Scheduling Straight-Line Code Using Reinforcement Learning and Rollouts

Amy McGovern, Eliot Moss, and Andrew G. Barto
$\{amy|moss|barto@cs.umass.edu\}$
Department of Computer Science
University of Massachusetts, Amherst
Amherst, MA 01003

June 4, 1999

Abstract

The execution order of a block of computer instructions on a pipelined machine can make a difference in its running time by a factor of two or more. In order to achieve the best possible speed, compilers use heuristic schedulers appropriate to each specific architecture implementation. However, these heuristic schedulers are time-consuming and expensive to build. We present empirical results using both rollouts and reinforcement learning to construct heuristics for scheduling basic blocks. In simulation, the rollout scheduler outperformed a commercial scheduler, and the reinforcement learning scheduler performed almost as well as the commercial scheduler.

Keywords: Reinforcement learning, Instruction scheduling, Rollouts

1 Motivation

Although high-level code is generally written as if it were going to be executed sequentially, most modern computers exhibit parallelism in instruction execution using techniques such as the simultaneous issue of multiple instructions. Often pipelines within a machine differ in functionality. For example, one pipeline may be specialized for floating-point operations and one for integer operations. In order to take the best advantage of multiple pipelines, when a compiler turns the high-level code into machine instructions, it employs an instruction scheduler to reorder the machine code. The scheduler needs to reorder the instructions in such as way as to preserve the original in-order semantics of the high level code while having the reordered code execute as quickly as possible. An efficient schedule can produce a speedup in execution of a factor of two or more.

Building an instruction scheduler can be an arduous process. Schedulers are specific to the architecture of each machine and the general problem of scheduling instructions is NP-Hard (Proebsting). Because of these characteristics, schedulers are currently built using hand-crafted heuristic algorithms. However, this method is both labor and time intensive. Building algorithms to select and combine heuristics automatically using machine learning techniques can save time and money. As computer architects develop new machine designs, new schedulers would be built automatically to test design changes rather than requiring hand-built heuristics for each change. This would allow architects to explore the design space more thoroughly and to use more accurate metrics in evaluating designs.

A second possible use of machine learning techniques in instruction scheduling is by the end user. Instead of scheduling code using a static scheduler trained on benchmarks when the compiler was written, a user would employ a learning scheduler to discover important characteristics of that user's code. The learning scheduler would exploit the user's coding characteristics to build schedules better tuned for that user.

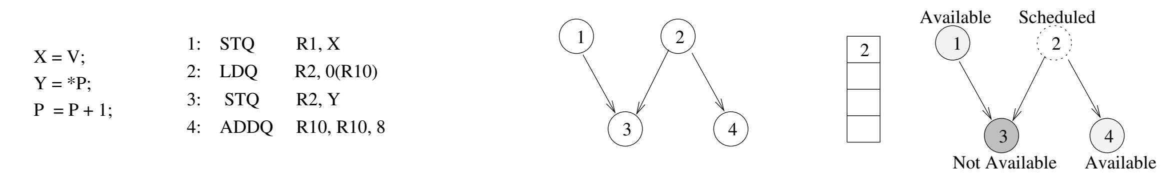
With these motivations in mind, we formulated and tested two autonomous methods of building an instruction scheduler. The first method used rollouts (Tesauro and Galperin, 1996, Bertsekas, et al.,1997a,b) and the second focused on reinforcement learning (RL) (Sutton & Barto, 1998). Both methods were implemented for the Digital Alpha 21064. The next section gives a domain overview and discusses results using supervised learning on the same task.

2 Domain overview

We focused on scheduling *basic blocks* of instructions on the 21064 version (DEC, 1992) of the Digital Alpha processor (Sites, 1992). A basic block is a set of machine instructions with a single entry point and a single exit point. By definition, a basic block does not contain any branches or loops. Our schedulers can reorder the machine instructions within a basic block but cannot rewrite, add, or remove any instructions. Many instruction schedulers built into compilers can insert or delete instructions such as no-ops. However, we were working directly with the compiled object file and had no method for changing the size of a block within the object file, only for reordering the block.

The goal of the scheduler is to find a least-cost valid ordering of the instructions in a block. The cost is defined as the (estimated) execution time of the block. A valid ordering is one that preserves the semantically necessary ordering constraints of the original code. These constraints are that potentially conflicting reads and writes (of both registers and memory) are not reordered. We insure validity by creating a dependency graph that directly represents the necessary ordering relationships in a directed acyclic graph (DAG). The task of scheduling is to find a least-cost total ordering consistent with the block's DAG.

Figure 1 gives a sample basic block and its DAG. In this example, the original high-level code is three lines of C code (Figure 1a). This C code is compiled into the assembly code shown in Figure 1b. The DAG in Figure 1c is constructed from Figure 1b in the following way. Instructions 1 and 2 have no dependencies and thus have no arcs into their nodes. However, instruction 3 reads from a register loaded in instruction 2 which means that instruction 2 must be executed before instruction 3. Also, if the compiler does not know that X and Y are non-overlapping variables, it must assume instruction 3 depends on instruction 1. Lastly, instruction 4 increments the value in the same register that instruction 2 reads, which means that instruction 4 must be executed after instruction 2. An example of using this DAG is shown by the partial schedule in Figure 1d. Here, instruction 2 has already been scheduled. This means that instructions 1 and 4 are available to be scheduled next. Instruction 3 must wait for instruction 1 to be scheduled before it can become available.



(a) C Code (b) Instruction Sequence to be Scheduled (c) Dependence Dag of Instructions (d) Partial Schedule

Figure 1: Example basic block code, DAG, and partial schedule

Using this example basic block (Figure 1) and a single pipeline machine, we can demonstrate how a small schedule change can affect the running time of a program. Assume that the schedules in Table 1 were to execute on a single pipeline machine where all instructions except loads took one cycle to execute and loads took two cycles to execute. Using these four instructions, we can create two valid schedules as shown in the table. When executing schedule A, instruction 2 is executed immediately before instruction 3. However, instruction 3 depends on the results of instruction 2, which takes two cycles to execute. This causes a pipeline stall for one cycle. Schedule B fills the second cycle of the load in the pipeline with instruction 4. Instruction 4 does not cause a stall because it has no dependency on the data being loaded in instruction

2. With the stall, Schedule A takes 5 cycles to execute while Schedule B takes only 4 cycles. This example illustrates differing execution times with only a small change in instruction ordering. On larger blocks and machines with multiple pipelines and with the ability to issue multiple instructions per cycle, the difference can be more dramatic.

Original Schedule		**Schedule A**	**Cycle count**		**Schedule B**	**Cycle count**
1. STQ	R1, X	Instr 1	1		Instr 1	1
2. LDQ	R2, O(R10)	Instr 2	2	← causes a stall	Instr 2	1
3. STQ	R2, Y	Instr 3	1		Instr 4	1
4. ADDQ	R10, R10, 8	Instr 4	1		Instr 3	1
Totals:			5			4

Table 1: Two valid schedules and running times for a single pipelined machines where loads take two cycles and all other instructions take one cycle.

The Alpha 21064 is a dual-issue machine with two different execution pipelines. One pipeline is more specialized for floating point operations while the other is for integer operations. Although many instructions can be executed by either pipeline, dual issue can occur only if the next instruction to execute matches the first pipeline and the second instruction matches the second pipeline. A given instruction can take anywhere from one to many tens of cycles to execute. Researchers at Digital have a publicly available 21064 simulator that also includes a heuristic scheduler for basic blocks. Throughout the paper, we will refer to this scheduler as *DEC*. The simulator gives the running time for a given scheduled block assuming all memory references hit the cache and all resources are available at the beginning of the block.

All of our schedulers used a greedy algorithm to schedule the instructions, i.e., they built schedules sequentially from beginning to end with no backtracking. This is referred to as list scheduling in compiler literature. Each scheduler reads in a basic block, builds the DAG, and schedules one instruction at a time until all instructions have been scheduled. When choosing the best instruction to schedule from a list of available candidates, each scheduler compares the current best choice with the next available instruction. The winner of the comparison moves onto the next comparison. After all candidates have been considered, the overall winner is scheduled and the DAG is updated to give a new set of candidate instructions. Our algorithms differ in the way in which instructions are compared at each step.

Moss, et al. (1997) showed that supervised learning techniques could induce excellent basic block instruction schedulers for this task. To do this, they trained several different supervised learning methods to choose the best instruction to schedule given a feature vector over the currently scheduled instructions and the candidate instructions. Each method learned a preference relation of choosing instruction i over instruction j at a given point in a partial schedule. The supervised learning methods used were the decision tree induction program ITI (Utgoff, Berkman, and Clouse, 1997), table lookup, the ELF function approximator (Utgoff and Precup, 1997), and a feed-forward artificial neural network (Rumelhart, Hinton, and Williams, 1986). More details of each method can be found in Moss, et al. (1997). They were able to show that each method nearly always made optimal choices (i.e. $96-97\%$ of the time) of instructions to schedule within blocks of size 10 or less.

Although all of the supervised learning methods performed quite well, they shared several limitations. Supervised learning requires exact input/output pairs. Generating these training pairs requires an optimal scheduler that searches every valid permutation of the instructions within a basic block and saves the optimal permutation (the schedule with the smallest running time). However, this search was too time-consuming to perform on blocks with more than 10 instructions, because optimal instruction scheduling is NP-hard (Stefanović, 1997 and Proebsting). This inhibited the methods from learning using larger blocks. Using a semi-supervised method such as RL or rollouts does not require generating training pairs, which means that the method can be applied to larger basic blocks and can be trained without knowing optimal schedules.

In order to test each scheduling algorithm, we used the 18 SPEC95 benchmark programs. Ten of these programs are written in FORTRAN and contain mostly floating point calculations. Eight of the programs are written in C and focus more on integer, string, and pointer calculations. Each program was compiled using the commercial Digital compiler at the highest level of optimization. We call the schedules output by the compiler *ORIG*. This collection has 447,127 basic blocks, containing 2,205,466 instructions. Although 92.34% of the blocks in SPEC95 have 10 instructions or less, the larger blocks account for a disproportionate amount of the running time. The small blocks account for only 30.47% of the overall running time. This may be because the larger blocks are loops that the compiler unrolled into extremely long blocks. By allowing our algorithms to schedule blocks whose size is greater than 10, we focus on scheduling the longer running blocks. We present timing results using the simulator.[1]

3 Rollouts

Rollouts are a form of Monte Carlo search, first introduced by Tesauro and Galperin (1996) for use in backgammon. Bertsekas, et al. (1997a,b) explored rollouts in other domains and proved important theoretical results. In the instruction scheduling domain, rollouts work as follows: suppose the scheduler comes to a point where it has a partial schedule and a set of (more than one) candidate instructions to add to the schedule. For each candidate, the scheduler appends it to the partial schedule and then follows a fixed policy π to schedule the remaining instructions. When the schedule is complete, the scheduler evaluates the running time and returns. When π is stochastic, this rollout can be repeated many times for each instruction to achieve a measure of the expected outcome. After rolling out each candidate, the scheduler picks the one with the best average running time. This process is illustrated graphically in Figure 2. In this example, the scheduler used a stochastic rollout policy π. Each of the three instructions was rolled out five times. The third instruction had the lowest overall average running time of 5.6 cycles and is chosen as the next instruction to schedule. The process is then repeated at the next decision point.

[1]We can include actual timings in a revision, if desired. Time did not permit obtaining them before the deadline.

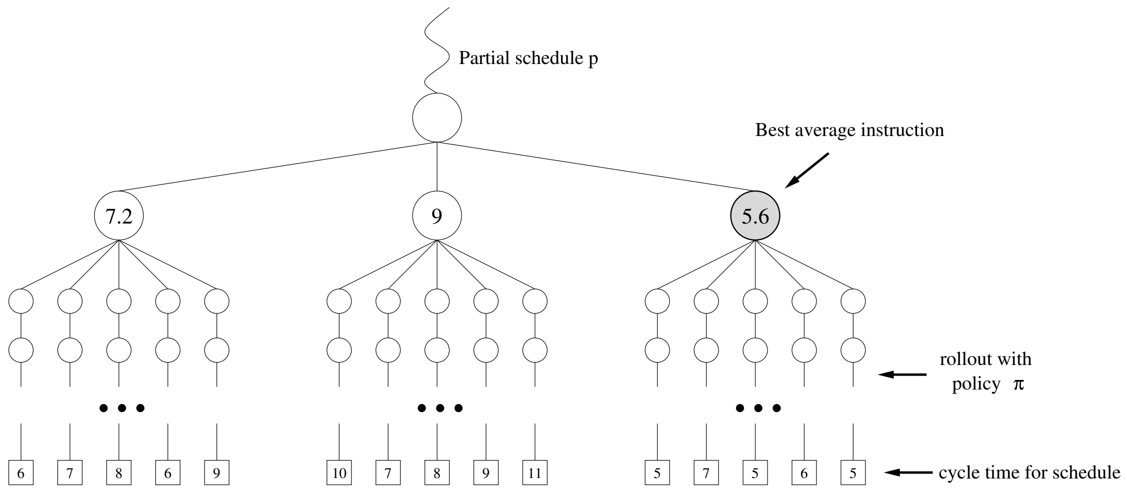


Figure 2: Diagram of the actions of a rollout scheduler when rolling out three different instructions five times each.

Our first set of rollout experiments compared three rollout policies π. While Bertsekas et al. (1997a,b) proved that if we used the DEC scheduler as π, we would perform no worse than DEC, an architect proposing a new machine might not have a good heuristic available to use as π, so we also considered policies more likely to be available. The obvious choice for an easily available rollout policy π is the random policy. We denote the use of the random policy for π in the rollout scheduler as *RANDOM*-π. Under this policy, the rollout makes all choices randomly. We also tested using heuristics for π in two ways. The first was the ordering produced by the optimizing compiler ORIG, denoted *ORIG*-π. The second heuristic policy tested was the DEC scheduler itself, denoted *DEC*-π. Although the full heuristics of DEC or ORIG may not be available to an architect while designing a machine, a simpler set of heuristics (which are more complicated than RANDOM) may be available. This allows us to understand how heuristics can help rollout schedulers.

The rollout scheduler performed only one rollout per candidate instruction when using ORIG-π and DEC-π because each is deterministic. Initially, we used 25 rollouts for RANDOM-π. Discussion of how the number of rollouts affects performance is presented in the next section. After performing a number of rollouts for each candidate instruction, we chose the instruction with the best average running time. As a baseline scheduler, we also scheduled each block with a valid but otherwise completely random ordering. A time analysis of a rollout scheduler shows that it takes $O(n^2m)$ where m is the number of rollouts and n is the number of instructions. A greedy scheduler with no rollouts takes only time $O(n)$. Because the running time increases quadratically with the number of instructions multiplied by the number of rollouts, we focused our rollout experiments on one program in the SPEC95 suite: *applu*. This program was written in Fortran and focuses on floating point operations.

Table 2 summarizes the performance of the rollout scheduler under each policy π as compared to the DEC scheduler on all 33,007 basic blocks of size 200 or less from applu. Because each basic block is executed a different number of times and is of a different size, measuring performance based on the mean difference in number of cycles across blocks is not a fair performance measure. To more fairly assess the performance, we used the ratio of the weighted execution time of the rollout scheduler to the weighted execution time of the DEC scheduler where the weights of each block are based on the number of times that block is executed. More concisely, the performance measure was:

$$\text{ratio} = \frac{\sum_{\text{all blocks}} (\text{rollout scheduler execution time} \times \text{number of times block is executed})}{\sum_{\text{all blocks}} (\text{DEC scheduler execution time} \times \text{number of times block is executed})}$$

This means that a faster running time than DEC on the part of our scheduler would give a ratio of less than one. All execution times in Table 2 are from the simulator.

Although Stefanović (1997) noted that rescheduling a block has no effect on 68% of the blocks of size 10 or less, the random scheduler performed very poorly. On applu, RANDOM was 31% slower than DEC. Overall, RANDOM is 36.9% slower than DEC. This number is a geometric mean across five runs of all 18 SPEC95 benchmark suites. A detailed summary of RANDOM's results on all suites in simulation can

Scheduler	**Ratio**
Random	1.3150
RANDOM-π	1.0560
ORIG-π	*0.9895*
DEC-π	*0.9875*

Table 2: Ratios of the weighted execution time of the rollout scheduler to the DEC scheduler. A ratio of less than one means that the rollouts outperformed the DEC scheduler. The entries in italics are those that outperformed DEC.

be found in the next section. Without adding any heuristics and just using rollouts with the random policy, RANDOM-π came within 5% of the running time of DEC. By using ORIG and DEC as π, the ORIG-π and DEC-π schedulers were able to outperform DEC. Both were about 1.1% faster than DEC. Although this improvement may seem small, the DEC scheduler is known to make optimal choices 99.13% of the time for blocks of size 10 or less (Stefanović, 1997). This 1.1% improvement is over all blocks of size 200 or less.

ORIG-π and DEC-π are deterministic policies, so the numbers reported above are only across one run. RANDOM-π and RANDOM were averaged geometrically across 5 runs. Each run of RANDOM-π on all blocks of 200 or less from applu took about six hours. This limited the number of runs we could perform.

Part of the motivation behind using rollouts in a scheduler was to obtain efficient schedules without spending the time to build a such precise heuristic as ORIG and DEC. With this in mind, we explored RANDOM-π more closely in a follow-up experiment.

Evaluation of the number of rollouts

This experiment considered how the performance of RANDOM-π varied as a function of the number of rollouts. To cover the possible space of the number of rollouts, we tested 1, 5, 10, 25, and 50 rollouts per candidate instruction. We also varied the metric for choosing among candidate instructions. Instead of always choosing the instruction with the best average performance, denoted AVG-RANDOM-π, we also experimented with selecting the instruction with the absolute best running time among its rollouts. We call this BEST-RANDOM-π. We hypothesized that selection of the absolute best path might lead to better performance overall because it meant selecting the first instruction on the most promising scheduling path. As before, we compared performance on all 33,007 basic blocks of size 200 or less from applu.

Figure 3 shows the performance of the rollout scheduler as a function of the number of rollouts. Performance is assessed in the same way as before: ratio of weighted execution times. Thus, the lower the ratio, the better the performance. Each data point represents the geometric mean over five different runs. Although one rollout may not be enough to fully explore a schedule's potential, performance of RANDOM-π with one rollout is 16% slower then DEC. This is a considerable improvement from RANDOM's performance of 31% slower than DEC on applu. By increasing the number of rollouts from one to five, the performance of AVG-RANDOM-π improved to within 10% of DEC. Improvement continued as the number of rollouts increased to 50 but performance leveled off around 5% slower then DEC. As the graph shows, the improvement per the number of rollouts drops off dramatically from 25 to 50.

Our hypothesis about BEST-RANDOM-π outperforming AVG-RANDOM-π was shown to be incorrect. Choosing the instruction with the absolute best rollout schedule did not yield any improvement in performance over AVG-RANDOM-π over any number of rollouts. We hypothesize that this is due to the stochastic nature of the rollouts. Once the rollout scheduler chooses an instruction to schedule, it repeats the rollout process again over the next set of candidate instructions. By choosing the instruction with the absolute best rollout, there is no guarantee that the scheduler will find that permutation of instructions again on the next rollout. When it chooses the instruction with the best average rollout, the scheduler has a better chance of finding a good schedule on the next rollout. The theory developed by Bertsekas,et al. (1997a,b) also predicts this answer.

Although the performance of the rollout scheduler can be excellent, rollouts are costly in time to run.

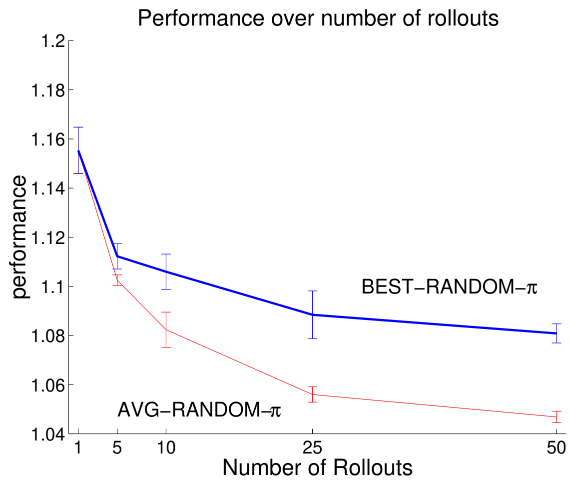


Figure 3: Performance of rollout scheduler with the random model as a function of the number of rollouts and the choice of evaluation function. The bars above and below each data point are one standard deviation from the mean.

As mentioned before, using 25 rollouts per block took over 6 hours to schedule one program. Although the majority of that time is spent in the simulator, a faster simulator will not be able to make the $O(n^2m)$ rollout scheduler perform as quickly as an $O(n)$ greedy scheduler. Unless the running time can be improved, rollouts cannot be used for all blocks in a commercial scheduler or in evaluating more than a few proposed machine architectures. However, because rollout scheduling performance is high, rollouts could be used to optimize the schedules on important (long running times or frequently executed) blocks within a program. With the performance and the timing of the rollout schedulers in mind, we looked to RL to obtain high performance at a faster running time. A RL scheduler would run in the $O(n)$ time of a greedy list scheduler.

4 Reinforcement Learning

Reinforcement learning (RL) is a collection of methods for approximating optimal solutions to stochastic sequential decision problems (Sutton & Barto, 1998). An RL system does not require a teacher to specify correct actions. Instead, it tries different actions and observes their consequences to determine which actions are best. More specifically, in the RL framework, a learning *agent* interacts with an *environment* over a series of discrete time steps $t = 0, 1, 2, 3, \ldots$. At each time t, the agent observes environment *state*, s_t, and chooses an action, a_t, which causes the environment to transition to state s_{t+1} and to emit a reward, r_{t+1}. The next state and reward depend only on the preceding state and action, but they may depend on these in a stochastic manner. The objective is to learn a (possibly stochastic) mapping from states to actions, called a *policy*, that maximizes the expected value of a measure of reward received by the agent over time. More precisely, the objective is to choose each action a_t so as to maximize the expected *return*, $\mathrm{E}\left\{\sum_{i=0}^{\infty} \gamma^i r_{t+i+1}\right\}$, where $\gamma \in [0, 1)$ is a discount-rate parameter.[2]

A common solution strategy is to approximate the *optimal value function*, V^*, which maps each state to the maximum expected return that can be obtained starting in that state and thereafter always taking the best actions. In this paper we use a *temporal difference* (TD) algorithm (Sutton, 1988) for updating an estimate, V, of V^*. After a transition from state s_t to state s_{t+1}, under action a_t with reward r_{t+1}, $V(s_t)$ is updated by:

$$V(s_t) \leftarrow V(s_t) + \alpha[r_{t+1} + \gamma V(s_{t+1}) - V(s_t)]$$

where α is a positive step-size (or learning rate) parameter. Here we are assuming that V is represented by a table with an entry for each state.

The next section describes how we cast the instruction scheduling problem as a task for RL.

[2] This is the most commonly studied framework for studying RL; many others are possible as well (Sutton & Barto, 1998).

The RL Scheduler

As with the supervised learning results presented in Moss, et al. (1997), our RL system learned a preference function between candidate instructions. That is, instead of learning the direct value of choosing instruction A or of choosing instruction B, the RL scheduler learned the value of choosing instruction A over choosing instruction B. In an earlier attempt to apply RL to instruction scheduling, Scheeff, et al. (1997) explored the use of non-preferential value functions. To do this, their system attempted to learn the value of choosing an individual instruction given a partial schedule without looking at the other candidate instructions. However, the results with non-preferential value functions were not as good as when the scheduler learned a preference function between instructions. A possible explanation for this is that the value of a given instruction is contextually dependent on the rest of the basic block. This is difficult to represent using local features but is easier to represent when the local features are used to compare candidate instructions. In other words, it is hard to predict the running time of a block from local information, but it is not as hard to predict the relative impact of two potential candidates. A number of researchers have pointed out that in RL, it is the relative values of states that are important in determining good policies (e.g., Utgoff and Clouse, 1991; Harmon et al., 1995; Werbos, 1992).

To adapt the TD algorithm to learning preferences between pairs of instructions, we regarded the system's states to be triples, each consisting of a current partial schedule and two different candidate instructions. We represented these triples by means of a set of discrete-valued features. Each feature was derived from knowledge of the DEC simulator. The features and our intuition for their importance are summarized in Table 3. Although these five features are not enough to completely disambiguate all choices between candidates, we observed in earlier studies of supervised learning in this problem that these features provide enough information to support about 98% of the optimal choices in blocks of size 10 or less.

Feature Name	Feature Description	Intuition for Use
Odd Partial (`odd`)	Is the current number of instructions scheduled odd or even?	If TRUE, we're interested in scheduling instructions that can dual-issue with the previous instruction.
Instruction Class (`ic`)	The Alpha's instructions can be divided into equivalence classes with respect to timing properties.	The instructions in each class can be executed only in certain execution pipelines, etc.
Weighted Critical Path (`wcp`)	The height of the instruction in the DAG (the length of the longest chain of instructions dependent on this one), with edges weighted by expected latency of the result produced by the instruction	Instructions on longer critical paths should be scheduled first, since they affect the lower bound of the schedule cost.
Actual Dual (`d`)	Can the instruction dual-issue with the previous scheduled instruction?	If Odd Partial is TRUE, it is important that we find an instruction, if there is one, that can issue in the same cycle with the previous scheduled instruction.
Max Delay (`e`)	The earliest cycle when the instruction can begin to execute, relative to the current cycle; this takes into account any wait for inputs for functional units to become available	We want to schedule instructions that will have their data and functional unit available earliest.

Table 3: Features for Instructions and Partial Schedule

The feature Odd Partial (`odd`) indicates whether the current instruction to schedule is lined up for issue in the first (even) or the second (odd) pipeline. This is necessary because the 21064 can only dual issue instructions if the first instruction matches the first pipeline in type and the second instruction matches with the second pipeline. The feature Actual Dual (`d`) further addresses this issue by indicating whether or not the given instruction can actually dual issue with the previously scheduled instruction. This is only relevant if `odd` is true. Both `odd` and `d` are binary features.

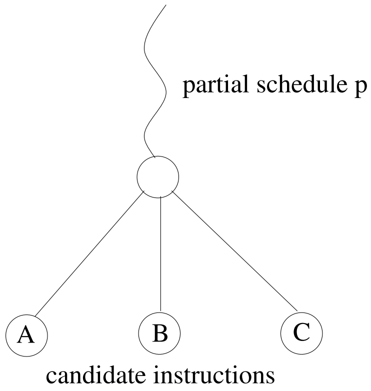
The Instruction Class (`ic`) feature divides the instructions into 20 equivalence classes, where all instructions in a class match to a certain pipeline and have the same timing properties. These equivalence classes were determined from the simulator.

The remaining two features focus on execution times. Weighted Critical Path (`wcp`) measures the longest path from a given instruction to a leaf node in the DAG. Edges in the DAG are weighted by the latency, as in the pipeline stall example given earlier. By scheduling instructions with higher `wcp` values earlier, the lower bound of the block's running time can be changed. Also, this may free up more instructions choices for use later in the block. The final feature, Max Delay (`e`), tells the learning system how many cycles the given instruction must wait before it can execute (i.e., how many possible stall cycles).

Using these features, states were represented in the following way. Given a partial schedule, p, and two candidate instructions to schedule, A and B, the value of `odd` is determined for p, and the values of the other features are determined for both instructions A and B. Moss, et al. (1997) showed in previous experiments that the actual value of `wcp` and `e` do not matter as much as the relative values between the two candidate instructions. Thus, for these features we used the signum (σ) of the difference of their values for the two candidate instructions. (Signum returns -1, 0, or 1 depending on whether the value is less than, equal to, or greater than zero.) Thus, the feature-vector representing each triple (p,A,B), where p is a partial schedule and A and B are candidate instructions, is:

$$\text{feature\_vec}(p,A,B) = [\texttt{odd}(p), \texttt{ic}(A), \texttt{ic}(B), \texttt{d}(A), \texttt{d}(B), \sigma(\texttt{wcp}(A) - \texttt{wcp}(B)), \sigma(\texttt{e}(A) - \texttt{e}(B))]$$

Since there are only 28,800 unique feature vectors of this form, we could use a lookup table to represent the value function. Further, the learning algorithm described below ensures that the vector representing (p,A,B) is the negative of the one representing (p,B,A)), which means that there are really only 14,400 unique feature vectors. Figure 4 gives an example partial schedule, a set of candidate instructions, and the resulting feature vectors. In this example, the RL scheduler has a partial schedule p and 3 candidate instructions A, B, and C. It constructs six feature vectors from this situation as outlined in the table on the right of the figure. However, it only needs to examine at most three of the vectors to make its choice because



Instruction pair	Feature vector
AB	feature_vec(p,A,B)
AC	feature_vec(p,A,C)
BC	feature_vec(p,B,C)
BA	feature_vec(p,B,A)
CA	feature_vec(p,C,A)
CB	feature_vec(p,C,B)

Figure 4: On the left is a graphical depiction of a partial schedule and three candidate instructions. The table on the right shows the feature vectors representing this information.

of the symmetry mentioned above.

The RL scheduler makes scheduling decisions using an ε-greedy action selection process (Sutton and Barto, 1998). For each partial schedule p, the scheduler finds the most preferred action through comparison of the values given by the current value function to all the instruction pairs (given p). With probability $(1-\varepsilon)$, $0 < \varepsilon < 1$, the most preferred instruction is scheduled, i.e., appended to p. With probability ε a random but legal instruction is scheduled. We used several values for ε as described below. This process is repeated until all instructions in the block have been scheduled.

We applied the TD algorithm (Equation 4) during this scheduling process as follows. Treating the feature vectors as states, we backed up values using information about which instruction was scheduled. For example, using the partial schedule and candidate instructions shown in Figure 4, after appending instruction A to partial schedule p, the RL scheduler updates the values for (p,A,B) and (p,A,C) according to Equation 4. It also updates the values of (p,B,A) and (p,C,A) with the negative of the reward used for (p,A,B) and (p,A,C) respectively.

Note that since the value function is only defined for states where choices are to be made, the final reward is assigned to the last choice point in a basic block even if further instructions are scheduled from that point (with no choices made).

Scheeff, et al. (1997) previously experimented with RL in this domain. However, the results were not as good as they had hoped. The difficulty seems to lie in finding the right reward structure for the domain (as well as learning preferences instead of pure values). A reward based on the number of cycles that it takes to execute the block does not work well because it punishes the learner on long blocks. To normalize for this effect, Scheeff, et al. (1997) rewarded the RL scheduler based on cycles-per-instruction (CPI). Although this reward function did not punish the learner for scheduling longer blocks, it also did not work particularly well. This is because CPI does not account for the fact that some blocks have more unavoidable idle time than others. We experimented with several reward functions to account for this variation across blocks. Each reward function is described in the next section along with the results of learning using that function.

Experimental Results

To test the RL scheduler, we used all 18 programs in the SPEC95 suite. To accelerate learning, we trained only on blocks of size 100 or less. However, this eliminated only a fraction of a percent of the total basic blocks in the 18 programs. To establish a baseline for our results, we also scheduled all blocks in all benchmark programs in SPEC95 using uniformly random scheduling choices. These results are summarized in Table 4. The performance metric is the same as for the rollout experiments (i.e., Equation 1). Each ratio is a geometric mean across 5 runs. Although the overall mean is 30% slower than DEC, several applications ran more than two times slower than DEC.

We experimented with two different reward functions. All reward functions gave zero reward until the RL scheduler had completely scheduled the block. The first final reward we used was:

$$r = \frac{(\text{cycles to execute block using DEC} - \text{cycles to execute block using RL})}{\text{number of instructions in block}}$$

Fortran programs				**C programs**			
App	**Ratio**	**App**	**Ratio**	**App**	**Ratio**	**App**	**Ratio**
applu	1.294	apsi	1.371	cc1	1.121	compress95	1.106
fpppp	1.343	hydro2d	1.266	go	1.176	ijpeg	1.214
mgrid	2.159	su2cor	1.387	li	1.077	m88ksim	1.111
swim	2.070	tomcatv	1.155	perl	1.148	vortex	1.103
turb3d	1.518	wave5	1.417	C geometric mean:			1.131
Fortran geometric mean:			1.468	Overall geometric mean:			1.307

Table 4: Simulated performance of the random scheduler on each application in SPEC95 as compared to DEC on all blocks of size 100 or less.

This rewards the RL scheduler positively for outperforming the DEC scheduler and negatively for performing worse than the DEC scheduler. This reward is normalized for block size.

We also wanted to test learning with a reward that did not depend on the presence of the DEC scheduler. The second final reward that we used was:

$$r = \frac{(\text{weighted critical path of DAG root} - \text{cycles to execute block using RL})}{\text{number of instructions in block}}$$

The weighted critical path (`wcp`) helps to solve the problem created by blocks of the same size being easier or harder to schedule than each other. When a block is harder to execute than another block of the same size, `wcp` tends to be higher, thus causing the learning system to receive a different reward. The feature `wcp` is correlated with the predicted number of execution cycles for the DEC scheduler with a correlation coefficient of $r = 0.9$. Again, the reward is normalized for block size.

For both reward functions described above, we trained the RL scheduler on all blocks of size 100 or less from compress95. This excluded only one block from compress95. The parameters were $\varepsilon = 0.5$ for the first 100 epochs and $\varepsilon = 0.25$ for the second 100 epochs. An epoch is one pass through the entire program scheduling all basic blocks of the given size. The learning rate, α, was 0.0001. After each epoch

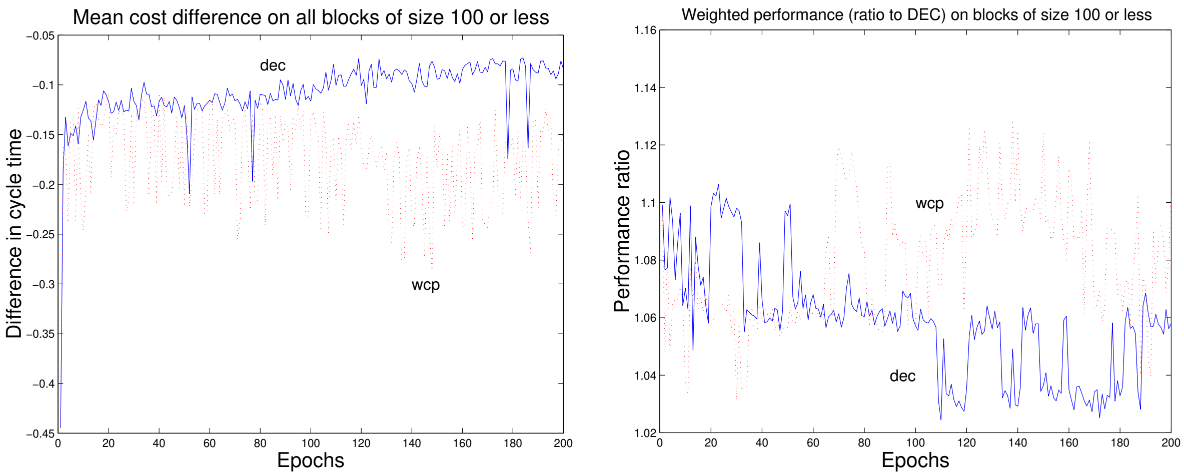


Figure 5: The figure on the left shows the difference in cost across all blocks in compress95 over the 200 training epochs for both of the reward functions we tested. The figure on the right shows the corresponding weighted performance of the system as compared to DEC for both rewards.

of training, the learned value function was evaluated greedily ($\varepsilon = 0$). Figure 5 shows the results of each reward function on the same performance ratio as before and on the mean difference in cycle time across blocks without regard to how often each block is executed.

As the figure shows, the RL scheduler performed the best using a reward function based on the DEC scheduler. To test the applicability of the learned value function to other programs we used the best value function across the first 200 epochs to greedily schedule the other 17 benchmarks. The results are shown in Table 5.

Fortran programs				**C programs**			
App	**Ratio**	**App**	**Ratio**	**App**	**Ratio**	**App**	**Ratio**
applu	1.093	apsi	1.174	cc1	1.042	compress95	1.024
fpppp	1.165	hydro2d	1.098	go	1.042	ijpeg	1.046
mgrid	1.465	su2cor	1.161	li	1.018	m88ksim	1.042
swim	1.375	tomcatv	1.057	perl	1.043	vortex	1.030
turb3d	1.231	wave5	1.197	C geometric mean:			1.036
Fortran geometric mean:			1.196	Overall geometric mean:			1.122

Table 5: Simulated performance of the greedy RL-scheduler on each application in SPEC95 as compared to DEC using the value function trained on compress95 with the DEC scheduler reward function.

Table 6 shows the same ratios for learning with the wcp reward function.

Fortran programs				**C programs**			
App	**Ratio**	**App**	**Ratio**	**App**	**Ratio**	**App**	**Ratio**
applu	1.247	apsi	1.307	cc1	1.059	compress95	1.031
fpppp	1.328	hydro2d	1.192	go	1.082	ijpeg	1.054
mgrid	1.812	su2cor	1.326	li	1.040	m88ksim	1.040
swim	1.076	tomcatv	1.089	perl	1.056	vortex	1.039
turb3d	1.470	wave5	1.348	C geometric mean:			1.050
Fortran geometric mean:			1.306	Overall geometric mean:			1.185

Table 6: Simulated performance of the greedy RL-scheduler on each application in SPEC95 using the best learned value function training on compress95 with the wcp reward function.

By training the RL scheduler on compress95 for 200 epochs, we have more than halved the difference between RANDOM and DEC for problems on which the RL scheduler had never been trained. This demonstrates good generalization across basic blocks. Although there are benchmarks that perform much more poorly than the rest (mgrid and swim), those benchmarks perform more than 100% worse than DEC under the RANDOM scheduler. Although the RL scheduler is still inferior to DEC (46% and 37% slower respectively), it has more than halved the difference between RANDOM and DEC.

Although these numbers are promising, we also experimented with further training and testing on each of the benchmarks. Starting with the value function learning from compress95, we trained on each of the other benchmark suites for 10 epochs and greedily evaluated (i.e., $\varepsilon = 0$) the updated value function at the end of each epoch. The best number from each of the 10 runs is reported in Table 7.

Discussion of Results

The results reveal several interesting effects. First, as one can tell from our splits of the programs into “Fortran” and “C” categories, there is a significant difference in performance between the two. This may be related to several factors: the code comes from different compilers; the Fortran programs use floating point

Fortran programs				C programs			
App	**Ratio**	**App**	**Ratio**	**App**	**Ratio**	**App**	**Ratio**
applu	1.115	apsi	1.163	cc1	1.046	compress95	1.024
fpppp	1.153	hydro2d	1.108	go	1.039	ijpeg	1.015
mgrid	1.367	su2cor	1.184	li	1.001	m88ksim	1.015
swim	1.405	tomcatv	1.076	perl	1.031	vortex	1.020
turb3d	1.216	wave5	1.153	C geometric mean:			1.024
Fortran geometric mean:			1.190	Overall geometric mean:			1.113

Table 7: Simulated performance of the greedy RL-scheduler on each application in SPEC95 after 10 epochs of training from the compress95 learned value function.

a lot (floating point instructions have high latencies) while the C ones do not; and we trained on a C program and even in the re-training, did not give much time for tuning to a rather different usage of instructions. This shows a clear opportunity for further work. For example, we could train using some Fortran blocks as well, or train (and evaluate) separately for Fortran and C, since the two are processed by distinct compilers and could have distinct instruction schedulers in practice.

Second, the additional training of 10 epochs seemed to help noticeably. This is interesting in that using a large number of epochs actually degraded performance, suggesting that after an initial short period of learning, the system was starting the unlearn (and perhaps later relearn).

Third, the wcp reward function, which is more realistic to have available in practice than the DEC simulator reward function, while doing better than random, did not perform as well as the simulator reward function. This is another area presenting opportunities for further work.

Finally, the "oscillations" apparent in the learning curves suggest that the problem has "hidden state". Actually, we know that the features we used do not capture everything needed to do perfect scheduling, yet the DEC scheduler uses only those features and does quite well. Still, perhaps some small number of additional features would help resolve or reduce the learning oscillations and improve performance.

There are several characteristics of the problem that perhaps are presenting themselves here. One is that each basic block is a problem instance, so unlike learning problems such as puddle worlds, we must learn general facts to carry over to other instances. It is as if we were asked to learn on several puddle worlds and then asked to generalize to other puddle worlds that are "similar" in some way that is not easy to articulate. Straightforward reward schemes do not work well, since the cost of interest (execution time) is not a good measure of the quality of a schedule—our two reward functions intend to capture the "degree of difficulty" of the problem instance, and reward based on that. Still, it seems clear that our wcp measure is not the best metric of difficulty. On the other hand, our RL scheme clearly gains significant competency at the task.

5 Conclusions

The advantages of the RL scheduler are its performance on the task, its speed, and the fact that it does not rely on any heuristics for training. Each run was much faster than with rollouts and the performance was much better than the performance of RANDOM. In a system where multiple architectures are being tested, RL could provide a good scheduler with minimal setup and training.

We have demonstrated two methods of instruction scheduling that do not rely on having heuristics and that perform quite well. Future work could address tying the two methods together while retaining the speed of the RL learner, issues of global instruction scheduling, scheduling loops, and validating the techniques on other architectures.

Acknowledgments

We thank John Cavazos and Darko Stefanović for setting up the simulator and for prior work in this domain, along with Paul Utgoff, Doina Precup, Carla Brodley, and David Scheeff. We wish to thank Andrew Fagg, Doina Precup, and Balaraman Ravindran for comments on earlier versions of the paper. We also thank the Center for Intelligent Information Retrieval for lending us a 21064 on which to test our schedules. This work is supported in part by the National Physical Science Consortium, Lockheed Martin, Advanced Technology Labs, and NSF grant IRI-9503687 to Roderic A. Grupen and Andrew G. Barto. We thank various people of Digital Equipment Corporation, for the DEC scheduler and the ATOM program instrumentation tool (Srivastava & Eustace, 1994), essential to this work. We also thank Sun Microsystems and Hewlett-Packard for their support.

References

Bertsekas, D. P. (1997). Differential training of rollout policies. In *Proc. of the 35th Allerton Conference on Communication, Control, and Computing*. Allerton Park, Ill.

Bertsekas, D. P., Tsitsiklis, J. N. & Wu, C. (1997). Rollout algorithms for combinatorial optimization. *Journal of Heuristics*.

DEC (1992). *DEC chip 21064-AA Microprocessor Hardware Reference Manual* (first edition Ed.). Maynard, MA: Digital Equipment Corporation.

Harmon, M. E., Baird III, L. C. & Klopf, A. H. (1995). Advantage updating applied to a differential game. In G. Tesauro, D. Touretzky, T. L. (Ed.), *Advances in Neural Information Processing Systems: Proceedings of the 1994 Conference* (pp. 353–360). San Mateo, CA: Morgan Kaufmann.

Moss, J. E. B., Utgoff, P. E., Cavazos, J., Precup, D., Stefanović, D., Brodley, C. E. & Scheeff, D. T. (1997). Learning to schedule straight-line code. In *Proceedings of Advances in Neural Information Processing Systems 10 (Proceedings of NIPS'97)*. MIT Press.

Proebsting, T. Least-cost instruction selection in DAGs is NP-Complete. http://www.research.microsoft.com/ toddpro/papers/proof.htm.

Rumelhart, D. E., Hinton, G. E. & Williams, R. J. (1986). Learning internal representations by error propagation. In D. E. Rumelhart & J. L. McClelland (Eds.), *Parallel Distributed Processing: Explorations in the Microstructure of Cognition, vol.1: Foundations*. Cambridge, MA: Bradford Books/MIT Press.

Scheeff, D., Brodley, C., Moss, E., Cavazos, J. & Stefanović, D. (1997). Applying reinforcement learning to instruction scheduling within basic blocks. Technical report, University of Massachusetts, Amherst.

Sites, R. (1992). *Alpha Architecture Reference Manual*. Maynard, MA: Digital Equipment Corporation.

Srivastava, A. & Eustace, A. (1994). ATOM: A system for building customized program analysis tools. In *Proceedings ACM SIGPLAN '94 Conference on Programming Language Design and Implementation* (pp. 196–205).

Stefanović, D. (1997). The character of the instruction scheduling problem. University of Massachusetts, Amherst.

Sutton, R. S. (1988). Learning to predict by the method of temporal differences. *Machine Learning*, *3*, 9–44.

Sutton, R. S. & Barto, A. G. (1998). *Reinforcement Learning. An Introduction*. Cambridge, MA: MIT Press.

Tesauro, G. & Galperin, G. R. (1996). On-line policy improvement using monte-carlo search. In *Advances in Neural Information Processing: Proceedings of the Ninth Conference*. MIT Press.

Utgoff, P. E., Berkman, N. & Clouse, J. (1997). Decision tree induction based on efficient tree restructuring. *Machine Learning*, *29(1)*, 5–44.

Utgoff, P. E. & Clouse, J. A. (1991). Two kinds of training information for evaluation function learning. In *Proceedings of the Ninth Annual Conference on Artificial Intelligence* (pp. 596–600). San Mateo, CA: Morgan Kaufmann.

Utgoff, P. E. & Precup, D. (1997). Constructive function approximation. Technical Report 97-04, University of Massachusetts, Amherst.

Werbos, P. (1992). Approximate dynamic programming for real-time control and neural modeling. In D. A. White & D. A. Sofge (Eds.), *Handbook of Intelligent Control: Neural, Fuzzy, and Adaptive Approaches* (pp. 493–525). New York: Van Nostrand Reinhold.