Building a Basic Block Instruction Scheduler with 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
(413) 545-1596 (Tel)
(413) 545-1249 (Fax)

September 21, 1999

Abstract

The execution order of a block of computer instructions on a pipelined machine can make a difference in running time by a factor of two or more. Compilers use heuristic schedulers appropriate to each specific architecture implementation to achieve the best possible program speed. 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 on all benchmarks tested, and the reinforcement learning scheduler outperformed the commercial scheduler on several benchmarks and performed well on the others. The combined reinforcement learning and rollout approach was also very successful. We present results of running the schedules on Compaq Alpha machines and show that the results from the simulator correspond well to the actual run-time results.

Keywords: Reinforcement learning, Instruction scheduling, Rollouts
Running Head: Instruction scheduling with RL

*First submission March, 1999. Accepted for publication and final submission September, 1999

1 Introduction

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. 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 scheduling instructions is NP-Complete (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 particular user.

When building a scheduler, there is a tradeoff between the running time of the compiler and the running time of the executable generated. Potentially, the longer the compiler runs, the more possibilities it can consider and the better the schedule it can generate. However, instead of exploring each individual program online, a compiler could make use of an algorithm tuned off-line. This would decrease the running time of the compiler while potentially sacrificing some quality of the final schedule.

With these motivations in mind, we formulated and tested two methods of building an instruction scheduler. The first method used rollouts (Woolsey, 1991; Abramson, 1990; Galperin, 1994; Tesauro and Galperin, 1996; Bertsekas *et al.*, 1997a,b) and the second focused on reinforcement learning (RL) (Sutton and Barto, 1998). We also investigated the effect of combining the two methods. All methods were implemented for the Compaq Alpha 21064. These methods address the time tradeoff directly. Rollouts evaluate schedules online during compilation while reinforcement learning trains on more general programs and runs more quickly at compile time. The next section gives a domain overview and discusses results using supervised learning on the same task. We then present the rollout scheduler, the reinforcement learning scheduler, and the results of combining the two methods.

2 Overview of Instruction Scheduling

We focused on scheduling *basic blocks* of instructions on the 21064 version (DEC, 1992) of the Compaq Alpha processor (Sites, 1992). A basic block is a set of machine instructions with a single entry point and a single exit point. It 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 execution time of the block. A valid ordering is one that preserves the semantically necessary ordering constraints of the original code. This means that potentially conflicting reads and writes of either registers or memory are not reordered. We ensure validity by creating a dependence 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.

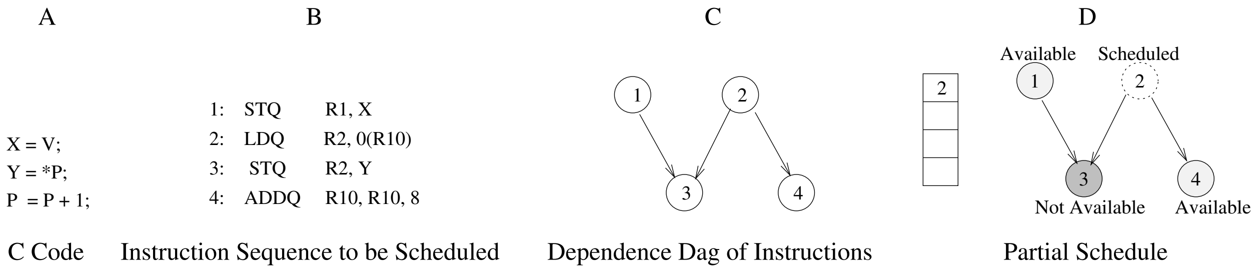


Figure 1: Example basic block code, DAG, and partial schedule

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 dependences 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. If the compiler does not know that X and Y are non-overlapping variables, it must assume that 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.

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. There are five valid schedules for the four instructions shown in Figure 1. Assume that two of these schedules were to execute on a single pipeline machine where loads take two cycles to execute and all other instructions take one cycle. The two example schedules are shown in Table 1. 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, on 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 machine 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 instruction after that matches the second pipeline. A given instruction can take anywhere from one to many tens of cycles to execute. Researchers at Compaq have made a 21064 simulator publicly available that also includes a heuristic scheduler for basic blocks. Throughout the paper, we refer to this scheduler as *DEC*[1]. 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. To counteract any differences between schedulers that might appear because of this assumption, we also ran the schedules generated in the simulator on a cluster of identical Compaq Alpha 21064 machines to obtain actual run-time results.

As the results show, using the simulator's assumptions often produces good results, but sometimes leads us to produce mediocre schedules. The whole point of our technique is to do well when scheduling programs we have not seen before and for which we have no execution time measurements, e.g., of cache hit/miss statistics. By using techniques that are more complex (both in the sense of running time complexity and in the sense of complexity of code) one could better approximate the status of all resources at the start of each block. However, we wished to see how well we could do considering each block independently, so such techniques are beyond the scope of this paper. Comparing the cost and effectiveness of block-local techniques with more global approaches to scheduling is an interesting topic for future work.

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 candidate. This continues until all viable 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

[1] Although Digital has been purchased by Compaq, we continue to refer to their scheduler as DEC to maintain consistency with our earlier papers.

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 representing the currently scheduled instructions and the candidate instructions. Each method learned a preference relation which defined when an instruction i is preferred over an 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) where they show that each method nearly always made optimal choices (i.e. $96 - 97\%$ of the time) of instructions to schedule for blocks in which computing the optimal schedule was feasible.

Although all of the supervised learning methods performed quite well, they shared several limitations. Supervised learning requires a training set consisting of correct, or approximately correct, input/output pairs. Generating useful training information requires an optimal scheduler that executes every valid permutation of the instructions within a basic block and saves the optimal permutation (the schedule with the smallest running time). As we expected from knowing that optimal scheduling is NP-Complete (Proebsting), this search was too time-consuming to perform on blocks with more than 10 instructions (Stefanović, 1997). This inhibited the supervised methods from learning using larger blocks although the methods were tested on larger blocks. Using a 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.

To test each scheduling algorithm, we used 18 SPEC95 benchmark programs (Reilly, 1995). 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 Compaq compiler at the highest level of optimization. We call the schedules output by the compiler *COM*[2]. This collection has 447,127 basic blocks, containing 2,205,466 instructions. Although 92.34% of the blocks in SPEC95 have 10 instructions or fewer, 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 and actual runs on Compaq Alpha 21064 machines. To decrease the running time of the simulator, we broke the 206 blocks whose size was greater than 100 instructions into blocks of size 100 (or less). After scheduling, these blocks were then concatenated together for actual execution. This constraint did not affect the results significantly but sped up the simulator considerably.

3 Rollouts

Rollouts are a form of Monte Carlo search, first introduced in the backgammon literature (Woolsey, 1991; Galperin, 1994; Tesauro and Galperin, 1996). In other domains, Abramson (1990) studied what we call RANDOM-π (below) in a game playing context, and Bertsekas *et al.* (1997a,b) proved important theoretical results for rollouts. 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. The scheduler appends each candidate 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

[2] In previous papers (McGovern and Moss, 1998; McGovern, Moss, and Barto, 1999) we referred to this as ORIG. This change is to reduce confusion.

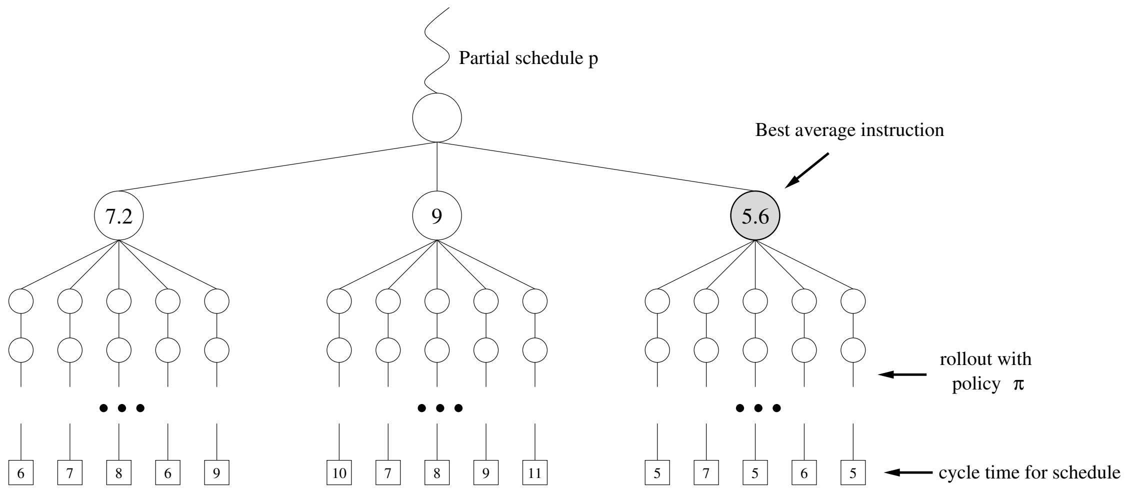


Figure 2: The actions of a rollout scheduler when rolling out three different instructions five times each. This process is repeated to schedule each instruction.

running time and returns. When π is stochastic, this rollout can be repeated many times for each instruction to achieve a estimate of the expected outcome. After rolling out each candidate (possibly repeatedly), the scheduler picks the one with the best estimated expected running time. This process is illustrated graphically in Figure 2. In this example, π is a stochastic policy. Each of the three instructions is rolled out five times. The third instruction has the lowest sample average running time of 5.6 cycles and is chosen as the next instruction to schedule. The process is then repeated at each successive decision point.

Our first set of rollout experiments compared three rollout policies. While 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 such a good heuristic policy available to use as π. Therefore, we also considered policies more likely to be available. The obvious choice is the random policy. We denote the use of the random policy for π in the rollout scheduler as *RANDOM*-π. Under this policy, a rollout makes all choices in a valid but uniformly random fashion. We also tested the use of two heuristic policies. The first was the ordering produced by the optimizing compiler COM, denoted *COM*-π. The second heuristic policy tested was the DEC scheduler itself, denoted *DEC*-π. Although the full heuristics of DEC or COM will not be available to an architect while designing a machine, a simpler set of heuristics (but more complicated than RANDOM) may be available.

The rollout scheduler performed only one rollout per candidate instruction when using COM-π and DEC-π because each is deterministic. Initially, we used 20 rollouts for RANDOM-π. Discussion of how the number of rollouts affects performance is presented later in this section. After performing a number of rollouts for each candidate instruction, the scheduler chose the instruction with the best running time averaged over the rollouts. As a baseline scheduler, we also scheduled each block with a valid but uniformly random ordering, denoted RANDOM.

Table 2 summarizes the performance of the rollout scheduler under each policy π as compared to the DEC scheduler on all 18 benchmark programs. 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 weight of each block is the number of times that block is executed. More concisely, we used the following performance measure:

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

If a scheduler produced a faster running time than DEC, the ratio would be less than one. All execution times in Table 2 are those of the simulator.

Despite the fact that Stefanović (1997) showed that rescheduling a block has no effect for 68% of the blocks of size 10 or less, the random scheduler performed very poorly. Overall, RANDOM was 27.8%

Simulator results					
Fortran programs					
App	**RANDOM**	**RANDOM-π**	**COM**	**COM-π**	**DEC-π**
applu	1.388	1.059	*0.991*	*0.982*	*0.978*
apsi	1.304	1.046	1.024	*0.999*	*0.990*
fpppp	1.246	1.049	1.043	1.004	*0.998*
hydro2d	1.175	1.028	1.018	1.006	*0.998*
mgrid	2.122	1.301	1.009	1.001	*0.967*
su2cor	1.187	1.017	1.054	1.018	*0.996*
swim	2.082	1.259	1.030	1.029	*0.975*
tomcatv	1.224	1.050	1.029	1.006	*0.998*
turb3d	1.375	1.063	1.177	1.038	*0.980*
wave5	1.400	1.092	1.032	1.001	*0.990*
Fortran geometric mean:	1.417	1.093	1.040	1.008	*0.987*
C programs					
App	**RANDOM**	**RANDOM-π**	**COM**	**COM-π**	**DEC-π**
cc1	1.117	1.007	1.022	1.001	*0.997*
compress95	1.099	*0.983*	*0.977*	*0.970*	*0.970*
go	1.173	1.011	1.028	*0.998*	*0.994*
ijpeg	1.167	1.010	1.014	*0.989*	*0.981*
li	1.083	1.006	1.012	1.001	*0.995*
m88ksim	1.113	1.001	1.042	*0.999*	*0.997*
perl	1.131	1.002	1.016	1.000	*0.996*
vortex	1.106	1.002	1.029	1.000	*0.998*
C geometric mean:	1.123	1.003	1.017	*0.995*	*0.991*
Overall geometric mean:	1.278	1.052	1.030	1.002	*0.989*

Table 2: Ratios of the simulated weighted execution time of RANDOM, COM, and rollout schedulers compared to the DEC scheduler. A ratio of less than one means that the scheduler outperformed the DEC scheduler. These entries are shown in italics.

slower than DEC. This number is a geometric mean of the performance ratios for 30 runs of each of the 18 SPEC95 benchmark suites. Without adding any heuristics and just using rollouts with the random policy, RANDOM-π came within 5% of the running time of DEC. This is also an average over 30 runs of each benchmark. Because COM, COM-π, and DEC-π are deterministic, their numbers are from one run. COM was only 3% slower than DEC and outperformed DEC on two applications. COM-π was able to outperform DEC on 6 applications. Over all the 18 benchmark applications, COM-π was essentially equal to DEC. The DEC-π scheduler was able to outperform DEC on all applications. DEC-π produced schedules that ran about

1.1% faster than the schedules produced by 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). We estimated that DEC is within a few percent of optimal on larger blocks but it is infeasible for us to find out exactly.

To test how well the simulator results corresponded with running the schedules on an actual machine, we also ran each scheduled binary on a cluster of identical Compaq Alpha 21064 machines. Because DEC, DEC-π, COM, and COM-π were deterministic, each had only one binary to run per benchmark. For these schedules, we ran each binary 15 times in a row on two separate but identical Alphas in single user mode. Each run was timed using `gnu_time` version 1.7. Due to their stochastic nature, RANDOM and RANDOM-π had 30 binaries per benchmark. Each binary was run once and timed as before. The numbers in Table 3 are the ratios of the average running time of the given schedule to DEC's average running time.

Table 3 demonstrates that binaries built using RANDOM still execute more slowly than those produced using DEC, although the differences in actual running times have decreased from 27.8% slower to only 7.4% slower. Because DEC is a heuristic scheduler tuned for the simulator, assumptions that it makes for the simulator can be detrimental on the actual machine. This is apparent when the performance of DEC is compared to RANDOM on the applications *go* and *vortex* where RANDOM was able to outperform DEC. It is hard to say exactly how or why RANDOM schedules led to better actual performance than did the DEC schedules on these benchmarks. A plausible explanation is that in a few blocks that are executed quite frequently, the DEC scheduler's assumptions were always violated and it ended up producing particularly bad schedules. For example, if certain loads *always* missed in the cache, then the DEC scheduler's assumption that they hit might well lead it to produce schedules that always stall for a long time, whereas RANDOM could well produce schedules that do not stall as much. Because of these model differences, DEC-π can perform worse than DEC on the actual machine.

Run time results					
Fortran programs					
App	**RANDOM**	**RANDOM-π**	**COM**	**COM-π**	**DEC-π**
applu	1.052	1.009	*0.991*	*0.994*	*0.995*
apsi	1.102	1.019	1.012	1.011	1.003
fpppp	1.108	1.160	*0.998*	*0.984*	*0.999*
hydro2d	1.033	1.009	*0.981*	*0.998*	*0.998*
mgrid	1.348	1.113	*0.988*	*0.989*	*0.975*
su2cor	1.142	1.087	*0.999*	1.022	1.040
swim	1.039	*0.994*	1.013	1.012	*0.985*
tomcatv	1.084	*0.996*	1.108	1.021	1.024
turb3d	1.204	1.138	*0.981*	*0.975*	*0.976*
wave5	1.045	*0.997*	*0.966*	*0.999*	1.018
Fortran geometric mean:	1.112	1.050	1.003	1.000	1.001
C programs					
App	**RANDOM**	**RANDOM-π**	**COM**	**COM-π**	**DEC-π**
cc1	1.019	1.002	1.002	1.009	1.001
compress95	1.006	1.004	1.013	1.012	1.011
go	*0.944*	*0.908*	1.010	1.019	1.048
ijpeg	1.130	1.006	*0.980*	*0.987*	*0.977*
li	1.040	1.078	*0.997*	*0.988*	*0.998*
m88ksim	1.070	1.002	*0.978*	*0.983*	*0.982*
perl	1.040	*0.998*	*0.986*	*0.989*	1.003
vortex	*0.985*	*0.973*	*0.961*	*0.971*	1.049
C geometric mean:	1.028	*0.996*	*0.991*	*0.994*	1.008
Overall geometric mean:	1.074	1.026	*0.998*	*0.998*	1.004

Table 3: Ratios of the actual run times for the RANDOM scheduler, the COM scheduler, and the rollout schedulers. Each is compared to DEC. The ratios for the executables which ran faster than DEC are shown in italics.

In simulation, the performance of the schedules produced by COM was slightly slower than those produced by DEC. On the actual runs, COM ends up slightly faster than DEC. This result is consistent with what was observed by Moss *et al.* (1997). The COM scheduler almost certainly pays more careful attention to inner loops, and particularly to how the resources left busy by previous iterations affect future iterations. We believe that this alone can explain why it performs better on the actual hardware and not as well on the DEC simulator model. The run-time rollout results are consistent with the results observed in the simulator.

Part of the motivation for using rollouts in a scheduler was to obtain efficient schedules without spending the time to build a such precise heuristic scheduler as COM 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 performed for each candidate instruction. We tested RANDOM-π using 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 BEST-RANDOM-π might lead to better performance overall because it meant scheduling the first instruction on the most promising scheduling path. Due to the long running time of the rollout scheduler, we focused on one program in the SPEC 95 suite: *applu*, which is written in Fortran and focuses on floating point operations.

Figure 3 plots the performances of the rollout schedulers AVG-RANDOM-π and BEST-RANDOM-π 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 30 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 than DEC, which is a considerable improvement over RANDOM's performance of 38% 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 at around 4% slower then DEC. As Figure 3 shows, the improvement-per-number-of-rollouts drops off dramatically from 25 to 50.

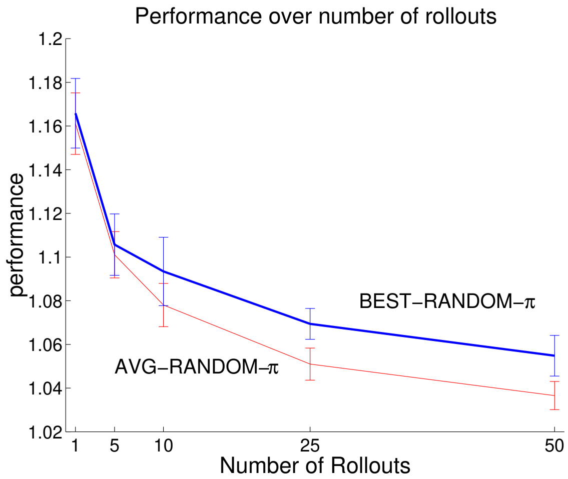


Figure 3: Simulated performance of AVG-RANDOM-π and BEST-RANDOM-π as a function of the number of rollouts as compared to the performance of DEC. The bars above and below each data point are one standard deviation from the geometric mean. A lower ratio indicates better performance.

Our hypothesis about BEST-RANDOM-π outperforming AVG-RANDOM-π was shown to be false. Choosing the instruction with the absolute best rollout schedule did not yield an improvement in performance over AVG-RANDOM-π over any number of rollouts. This is probably 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, which is in accord with the theoretical results of Bertsekas *et al.* (1997).

Although the performance of the rollout scheduler can be excellent, rollouts take a long time to run. A rollout scheduler takes $O(n^2m)$ number of basic operations, where m is the number of rollouts and n is the number of instructions. A greedy scheduler with no rollouts takes only $O(n)$. 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 for important blocks (those with long running times or which are frequently executed) 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. An 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 and 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.[3]

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,

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

V, of the value function of the current policy. At the same time, we use these estimates to update the policy. This results in a kind of generalized policy iteration (Sutton and Barto, 1998) that tends to improve the policy over time. 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:

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

where α is a positive step-size (or learning rate) parameter. This is called a backup. 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.

The RL Scheduler

As in the supervised learning results presented by 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 instruction B, the RL scheduler learned the difference of the returns resulting from choosing instruction A over 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 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 candidate instructions. 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; Utgoff and Precup, 1997; Harmon *et al.* 1995; Werbos, 1992).

Our RL scheduler learned using a feature vector calculated from the current partial schedule and two different candidate instructions. This adapted the TD algorithm to learning over pairs of instructions. Each feature was derived from knowledge of the DEC simulator. The features and our intuition for their importance are summarized in Table 4. 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. These features are general enough to help on larger blocks as well. The scheduler learned using a linear function approximator over the feature vector.

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 useful because the 21064 can only dual issue instructions if the first instruction matches the first pipeline in type and the next 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

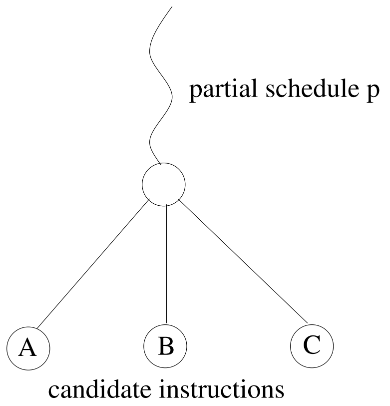
Feature Name	Feature Description	Intuition for Use
Odd Partial (`odd`)	Signifies whether the current number of instructions scheduled is odd or even.	If TRUE, we're interested in scheduling instructions that can dual-issue with the previous instruction.
Actual Dual (`d`)	Denotes whether the current instruction can dual-issue with the previous scheduled instruction or not.	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.
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.
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 and for functional units to become available.	We want to schedule instructions that will have their data and functional unit available first.

Table 4: Features for Instructions and Partial Schedule

lower bound of the block's running time can be changed. Also, this may free up more instruction 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, the feature vector was constructed 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. We used the signum (σ) of the difference of their values for the two candidate

A



B

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: Panel A shows a graphical depiction of a partial schedule and three candidate instructions. Panel B gives the feature vectors representing this situation.

instructions for these two features[4]. Thus, the feature-vector representing each triple (p,A,B), where p is a partial schedule and A and B are candidate instructions, is:

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

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 shown by in the table in Figure 4B.

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 pairs of candidate instructions (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 candidate instruction is scheduled. This process is repeated until all instructions in the block have been scheduled.

[4]Signum returns -1, 0, or 1 depending on whether the value is less than, equal to, or greater than zero.

We applied the TD algorithm (Equation 1) 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 1. 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 two reward functions to account for this variation across blocks.

Both reward functions gave zero reward until the RL scheduler had completely scheduled the block. The first final reward we used was:

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

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 similar to the reward Zhang and Dietterich (1995) used for job shop scheduling.

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.

Experimental Results

To test the RL scheduler, we used all 18 programs in the SPEC95 suite. As a baseline for our results, we also compared the performance of the RL scheduler to the performance of the RANDOM scheduler shown in Tables 2 and 3.

For both reward functions described above, we trained the RL scheduler on the application *compress95*. The parameters were $\varepsilon = 0.05$ and a learning rate, α, of 0.001. After each epoch of training, the learned value function was evaluated greedily ($\varepsilon = 0$). An epoch is one pass through the entire program scheduling all basic blocks. Figure 5 shows the results of each reward function using the same performance ratio as before and on the mean difference in cycle time between the RL scheduler and DEC across blocks without regard to how often each block is executed. Both measures give an estimation of how quickly the RL schedules were executing as compared to the DEC schedules but from different angles. These results are from the simulator.

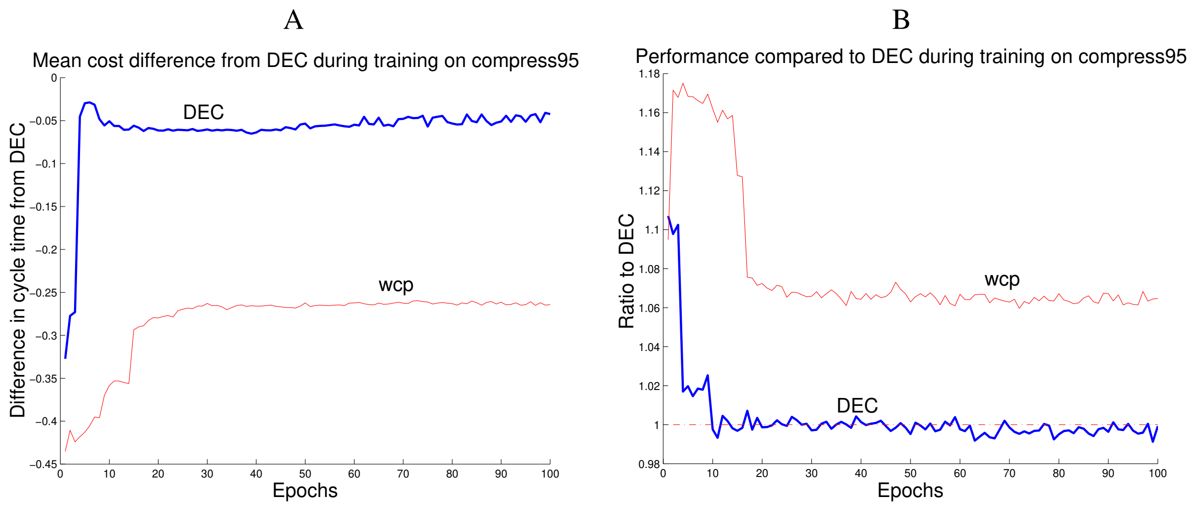


Figure 5: Panel A shows the difference in cost across all blocks in compress95 over the 100 training epochs for both of the reward functions we tested. Panel B shows the corresponding weighted performance of the system as compared to DEC for both rewards.

As the figure shows, the RL scheduler performed the best using the 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 from the 100 epochs of training to greedily schedule the other 17 benchmarks. The results are shown in Table 5. As before, we also tested the schedules on Compaq Alphas. The method was the same as for rollouts. These results are also shown in Table 5.

Performance using the DEC reward function											
Fortran programs						**C programs**					
App	**Sim**	**Real**	**App**	**Sim**	**Real**	**App**	**Sim**	**Real**	**App**	**Sim**	**Real**
applu	1.094	1.017	apsi	1.090	1.029	cc1	1.023	1.007	compress95	*0.991*	*0.967*
fpppp	1.086	1.038	hydro2d	1.036	1.002	go	1.034	*0.925*	ijpeg	1.025	1.031
mgrid	1.410	1.132	su2cor	1.035	1.012	li	1.016	1.004	m88ksim	1.012	*0.983*
swim	1.569	1.007	tomcatv	1.061	1.028	perl	1.022	*0.997*	vortex	1.035	*0.977*
turb3d	1.145	1.016	wave5	1.089	*0.994*	C geometric mean:				1.020	*0.986*
Fortran geometric mean:				1.151	1.027	Overall geometric mean:				1.090	1.009

Table 5: 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. The entries in the Sim columns are from the simulator, and the entries in the Real columns are from actual runs of the binaries. Better performance than DEC is indicated by italics.

Table 6 shows the same ratios for learning with the wcp reward function. Because the performance of the wcp scheduler was not as good as we had hoped, we tested this scheduler only in simulation.

Performance using the WCP reward function							
Fortran programs				**C programs**			
App	**Ratio**	**App**	**Ratio**	**App**	**Ratio**	**App**	**Ratio**
applu	1.215	apsi	1.199	cc1	1.092	compress95	1.060
fpppp	1.158	hydro2d	1.096	go	1.118	ijpeg	1.133
mgrid	1.504	su2cor	1.118	li	1.057	m88ksim	1.084
swim	1.651	tomcatv	1.116	perl	1.093	vortex	1.075
turb3d	1.330	wave5	1.299	C geometric mean:			1.089
Fortran geometric mean:			1.258	Overall geometric mean:			1.180

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 100 epochs, we were able to outperform DEC on compress95. The RL scheduler came within 2% of the performance of DEC on all C applications and within 15% on unseen Fortran applications. Although the Fortran performance is not as good as that on the C applications, the RL scheduler has more than halved the difference between RANDOM and DEC. 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. In actual execution of the learned schedules, the RL scheduler outperformed DEC on the C applications while coming within 2% of DEC on the Fortran applications.

Although the scheduler trained using the wcp reward function did not perform as well as the scheduler trained using the DEC reward function, it came within 18% of DEC overall. Although this is twice as slow as the performance of the scheduler trained using the DEC reward function, it is still 10% faster than RANDOM overall. The performance of the wcp scheduler may be improved by further examination of the reward function.

Because of the vast difference in generalization to Fortran programs from C programs, we also experimented with training the RL scheduler on the Fortran program applu with parameters $\alpha = 0.0001$ and $\varepsilon = 0.05$. As before, we trained for 100 epochs using the DEC based reward function and evaluated each epoch greedily ($\varepsilon = 0$) after training. We took the best value function from the 100 epochs and tested it on the other 17 benchmark applications. After testing in the simulator, we also ran each binary on a Compaq Alpha as described above. Table 7 shows the simulator and actual run-time results.

Fortran programs						**C programs**					
App	**Sim**	**Real**	**App**	**Sim**	**Real**	**App**	**Sim**	**Real**	**App**	**Sim**	**Real**
applu	1.061	1.005	apsi	1.063	1.026	cc1	1.024	1.002	compress95	1.001	1.034
fpppp	1.054	1.016	hydro2d	1.031	1.002	go	1.035	*0.980*	ijpeg	1.042	1.015
mgrid	1.204	1.053	su2cor	1.035	1.048	li	1.022	*0.996*	m88ksim	1.038	1.078
swim	1.276	*0.985*	tomcatv	1.054	1.057	perl	1.038	*0.994*	vortex	1.029	*0.999*
turb3d	1.059	*0.995*	wave5	1.078	1.009	C geometric mean:				1.029	1.012
Fortran geometric mean:				1.089	1.019	Overall geometric mean:				1.062	1.016

Table 7: Performance of the greedy RL-scheduler on each application in SPEC95 as compared to DEC using the value function trained on applu with the DEC scheduler reward function. The entries in the Sim columns are from the simulator, and the entries in the Real columns are from actual runs of the binaries.

The simulated performance on Fortran applications improved from 15% slower than DEC to only 8% slower than DEC. At the same time, performance on unseen C programs slowed by slightly less than 1%. The same is true for the run time results, where Fortran performance improved slightly while C performance dropped a small amount. This suggests the desirability of having separate schedulers for Fortran and C programs.

We also experimented with further training and testing on each of the benchmarks. Starting with the value function learned from compress95, we trained the RL scheduler 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. This is similar to a user profiling her code and then rescheduling that code based on the results of the profiling.

Fortran programs				C programs			
App	**Ratio**	**App**	**Ratio**	**App**	**Ratio**	**App**	**Ratio**
applu	1.087	apsi	1.080	cc1	1.017	compress95	0.991
fpppp	1.055	hydro2d	1.024	go	1.034	ijpeg	1.027
mgrid	1.359	su2cor	1.034	li	1.009	m88ksim	1.008
swim	1.485	tomcatv	1.053	perl	1.023	vortex	1.027
turb3d	1.090	wave5	1.078	C geometric mean:			1.017
Fortran geometric mean:			1.126	Overall geometric mean:			1.076

Table 8: Simulated performance of the greedy RL-scheduler on each application in SPEC95 after 10 epochs of training from the compress95 learned value function.

The best number from each of the 10 runs is reported in Table 8.

The 10 epochs of additional training made the performance of the RL scheduler 2% faster overall than without the additional training.

5 Combining Reinforcement Learning with Rollouts

The encouraging results of RL and of rollouts suggested testing the two methods together. Using the learned value function from training on compress95 as the rollout policy π, we tested a scheduler we call RL-π. We used only one rollout per candidate instruction and evaluated the RL policy in a greedy manner. The simulated and actual run-time results are shown in Table 9. By adding only one rollout, we were able to improve the C results to be faster than DEC overall. The Fortran results improved from 15% slower to only 4.7% slower than DEC. When executing the binaries from the RL-π schedules, a user would see slightly faster performance than DEC.

Fortran programs						C programs					
App	**Sim**	**Real**	**App**	**Sim**	**Real**	**App**	**Sim**	**Real**	**App**	**Sim**	**Real**
applu	1.017	*0.999*	apsi	1.015	1.012	cc1	1.000	1.005	compress95	*0.974*	1.014
fpppp	1.021	*0.992*	hydro2d	1.006	*0.999*	go	1.001	*0.991*	ijpeg	*0.986*	*0.986*
mgrid	1.143	1.027	su2cor	1.006	1.010	li	*0.995*	1.005	m88ksim	*0.998*	*0.979*
swim	1.176	1.003	tomcatv	1.035	*0.999*	perl	*0.996*	*0.981*	vortex	1.001	*0.984*
turb3d	1.039	*0.988*	wave5	1.025	*0.992*	C geometric mean:				*0.994*	*0.993*
Fortran geometric mean:				1.047	1.002	Overall geometric mean:				1.023	*0.998*

Table 9: Performance of the RL-π scheduler on each application in SPEC95 as compared to DEC. The entries in the Sim columns are from the simulator, and the entries in the Real columns are from actual runs of the binaries.

6 Conclusions

We have demonstrated two successful methods of building instruction schedulers for straight line code. The first method, rollouts, was able to outperform a commercial scheduler both in simulation and in actual run-time results. The downside of using a rollout scheduler is its inherently slow running time. By using an RL scheduler, we were able to maintain good performance while significantly speeding scheduling time. Lastly, we showed that a combination of RL and rollouts was able to compete with the commercial scheduler. In a system where multiple architectures are being tested, any of these methods could provide a good scheduler with minimal setup and training.

Future work could address simulator issues as well as more general instruction issues. At the simulator level, we would like to refine the feature set and adjust the model such that it can handle inner loops more realistically. Instead of only timing a block until the last instruction starts to execute, the simulator could time each block until the data from the last instruction is available. This would help a scheduler to learn to schedule loops in a better manner. At a more general level, we would like to address issues of global instruction scheduling and validating the techniques on other architectures and instruction sets. We envision using abstraction and reinforcement learning to build a global instruction scheduler.

Acknowledgments

We thank John Cavazos and Darko Stefanović for setting up the simulator and for their 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, Marc Pickett I, and James Davis for comments on earlier versions of the paper and the anonymous reviewers for their helpful comments. We also thank the Computer Science Computing Facility and the Center for Intelligent Information Retrieval for lending us 21064 machines on which to test our schedules. This work was supported in part by the National Physical Science Consortium, Lockheed Martin, Advanced Technology Labs, AFOSR grant F49620-96-1-0234 to Andrew G. Barto, and NSF grant IRI-9503687 to Roderic A. Grupen and Andrew G. Barto. We thank various people of Compaq/Digital Equipment Corporation, for the DEC scheduler and the ATOM program instrumentation tool (Srivastava & Eustace, 1994), which were essential to this work. We also thank Sun Microsystems and Hewlett-Packard for their support.

References

Abramson, B. (1990). Expected-outcome: A general model of static evaluation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, *12(2)*, 182–193.

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.

Galperin, G. (1994). Learning and improving backgammon strategy. In *Proceedings of the CBCL Learning Day*. Cambridge, MA.

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.

McGovern, A. & Moss, E. (1998). Scheduling straight-line code using reinforcement learning and rollouts. In Solla, S. (Ed.), *Advances in Neural Information Processing Systems 11*. Cambridge, MA: MIT Press.

McGovern, A., Moss, E. & Barto, A. G. (1999). Basic-block instruction scheduing using reinforcement learning and rollouts. In Dean, T. (Ed.), *Proceedings of IJCAI 1999 Workshops*. San Francisco, CA: Morgan Kaufmann Publishers, Inc.

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.

Reilly, J. (1995). SPEC describes SPEC95 products and benchmarks. SPEC Newsletter.

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. (1997a). Constructive function approximation. Technical Report 97-04, University of Massachusetts, Amherst.

Utgoff, P. E. & Precup, D. (1997b). Relative value function approximation. Technical Report UM-CS-97-003, 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.

Woolsey, K. (1991). Rollouts. *Inside Backgammon*, *1(5)*, 4–7.

Zhang, W. & Dietterich, T. G. (1995). A reinforcement learning approach to job-shop scheduling. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence* (pp. 1114–1120).