Basic-block Instruction Scheduling 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

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, both the rollout scheduler and the reinforcement learning scheduler outperformed a commercial scheduler on several applications.

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. 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 a 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 instruction scheduling 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.

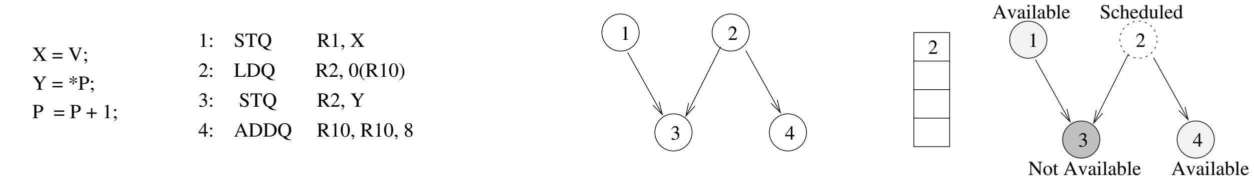
A second possible use of machine learning techniques in instruction scheduling is by the end user. Instead of scheduling code using a static scheduler trained on benchmarks when the compiler was written, a user would employ a learning scheduler to discover important characteristics of that user's code.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

Instruction scheduling is a large-scale optimization problem in several ways. First, there can be millions of instructions and tens of thousands of basic blocks within a given program. Scheduling each block optimally using exhaustive search is much too difficult and time-consuming to work in practice. Second, the problem of generalizing the scheduler from the training programs to all possible user programs is also an optimization problem.

With these motivations in mind, we formulated and tested two autonomous methods of building an instruction scheduler. The first method used rollouts (Bertsekas, 1997; Bertsekas *et al.*, 1997; Tesauro and Galperin, 1996) and the second focused on reinforcement learning (RL) (Sutton and Barto, 1998). Both methods were implemented for the Digital Alpha 21064. The next section gives a brief overview of the domain. For a complete description, see McGovern *et al.* (1999).

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. Our schedulers can reorder the machine instructions within a basic block but cannot rewrite, add,



(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

or remove any instructions. The goal of the scheduler is to find the fastest valid ordering of the instructions in a block. A valid ordering is one that preserves the in-order semantics of the original code. We insure validity by creating a dependency graph that directly represents the necessary ordering relationships in a directed acyclic graph (DAG). An example basic block written in C is shown in Figure 1. The figure also shows the assembled program, its resulting DAG, and an example partial schedule.

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 next instruction to schedule from a list of available candidates, each scheduler use a different evaluation function to decide which available instruction is best to schedule next.

Moss *et al.* (1997) showed that several supervised learning techniques could induce excellent ($96-97\%$ optimal choices within blocks of size 10 or less) basic block instruction schedulers for this task. Although each of these 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 is too time-consuming to perform on blocks with more than 10 instructions, because optimal instruction scheduling is NP-hard (Stefanović, 1997, 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 knowledge of 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 (69.53%). 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.

3 Rollouts

Rollouts are a form of Monte Carlo search, first introduced by Tesauro and Galperin (1996) for use in backgammon. Bertsekas (1997) and Bertsekas *et al.* (1997) 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

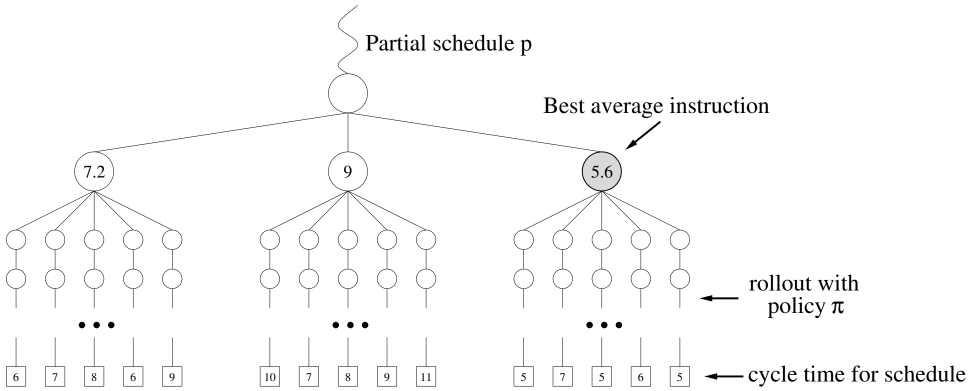


Figure 2: Diagram of the actions of a rollout scheduler when rolling out three different instructions five times each.

time. This process is illustrated graphically in Figure 2.

Our first set of rollout experiments compared three rollout policies π. While Bertsekas (1997) and Bertsekas *et al.* (1997) 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 $O(n^2 m)$ 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 1 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 during a run of the entire program. More concisely, the performance measure was:

$$\text{ratio} = \frac{\sum_{\text{all blocks}} (\text{rollout time} \times \#\text{ times executed})}{\sum_{\text{all blocks}} (\text{DEC time} \times \#\text{ times executed})}$$

where time is the number of cycles that block took to execute. This means that a faster running time than DEC on the part of our scheduler would give a ratio of less than one.

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

Table 1: Ratios of the weighted execution time of the rollout scheduler to the DEC scheduler on the SPEC95 program applu. A ratio of less than one means that the rollouts outperformed the DEC scheduler. The entries in italics are those that outperformed DEC.

Although Stefanović (1997) noted that rescheduling a block has no effect on 68% of the blocks of size 10 or

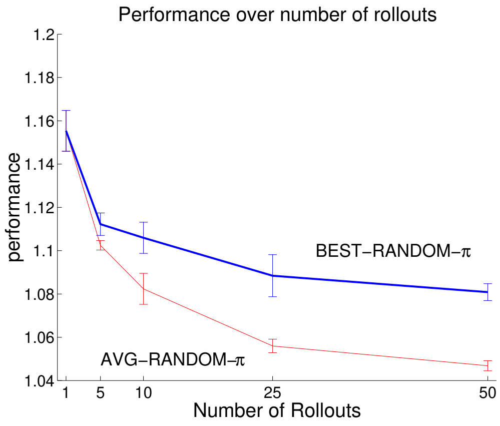


Figure 3: Performance of rollout scheduler with the random model as a function of the number of rollouts and the choice of evaluation function. The bars above and below each data point are one standard deviation from the mean.

less, the random scheduler performed very poorly. On applu, RANDOM was 31% slower than DEC. Overall, RANDOM is 30.1% 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 can 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 (1997) and Bertsekas *et al.* (1997) also predicts this answer.

Although the performance of the rollout scheduler can be excellent, rollouts are costly in time. As mentioned before, using 25 rollouts per block required 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 with a faster running time. RL schedulers run in the $O(n)$ time of a greedy list scheduler.

4 Reinforcement Learning

We use the standard formulation of reinforcement learning (Sutton and Barto, 1998) which is summarized below. For the sake of brevity, we omit a complete description of reinforcement learning and of how we cast the instruction scheduling problem as a reinforcement learning task. See McGovern *et al.* (1999) for more detail.

In the RL framework, a learning agent interacts with an environment over a series of discrete time steps. At each time step, t, the agent observes the *state*, s_t, of the environment and chooses an action which causes the environment to transition to a new state, s_{t+1} and to emit a reward r_{t+1}. The next state and reward depend only on the previous state and action, possibly in a stochastic manner. The objective of the learning agent is to learn a (possibly stochastic) mapping from states to actions that maximizes the expected value 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.

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.

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. (Utgoff and Clouse, 1991; Harmon *et al.*, 1995; Werbos, 1992)

To adapt the TD algorithm to learning preferences between pairs of instructions, we defined a feature vector over the current partial schedule and each pair of candidate instructions. The value function was approximated by a linear weighting over the feature vector. Each feature was derived from knowledge of the DEC simulator. The features and our intuition concerning their importance are summarized in Table 2. 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.

The feature vector $\vec{f}$ for each triple (p,A,B), where p is a partial schedule and A and B are candidate instructions, is:

$$\begin{aligned}\vec{f}(p,A,B) \;=\; & [\mathtt{odd}(p),\mathtt{ic}(A),\mathtt{ic}(B),\\ & \mathtt{d}(A),\mathtt{d}(B),\\ & \sigma(\mathtt{wcp}(A)-\mathtt{wcp}(B)),\\ & \sigma(\mathtt{e}(A)-\mathtt{e}(B))]\end{aligned}$$

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.) Because the features `odd`, `ic`, and `d` are categorical, they were represented as bit vectors.

Previous experiments with a table lookup representation did not perform or generalize as well as by using the linear function approximator (McGovern *et al.*, 1999). This suggests that the value information is mostly contained in a low order feature representation which a linear function approximator is able to capture but a table lookup representation is not.

During learning, the RL scheduler makes scheduling decisions using an ε-greedy action selection process (Sutton and Barto, 1998). This means that scheduler chooses the most preferred action $(1-\varepsilon)$% of the time and a random but legal action ε% of the time.

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 2: Features for Instructions and Partial Schedule

As mentioned before, Scheeff *et al.* (1997) previously experimented with RL in this domain and their results were not as successful as they had hoped. One 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 two 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. This eliminated only a fraction of a percent of the total basic blocks in the 18 programs while speeding training considerably. To establish a baseline for our results, we also scheduled all benchmark programs in SPEC95 using uniformly random scheduling choices. These results are summarized in Table 3. 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, some applications ran more than two times slower than DEC.

RANDOM scheduling

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

Table 3: Simulated performance of the RANDOM scheduler on each application in SPEC95 as compared to DEC on all blocks of size 100 or less.

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:

$$r_{DEC} = \frac{(\text{DEC time} - \text{RL time})}{\text{number of instructions in block}}$$

where time is the number of cycles the block took to execute. This rewards the RL scheduler positively for outperforming the DEC scheduler and negatively for performing worse than the DEC scheduler. This reward is

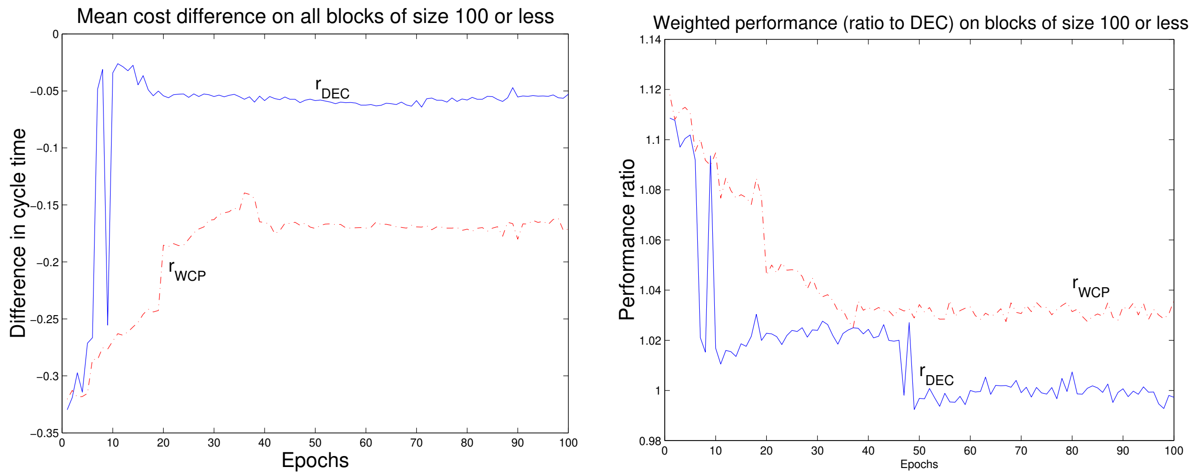


Figure 4: The figure on the left shows the difference in cycle time between RL and DEC across all blocks of compress95 for each of the the 100 training epochs for both of the reward functions we tested. The figure on the right shows the corresponding weighted performance of the system as compared to DEC for both rewards.

normalized for block size.

It is not always realistic to assume that we already have a good scheduler on hand for use in rewarding the RL scheduler. To test a reward function that did not depend on the presence of the DEC scheduler, we used the following final reward function:

$$\begin{aligned} \text{max\_time} &= \max(\text{wcp of DAG root}, \frac{\#\text{instructions}}{2}) \\ r_{WCP} &= \frac{(\text{max\_time} - \text{RL time})}{\text{number of instructions in block}} \end{aligned}$$

The weighted critical path (`wcp`) of the root node in the DAG 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 wcp of the DAG root is correlated with the predicted number of execution cycles for the DEC scheduler with a correlation coefficient of $r = 0.9$. The number of instructions divided by two gives another lower bound on the running time of the block. If all instructions in the block take only one cycle to execute and the block is fully pipelined, the number of instructions divided by two is the fastest possible execution time for the block. The max_time feature is correlated with DEC time with $r = 0.91$. 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 for 100 epochs. This excluded only one block from compress95. An epoch is one pass through the entire program scheduling all basic blocks of the given size. The parameters were $\varepsilon = 0.05$, $\gamma = 0.9$, and $\alpha = 0.001$. After each epoch of training, the learned value function was evaluated greedily ($\varepsilon = 0$). Figure 4 shows the results of greedily evaluating the learned value function for 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 (DEC time - RL time).

RL trained on compress95 with r_{DEC}

App	Ratio	App	Ratio
Fortran programs			
applu	1.106	apsi	1.117
fpppp	1.106	hydro2d	1.067
mgrid	1.453	su2cor	1.163
swim	1.559	tomcatv	1.042
turb3d	1.213	wave5	1.145
Fortran geometric mean:			1.187
C programs			
cc1	1.035	*compress95*	*0.998*
go	1.048	ijpeg	1.016
li	1.012	m88ksim	1.010
perl	1.033	vortex	1.029
C geometric mean:			1.023
Overall geometric mean:			1.111

Table 4: 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.

As the figure shows, the RL scheduler performed the best using the reward function based on the DEC scheduler (r_{DEC}). To test the applicability of the learned value function to other programs we used the value function from the end of the 100th epoch to greedily schedule the other 17 benchmarks. The results are shown in Table 4.

After training with r_{DEC} for 100 epochs on compress95, the RL scheduler was able to beat the commercial DEC scheduler on compress95. We also brought the performance on unseen C programs to within 2% of the performance of DEC and to within 18% for unseen Fortran programs. 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. Despite the fact that the performance of the RL scheduler is still inferior to DEC on these programs (45% and 55% slower respectively), it has more than halved the difference between RANDOM and DEC.

Table 5 shows the same ratios for the RL scheduler which was trained using r_{WCP}.

RL trained on compress95 with r_{WCP}

Fortran programs			
App	**Ratio**	**App**	**Ratio**
applu	1.163	apsi	1.161
fpppp	1.200	hydro2d	1.098
mgrid	1.863	su2cor	1.157
swim	1.510	tomcatv	1.058
turb3d	1.230	wave5	1.199
Fortran geometric mean:			1.246
C programs			
cc1	1.038	compress95	1.025
go	1.067	ijpeg	1.072
li	1.045	m88ksim	1.045
perl	1.040	vortex	1.026
C geometric mean:			1.045
Overall geometric mean:			1.152

Table 5: 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.

Although the RL scheduler was able to learn a competent scheduling policy using r_{WCP}, it was not as successful as when it trained using r_{DEC}. The overall mean performance slowed from 11% slower than DEC to 15% slower than DEC. Although this is significantly better than RANDOM, it is not as good as we had hoped. This is possibly because the current simulator places some limitations on the value of wcp which cause it to not be a perfect predictor of DEC's running time. However, these results point to the idea that a similar reward function using wcp can be structured to perform at least as well as learning with r_{DEC}. Future work will address this issue.

The above results reveal an interesting effect about the type of program being scheduled. 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, including: the code comes from different compilers; and the Fortran programs use mainly floating point operations (which have high latencies) while the C programs do not. In practice, C programs and Fortran programs are processed by distinct compilers and often have distinct instruction schedulers. To account for this, we also trained the RL scheduler using r_{DEC} for 100 epochs on the Fortran program applu. All parameters remained the same except for the learning rate which was reduced to $\alpha = 0.0005$.

By training on applu, the RL scheduler improved its performance on applu to only 5% slower than DEC. The scheduler was previously only able to achieve a performance of 10% slower than DEC from training on the C program compress95. This is a significant improvement. To test how well this improvement generalized to the other Fortran programs, we took the best learned value function from the 100 training epochs on applu and greedily evaluated it on the other 17 benchmarks. These results are given in Table 6.

RL trained on applu with r_{DEC}

Fortran programs			
App	**Ratio**	**App**	**Ratio**
applu	1.053	apsi	1.099
fpppp	1.089	hydro2d	1.049
mgrid	1.327	su2cor	1.126
swim	1.271	tomcatv	1.034
turb3d	1.166	wave5	1.119
Fortran geometric mean:			1.130
C programs			
cc1	1.031	compress95	1.007
go	1.044	ijpeg	1.015
li	1.011	m88ksim	1.010
perl	1.043	vortex	1.035
C geometric mean:			1.025
Overall geometric mean:			1.082

Table 6: Simulated performance of the greedy RL-scheduler on each application in SPEC95 using the best learned value function from training on applu over 100 epochs with the DEC reward function.

As the table shows, training on the Fortran program applu significantly improved the performance of the Fortran programs (Fortran performance improved to 1.13 from 1.18) while barely hurting the performance on the C programs. Furthermore, the performance on the particularly difficult programs, mgrid and swim, has improved significantly (1.45 to 1.32 and 1.55 to 1.27, respectively). This experiment points to a need for more exploration of separate schedulers for each programming language.

Although the results of training on one program and testing on the other 17 benchmark programs were quite promising, we wanted to see how much further training on each program could improve the results. This could be similar to a user profiling code and then rescheduling the code based on the results of the profiling. Instead of training from a new uninitialized value function for each benchmark suite, we initialized the value function from the 100th epoch of training on compress95. We trained each application for 10 epochs using the parameters, $\alpha = 0.001$, $\gamma = 0.9$, and $\epsilon = 0.05$. After each training epoch, we evaluated the new value function greedily. The best number from each of the 10 epochs is reported in Table 7.

RL cross training

App	Ratio	App	Ratio
Fortran programs			
App	**Ratio**	**App**	**Ratio**
applu	1.068	apsi	1.094
fpppp	1.100	hydro2d	1.063
mgrid	1.368	su2cor	1.130
swim	1.354	tomcatv	1.033
turb3d	1.159	wave5	1.109
Fortran geometric mean:			1.143
C programs			
cc1	1.021	*compress95*	*0.998*
go	1.035	*ijpeg*	*0.999*
li	1.010	m88ksim	1.006
perl	1.021	vortex	1.028
C geometric mean:			1.015
Overall geometric mean:			1.084

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.

It is clear that the additional training helps the performance of the RL scheduler noticeably. The scheduler was able to outperform DEC on the C program ijpeg as well as on compress95. The overall C mean improved from 1.02 to 1.01 while the Fortran mean improved to 1.14 from 1.18. Further training may improve the results even more. This is a useful result since an end-user of an RL scheduler could use this to quickly profile and reschedule important programs.

There are several characteristics of the scheduling problem that perhaps are presenting themselves here. One is that each basic block is a problem instance, so unlike learning problems such as gridworlds, we must learn general facts to carry over to other instances. It is as if we were asked to learn on several gridworlds and then asked to generalize to other gridworlds 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 need to rely on any heuristics for training. Each run was much faster by an order of magnitude than with rollouts and the performance did not suffer considerably. RL was able to outperform DEC on two applications in the SPEC95 benchmark suite and was able to perform competitively overall. In a system where multiple architectures are being tested, RL could provide a good scheduler with minimal setup and training.

We have previously experimented with a combined RL and rollout scheduler but did not pursue the ideas because of the speed of rollouts. This area is worth pursuing with the current RL results. A combined scheduler might be able to outperform DEC on all SPEC 95 suites.

This paper has 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, Balaraman Ravindran, and Daniel Bernstein 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 and Eustace, 1994), essential to this work. We also thank Sun Microsystems and Hewlett-Packard for their support.

References

(1) Dmitri P. Bertsekas, John N. Tsitsiklis, and C. Wu. Rollout algorithms for combinatorial optimization. *Journal of Heuristics*, April 1997.

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

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

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

(5) Amy McGovern, Eliot Moss, and Andrew G. Barto. Scheduling straight-line code using reinforcement learning and rollouts. Technical Report 99-23, University of Massachusetts, Amherst, April 1999.

(6) J. Eliot B. Moss, Paul E. Utgoff, John Cavazos, Doina Precup, Darko Stefanović, Carla E. Brodley, and David T. Scheeff. Learning to schedule straight-line code. In *Proceedings of Advances in Neural Information Processing Systems 10 (Proceedings of NIPS'97)*. MIT Press, 1997.

(7) Todd Proebsting. Least-cost instruction selection in DAGs is NP-Complete. http://www.research.microsoft.com/ toddpro/papers/proof.htm.

(8) David Scheeff, Carla Brodley, Eliot Moss, John Cavazos, and Darko Stefanović. Applying reinforcement learning to instruction scheduling within basic blocks. Technical report, University of Massachusetts, Amherst, 1997.

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

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

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

(12) Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning. An Introduction*. MIT Press, Cambridge, MA, 1998.

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

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

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

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