the size of files using adaptive Lempel-Ziv coding	1 422	4 596	20 152	4.385
126.gcc	based on the GNU C compiler version 2.5.3, builds SPARC code	133 049	77 269	332 184	4.299
099.go	Artificial intelligence: plays the game of *go*	25 362	16 095	80 900	5.026
132.ijpeg	Graphic compression and decompression	17 449	12 033	70 928	5.894
130.li	LISP interpreter running the Gabriel benchmarks	4 323	8 056	36 668	4.552
124.m88ksim	Motorola 88100 microprocessor simulator: runs test program	12 026	10 121	46 438	4.588
134.perl	Manipulates strings (anagrams) and prime numbers in Perl	21 078	22 590	111 849	4.951
147.vortex	Subset of a full object oriented database program called VORTEx (Virtual Object Runtime EXpository)	41 034	32 624	180 331	5.528
Total		255 743	183 384	879 450	4.902

Table 7: Properties of SPEC95 C benchmarks.

Benchmark	Original	DT	Random
129.compress	3.93%	0.02%	10.71%
126.gcc	3.94%	0.00%	16.79%
099.go	2.40%	0.01%	10.01%
132.ijpeg	0.71%	0.31%	17.93%
130.li	3.42%	0.00%	12.32%
124.m88ksim	4.99%	0.00%	11.16%
134.perl	2.70%	-0.24%	12.59%
147.vortex	1.67%	0.02%	14.06%
Overall Avg	2.97%	0.02%	13.20%

Table 8: Results for scheduling C benchmarks using Original, Decision Tree (DT), and Random Schedulers. Each percentage is the amount worse than the PDS.

ing the decision tree algorithm try and derive comparisons from the training data. Thus, we have simplified the problem by manipulating the training data based on *a priori* knowledge.

7.3.3 Results with Black and Gray Features

We also ran an experiment using only black and gray features. First, we identified a set of black and gray features, which we believed would be effective in describing and therefore in differentiating instructions. We also choose black and gray features which we believed would do a reasonably good job of modeling the current state of the architecture during scheduling. These features seemed sufficient to allow a MLC to learn biases towards better instructions to schedule at given decision points. The features are presented in Table 10 and the results of scheduling using decision trees induced using these features are presented in Table 9 (column BG). These results were disappointing. Our scheduler did not perform better than random, and in fact for some benchmarks performs much worse. The results indicate that either we were unable to identify the predictive black and

Benchmark	Original	DT (6)	DT (3)	BG	Random
110.applu	0.05%	0.98%	0.34%	12.82%	14.30%
141.apsi	0.55%	0.24%	0.17%	11.43%	12.82%
145.fpppp	7.33%	1.24%	0.31%	38.69%	33.95%
104.hydro2d	0.19%	0.01%	0.01%	13.89%	15.22%
107.mgrid	0.23%	0.75%	0.04%	40.49%	45.06%
103.su2cor	1.23%	0.13%	-0.19%	18.05%	24.75%
102.swim	0.72%	1.43%	0.00%	50.17%	41.57%
101.tomcatv	-0.07%	1.33%	0.00%	37.35%	30.56%
125.turb3d	0.45%	1.58%	1.28%	49.74%	30.45%
146.wave5	4.04%	0.59%	0.17%	38.20%	27.52%
Overall Avg	1.47%	0.83%	0.21%	31.08%	27.62%

Table 9: Results for benchmarks using the DT schedulers on three different feature sets. The number in the parenthesis indicates the size of the feature set.

gray features needed for a good MLC scheduler or that MLC schedulers require some white box features to obtain schedule qualities comparable to hand tuned schedulers. The latter cause seems more likely, but further experiments are required to reinforce this hypothesis.

Feature Name	Feature Description	Hardware Dependence
Features Independent of Candidate Instructions		
Fraction Scheduled	Fraction of instructions scheduled (real in the range of [0,1])	black
Param Class Of Prior Instr	Classes of last N instructions scheduled. (N=4)	gray
Features Dependent on Candidate Instructions		
Instruction Class	The class of proposed instruction	gray
Param Depends On Prior Instr	Does candidate instruction depend on Nth prior instruction? Check last four scheduled instructions.	gray
Class Of Most Recent Pred	The class of the most recently issued (immediate) predecessor	gray
Num Of Imm Succs	The number of directly dependent instructions.	black
Num Of Trans Succs	The number of directly or indirectly dependent instructions.	black
Num Of Imm Preds	Number of (immediate) instructions this instruction is dependent on.	black
Critical Path	The height of the instruction in the DAG (the length of the longest chain of instructions dependent on this one) (edges with unit weights)	black
Dist From Last Pred Issued	How far back is the closest instruction on which this instruction depends?	black
Fractional Height	Fractional height of the instruction in the longest dependence chain running from top to bottom of the DAG.	black
Num Dominated Imm Succs	Number of dominated nodes among the immediate successors	black
Reachability Count	The number of insructions that can be reached from this instruction in the DAG.	black

Table 10: Black and Gray Features used for Decision Tree Experiment.

8 Related Work

Alessandro De Gloria and Paolo Faraboschi suggest a method of code compaction on very long instruction word (VLIW) architectures using Boltzmann machines [12]. Code compaction in VLIW architectures is the problem of assigning instructions into a minimum number of large instruction

packets (each packet corresponds to a very long instruction word, hence the name VLIW). The problem of code compaction has many of the same properties as instruction scheduling. Boltzmann machines are a class of neural networks that use simulated annealing in their learning procedure [13]. They look at compacting code for seven basic blocks ranging in size from 13 to 39. The show by performing several hundred annealing iterations, that they are able to achieve performance equivalent to "hand-compiled" code. Unfortunately, like simulated annealing, this methods is very computation intensive, requiring many iterations over a training set before the learning procedure converges. Another problem with the scheduling technique introduced in the paper is that it is allowed to schedule instructions arbitrarily without regard to data dependency constraints. Therefore, along with valid schedules the technique can also produce many invalid schedules. Their solution is to check the validity of the schedule with the original DDD, but this adds costly superfluous computation. Also, a Boltzmann machine must be retrained for each different basic block that is being scheduled. To alleviate the problem of the expensive learning procedure, the authors suggest the possibility of having many Boltzmann machines implemented in hardware and scheduling different basic blocks concurrently.

Beaty applied genetic algorithms to the problem of instruction scheduling [14]. He presented a method in which a genetic algorithm is allowed to choose the order of instructions to be placed in the schedule. The genetic algorithm trains and schedules simultaneously, and it must be retrained for every new block. Beaty shows that genetic algorithms are able to schedule code as well or better than existing scheduling methods on a few benchmarks. Like Boltzmann machines, genetic algorithms are prohibitively expensive requiring several hundred iterations before a good schedule is found. This method also produces invalid schedules which adds to its high cost.. Given the high computation cost of genetic algorithms and the large number of iterations needed to schedule a block, this is not a viable solution for commercial compilers. In contrast to these methods, we do not do online learning; we train our neural network and induce our decision trees before scheduling and the MLCs produced remain fixed when scheduling, i.e., they are not updated based on any feedback that might be obtained during scheduling. A consequence of this is that our technique is extremely efficient and a viable solution for commercial compilers. Also, our method does not produce invalid schedules. Our MLC schedulers use the DDD at all times during scheduling and therefore all schedules they produce are valid.

9 Future Work

There are several areas for promising research pertaining to this work. We would eventually like to experiment with different architectures, both existing and proposed. We would like to experiment with complicated architectures, for example, architectures with additional functional units or more complicated pipelines or different architectures such as VLIW architectures or multiprocessors. Our experiments are currently running on a simulator for a Alpha 21064. This architecture has only two functional units, a floating point and a integer functional unit, thus a hand-coded instruction scheduler may be able to schedule well for it. More complicated architectures will be harder to schedule for and therefore our scheme may prove more beneficial. An experimental tool is being built to let us experiment quickly with simulators of different architectures. These simulators will be required to return values of features corresponding to their internal state, thus providing essential information to build good schedulers. Many of these features being used may be too

computationally expensive for practical use, therefore future work remains on experimenting with features that are cheaper to compute, but are reasonable approximations of the expensive features.

We would like to be consider harder problems, such as global scheduling or scheduling with register allocation. These problems are hard enough that hand-tuned heuristics may not be adequate. Therefore our learning techniques may be able to show significant improvement over them. Looking at harder problems might preclude working in a supervised learning setting. These problems might therefore require reinforcement learning techniques. In reinforcement learning, we can let the learner schedule and give it a quality rating for the final schedule. The learner is rewarded for good schedules and punished for bad schedules, and thus learns to prefer actions it has tried in the past and found effective in producing a reward.

It is unclear whether we will see similar levels of performance scheduling different languages. Therefore, we would like to investigate scheduling over different languages, such as C++, Modula-3, Java, and Ada. This would allow us to perform cross-validation over a larger training set, that is training using multiple benchmarks from multiple languages. It may be that our technique proves more beneficial for some languages than for others. For instance, simple heuristics found in commercial schedulers may be able to schedule Fortran programs better than C++ code. This may be due to the types of programs for which a certain language is used, the constructs in a language, or the coding style a language induces.

Our schedule costs are *approximations* of what should happen in hardware, and therefore, we cannot be completely sure what our results would be if we were running on hardware, or at least a better simulator (e.g., one that modeled the cache hierarchy and therefore modeled "global" effects on a basic block). To this end, we would like eventually to integrate our scheduler into the back end of a compiler or at the very least, into a simulator that modeled more precisely what was happening in hardware.

10 Conclusion

We have shown that integrating a machine learning component into an instruction scheduler is not only feasible, but extremely profitable. The results show that an decision tree scheduler schedules programs comparable to an extremely good hand-coded scheduler given the right set of features. We have also shown that schedulers built with decision trees typically outperform schedulers built with neural networks. Given the nature of the task, that is, to learn an appropriate predicate for scheduling, decision trees seem more suited to the task.

References

[1] Digital Equipment Corporation, Maynard, MA, *DECchip 21064-AA Microprocessor Hardware Reference Manual*, first ed., Oct. 1992.

[2] J. A. Fisher, "Trace scheduling: A technique for global microcode compaction," *IEEE Trans. Computers*, vol. C-30, pp. 478–490, July 1981.

[3] A. Nicolau, "Percolation scheduling: A parallel compilation technique," Tech. Rep. TR85-678, Cornell University, Computer Science Department, May 1985.

[4] Brightwell and Winkler, "Counting linear extensions is #P-complete," in *STOC: ACM Symposium on Theory of Computing (STOC)*, 1991.

[5] H. G. Baker, "Precise instruction scheduling without a precise machine model," *ACM Computer Architecture News*, vol. 19, pp. 4–8, Dec. 1991.

[6] A. Zell, N. Mache, T. Sommer, and T. Korb, "Design of the SNNS neural network simulator," in *Proc. "OGAI-91, Seventh Austrian Conf. on Artificial Intelligence"*, Sept. 1991.

[7] G. Tesauro and T. J. Sejnowski, "A parallel network that learns to play backgammon," *Artificial Intelligence*, vol. 39, pp. 357–390, July 1989.

[8] B. W. Wah, P. Mehra, and C.-C. Teng, "Comparator neural network for dynamic prediction," in *Proceedings of the 2nd International Symposiumn on Neural Networks.*, pp. 571–580, 1994.

[9] J. R. Quinlan, *C4.5: Programs for Machine Learning*. Morgan Kaufmann, 1992.

[10] P. E. Utgoff, "Decision tree induction based on efficient tree restructing," Tech. Rep. 95-18, Department of Computer Science, University of Massachusetts, Amherst, MA, Mar. 1995.

[11] D. Stefanovic, "The character of the instruction scheduling problem." Unpublished working paper, Mar. 1997.

[12] A. D. Gloria and P. Faraboschi, "A Boltzmann machine approach to code optimization," *Parallel Computing*, vol. 17, pp. 969–982, December 1991.

[13] S. Haykin, *Neural Networks, A Comprehensive Foundation*. New York, NY: Macmillan, 1994.

[14] S. J. Beaty, *Instruction Scheduling using Genetic Algorithms*. PhD thesis, Colorado State University, Fort Collins, Colorado 80523, Oct. 1991.

11 Appendix

```
if (odd_partial) {
    if (any_actual_dual = yes) {
       if (actual_dual == left) {
          if (actual_dual == right)   {
             if (max_rank == left) {
                  decision = True;
             }
             else {
               decision = False;
             }
           else   {
            decision = True;
           }
       }
        else {   /* left can't dual */
          if (actual_dual == right )   {
              decision = False;
           }
          else   {
             decision = True;
           }
        }
     }
    else  {  /* No instructions can dual issue */
         if (max_fu_data_delay_adv == right) {
             decision = True;
        }
        else if (max_fu_data_delay_adv == left) {
             decision = False;
        }
        else {
          if (max_rank == left) { /*left on critical path*/
             decision = True;
          }
         else { /*right on critical path*/
            decision = False;
          }
      }
     }
}
else {
  if (max_fu_data_delay == right)  {
       decision = True;
   }
  else if (max_fu_data_delay == left) {
      decision = False;
   }
  else  {
      if (max_rank == left) { /*left on critical path*/
          decision = True;
      }
      else {
          decision = False; /*right on critical path*/
      }
  }
}
```

Algorithm 1: The algorithm for the Digital predicate. If decision is set True, we prefer the first instruction, otherwise we prefer the second instruction.

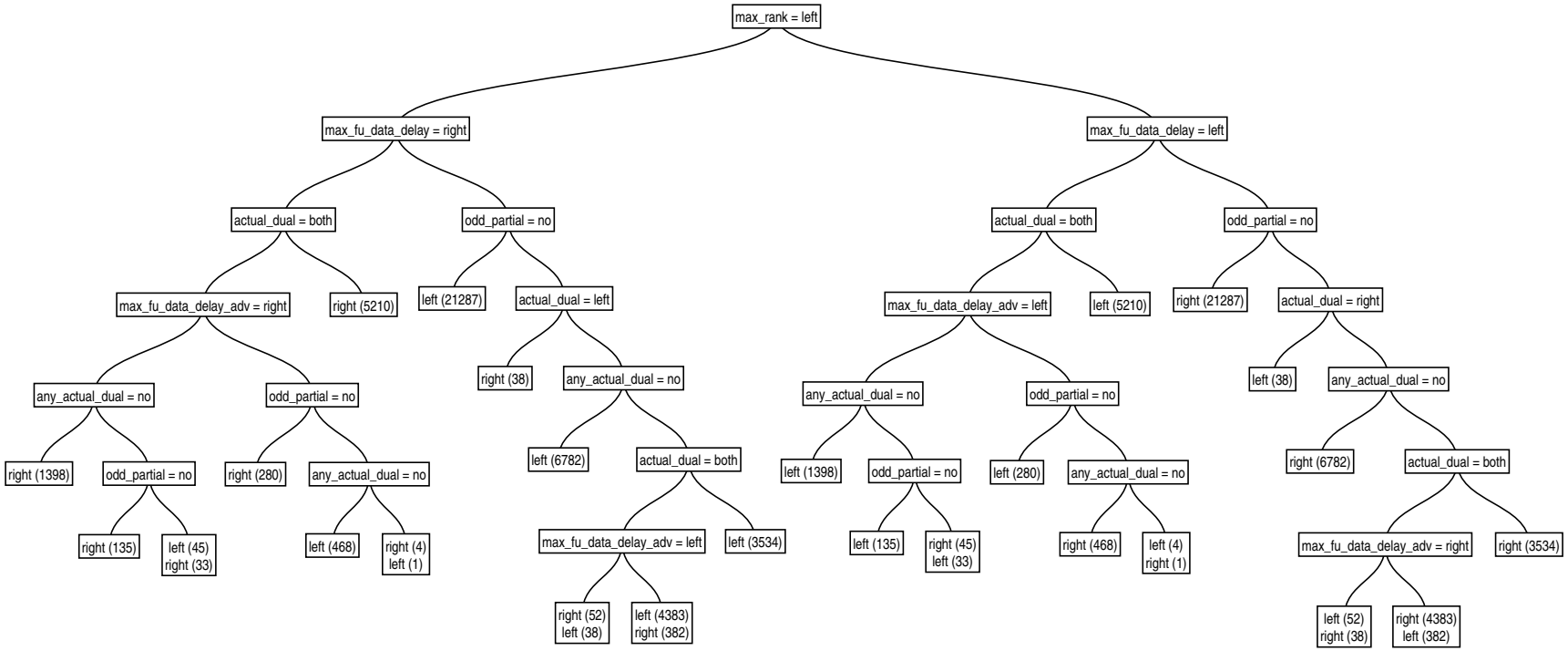


Figure 6: A decision tree induced by the ITI using the Digital predicate features. The features have been converted from their numeric form to labeled values.