Proceedings of the 2007 Winter Simulation Conference
S. G. Henderson, B. Biller, M.-H. Hsieh, J. Shortle, J. D. Tew, and R. R. Barton, eds.

OPTIMIZING TIME WARP SIMULATION WITH REINFORCEMENT LEARNING TECHNIQUES

Jun Wang
Carl Tropper

School of Computer Science
3480 University Street, Room 318
McGill University
Montreal, QC H3A 2A7, CANADA

ABSTRACT

Adaptive Time Warp protocols in the literature are usually based on a pre-defined analytic model of the system, expressed as a closed form function that maps system state to control parameter. The underlying assumption is that this model itself is optimal. In this paper we present a new approach that utilizes Reinforcement Learning techniques, also known as simulation-based dynamic programming. Instead of assuming an optimal control strategy, the very goal of Reinforcement Learning is to find the optimal strategy through simulation. A value function that captures the history of system feedbacks is used, and no prior knowledge of the system is required. Our reinforcement learning techniques were implemented in a distributed VLSI simulator with the objective of finding the optimal size of a bounded time window. Our experiments using two benchmark circuits indicated that it was successful in doing so.

1 INTRODUCTION

Two major classes of synchronization protocols have been developed for distributed simulation. In a conservative protocol, a logical process (LP) executes the next scheduled event only when it is safe to do so. Waiting for an assurance of safety means that the LPs have to either block for some period of time or send synchronization information to each other, thereby incurring extra overhead. The most well-known optimistic protocol, Time Warp, does the opposite. In Time Warp, an LP always assumes it is safe to execute the next scheduled event. If the assumption turns out to be incorrect, the LP must undo all computations subsequent to the mistake and *roll back* to a previous state. While Time Warp has the potential of making better use of the system's parallelism, it is well known that over-optimism can lead to instability. In the worst case, the LPs spend most of their time rolling one another back, making it impossible for the simulation to progress (Lubachevsky, Shwartz, and Weiss 1991). Another major problem with Time Warp is its memory consumption due to the need to save the states of LPs as well as the events.

From the efforts to address the above problems a class of hybrid protocols have emerged, which either impart optimism to conservative protocols or limit the optimism of optimistic protocols (Das 2000). In this paper we are only concerned with adaptive Time Warp. As defined in Reynolds (1988), an adaptive protocol modifies its behavior dynamically in response to changes in the simulation process. In other words, an adaptive protocol dynamically adjusts certain control parameter(s) of the simulation in order to achieve the best performance possible. In adaptive Time Warp the control parameter is usually based on an analytic model and is expressed as a closed-form function of system state measurements. The goal of the adaptive protocol is usually to reduce excessive rollbacks. The underlying assumption is that this mapping from system state to control parameter is optimal.

In this paper, we present an approach to determining the optimal control parameter(s) for a Time Warp simulator that is based on techniques of Reinforcement Learning (RL), an area of Machine Learning. The most important factor that makes Reinforcement Learning an attractive method to optimizing Time Warp is that it requires no analytic model of the system. The control parameter is not a predefined function of simulation state. Rather, it is determined from simulation experience. In other words, instead of assuming an optimal control strategy, we attempt to acquire one through learning. Another important factor is that these techniques have very low implementation and run-time overhead. Although Reinforcement Learning techniques have been used in parallel computation, most notably in solving load balancing problems (Parent, Verbeeck, and Lemeire 2002, Schaerf, Shoham, and Tennenholtz 1995), to our knowledge, it has not been used in Time Warp.

The rest of the paper is organized as follows. Section 2 provides background information on adaptive Time Warp. Section 3 gives an overview of Reinforcement Learning. In section 4 we discuss how Reinforcement Learning techniques are used in our adaptive Time Warp protocol. In section 5 we present test results for our distributed VLSI simulator on some benchmark circuits and our analysis of the results. Finally, section 6 contains our conclusions.

2 ADAPTIVE TIME WARP

It is well known that Time Warp is prone to an excessive usage of memory and an explosive growth in the number of rollbacks. Excessive memory usage can cause performance degradation in virtual memory systems, and uncontrolled rollbacks can become dominant in the simulation process and even cause system deadlock in the most extreme cases.

An obvious solution to this problem is to limit optimism. For example, an early protocol called Moving Time Window (MTW) (Sokol, Briscoe, and Wieland 1988) controls optimism by only allowing events whose timestamps are within a certain window to be executed optimistically. The window size, however, is determined in advance and remains constant throughout the simulation. Obviously, the difficulty is to statically determine an appropriate window size before the simulation.

Adaptive Time Warp protocols go one step further. Instead of using a pre-determined constant value for the control parameter(s), they adjust the control parameters dynamically based on "knowledge of selected aspects of the state of the simulation" (Reynolds 1988). Not surprisingly, most adaptive protocols choose to control memory usage or use a time window which blocks overly optimistic executions. For example, in Panesar and Fujimoto (1997) optimism is controlled by adjusting the size of a window that defines an upper bound on the number of uncommitted events an LP can schedule; in Palaniswamy and Wilsey (1993) a moving bounded time window is defined such that only events scheduled within the window can be optimistically executed. Das (2000) provides a survey of the adaptive Time Warp protocols.

While the adaptive protocols have all been shown to outperform pure Time Warp in some applications, we feel that there are some drawback to their use.

The first problem is the use of analytic models for the control parameters. The value of a control parameter c is expressed as a closed form function of a vector $\vec{S}$ of some system state variables using a pre-determined model: $c = f(\vec{S})$. For example, the window in Panesar and Fujimoto (1997) is based on a queuing model; the checkpoint interval in Lin et al. (1993) is defined as a function of parameters such as state-saving time and event execution time. Obviously, the quality of the protocols is heavily dependent on the quality of the model. Because of the simplifications which are necessary to make in the course of developing such a model, the control strategy is not certain to be optimal. (A notable exception to using an analytic model is the adaptive protocol proposed by Ferscha (1995), where the control element of the system tries to build a statistical model of the system from simulation experience.)

Another drawback, and probably the most common one, is the lack of evaluation of the effectiveness of the control mechanism. The control function is assumed to be optimal. In RL parlance, a policy is determined in advance, but its effectiveness is not evaluated. A typical example is an adaptive protocol that tries to control rollbacks by forcing an LP to block for a short period of time based on its current local time and the global virtual time (GVT). The protocol calculates an amount of blocking time at an LP using a pre-defined function and blocks the LP in hope of controlling rollbacks. It never measures whether the blocking is effective. Worse yet, there is the danger of pursuing subgoals such as reducing the number of rollbacks while ignoring the real goal, which is to minimize simulation time.

The RL techniques we present in this paper do not require a pre-determined model. In fact, RL is known as a model-free approach. We do not assume any knowledge of what value the control parameter should assume in a system state, because that is exactly the knowledge we try to acquire. In other words, we try to optimize the system performance p as a function of the control parameter c and system state $\vec{S}$ without the closed form of a function.

3 REINFORCEMENT LEARNING

This section gives an overview of reinforcement learning (RL). For a more in-depth introduction, the reader is referred to Gosavi (2003), Sutton and Barto (1998), Kaelbling, Littman, and Moore (1996).

3.1 Definitions

Artificial Intelligence studies the interactions between a rational agent and its environment. An environment is represented by a set of states which an agent can perceive and take actions to change. Learning, "allows the agent to operate in initially unknown environments and to become more competent than its initial knowledge alone might allow" (Russell and Norvig 2003).

Reinforcement Learning, as defined in Sutton and Barto (1998), is "learning what to do–how to map situations to actions–so as to maximize a numerical reward signal". This definition describes three aspects of RL: what the agent tries to learn is what action to take in each state, the means of learning is through the reward signal, and the objective of learning is to maximize the rewards received.

More formally, we define an RL system as a five-tuple: $\{S, A, \pi, RF, VF\}$, where S is a set of states of the environment, A is a set of actions the agent can take, and the other three elements are policy, reward function, and value function, respectively.

A *policy* is a mapping from a state of the environment to an action to be taken by the agent. In other words, the policy dictates the behavior of the agent. The purpose of learning is to find an optimal policy. In general, a policy is stochastic.

A *reward function* is a mapping from the state or state-action pair of the environment to a numerical value called a reward signal, which is an indication of the desirability of the state or state-action pair. Given the central role of reward in an RL system, care must be taken to ensure that the reward function reflects what the system is designed to achieve as opposed to achieving a subgoal. For example, in a Time Warp simulator if we give a reward for reducing the number of rollbacks instead of for decreasing the execution time, the simulation could end up behaving in a conservative manner with very few rollbacks at the expense of a big simulation time.

A closely related concept is that of a return. If the sequence of rewards received after step t is $r_{t+1}, r_{t+2}, r_{t+3}, \ldots$, then the return from step t onward is:

$$R_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \ldots$$

where γ is a number, $0 \leq \gamma \leq 1$, called the discount rate. The purpose of the discount rate is twofold. First, it gives more weight to recent rewards than to future rewards in the determination of the current return. Second, it makes it possible to have a single definition of return for both episodic tasks, where the task breaks naturally into subsequences with a final state (such as playing chess), and continuing tasks with a long life span. The goal of an RL agent is to take actions to maximize expected return.

A *value function* of a given policy defines the expected return an RL agent can receive. There are two value functions of particular interest. The *state-value function* of a state s under a policy π is defined as

$$V^{\pi}(s) = E_{\pi}\{R_t | s_t = s\} = E_{\pi}\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} | s_t = s\}.$$

This is the expected return under the policy π starting from state s.

The *action-value function* of taking an action a in a state s is defined as

$$\begin{aligned} Q^{\pi}(s,a) &= E_{\pi}\{R_t | s_t = s, a_t = a\} \\ &= E_{\pi}\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} | s_t = s, a_t = a\}. \end{aligned}$$

This is the expected return under policy π, starting from taking action a in state s.

3.2 Solution Methods

We now turn to the methods used in solving an RL problem. RL has its roots in dynamic programming (DP). If the probability of reward and state transition is known, then the optimal policy can be uniquely determined with DP methods. RL is model-free, meaning we do not have any initial knowledge of the system, and we essentially try to find the optimal policy through statistical sampling. That is why RL is also referred to as simulation-based DP (Gosavi 2003). Here we shall only discuss two RL methods.

3.2.1 *N*-armed Bandit Method for Non-associated Problems

The simplest Reinforcement Learning problem is the non-associated problem, also referred to as single-state problem (Kaelbling, Littman, and Moore 1996), in which the environment has only one state. A classic example is the n-armed bandit problem in which the agent is repeatedly faced with the problem of choosing one of the n levers of a slot machine. Pulling each lever results in different payoffs (rewards) and the rewards are stochastic. The agent's goal is to acquire the maximum long term reward.

In the n-armed bandit problem, each lever (action) has an expected reward, i.e., the value of the action. Since the agent doesn't know the value of the actions in advance, a straightforward solution is to use the running average of the rewards for each action as an estimate of the action's value. Using the action-value function, when a state-action pair is selected for the $(k+1)$st time, the value is updated as:

$$Q_{k+1} = \frac{1}{k+1} \sum_{i=1}^{k+1} r_i = Q_k + \frac{1}{k+1}(r_{k+1} - Q_k), \qquad (1)$$

where r_{k+1} is the reward after the $(k+1)$st selection of the action.

This is one form of the so-called updating rule extensively used in RL. The general form is as follows:

$$\begin{aligned} NewEstimate = {} & OldEstimate + \\ & StepSize(Target - OldEstimate). \end{aligned}$$

Hereinafter we shall refer to the RL method that uses (1) as the updating rule as the *Bandit Method*.

An important issue here is the tradeoff between exploitation and exploration. At any point in the process, there is at least one action whose estimated value is the best. This action is called the *greedy action*. Exploitation

means taking the greedy action. A greedy method is one that always exploits. Exploration means taking an action other than the greedy action. The purpose of exploration is to discover other actions that might be better than the greedy action.

The ε-greedy method is one that performs both exploitation and exploration. With probability $1-\varepsilon$, where ε is a small positive number, it takes the greedy action, i.e., it exploits, and with probability ε it selects an action randomly. Experiments have shown that in general the ε-greedy method outperforms the greedy method for the n-armed bandit problem (Sutton and Barto 1998).

3.2.2 Q-learning

For problems with more than one state, we need to find out which action is the best for each state in order to maximize expected return. The solution method is referred to as *policy iteration*, which conceptually consists of repeated execution of two tasks: *policy evaluation* and *policy improvement*, as follows:

$$\pi_0 \xrightarrow{E} V^{\pi_0} \xrightarrow{I} \pi_1 \xrightarrow{E} V^{\pi_1} \xrightarrow{I} \pi_2 \xrightarrow{E} \dots$$

where $\xrightarrow{E}$ represents policy evaluation and $\xrightarrow{I}$ policy improvement. Policy evaluation computes the value functions V^{π_i} under the policy π_i. If the model of the environment is not available, then the evaluation is based on estimates. Policy improvement then refines the policy based on the value functions. This is usually done in a greedy way, i.e., the policy is modified such that the action taken in a state is the one with the greatest value at the moment.

Strictly separating policy evaluation and policy improvement can lead to very extensive computations. Therefore, Reinforcement Learning methods usually combine the two tasks for faster convergence.

Q-learning is one of the Time Difference (TD) methods, so named because the reward and the value of the current state are used to improve the estimate for the previous state. Q-learning can be expressed as follows:

$$Q(s_t,a_t) \leftarrow (1-\alpha)Q(s_t,a_t) + \alpha[r_{t+1} + \gamma \max_a Q(s_{t+1},a)], \quad (2)$$

where α is a small number (usually 0.1 or smaller) called the learning step, and γ is the discount rate. Essentially, the estimate for $Q(s_t,a_t)$, the value of the state-action pair at time t is updated using the best estimated value of the next state. In Figure 1 we show the Q-learning algorithm adopted from Sutton and Barto (1998). It should be noted that for the values to converge it is essential to ensure each state-action pair is visited enough times. This means some sort of exploration, for instance, using the ε-greedy method, has to be performed. Also, Q-learning can be applied even if there is only one state. For example, the learning method in Parent, Verbeeck, and Lemeire (2002) is one-state Q-learning.

```
Initialize Q(s,a) arbitrarily
Repeat for each episode
    Initialize s
    Repeat for each step
        Choose a from s using policy derived
                from Q (e.g., ε-greedy)
        Take action a, observe r, s'
        Q(s_t,a_t) ← (1−α)Q(s_t,a_t)+
                α[r_{t+1} + γ max_a Q(s_{t+1},a)]
        s ← s'
```

Figure 1: The Q-learning algorithm.

4 REINFORCEMENT LEARNING IN OPTIMIZING TIME WARP

Adaptive Time Warp is essentially a problem of optimal control of a stochastic process. As discussed above, the environment in an RL problem is usually stochastic and finding an optimal policy in such an environment can be regarded as an optimal control problem. This is basically the motivation for us to apply RL techniques to adaptive Time Warp.

It should be clear from the above discussion, that from the Machine Learning point of view, previous adaptive Time Warp protocols essentially adopt a pre-defined fixed policy, this is the key difference from the RL approach.

In summary, the RL approach has the following features:

- RL doesn't require an analytic or statistic model for the control parameters. In fact, RL requires no knowledge of the environment at all. The very point is to find the optimal policy.
- From control-theoretical viewpoint, RL control is closed-loop. By definition, RL is based on feedback from the environment. In fact, it does not just use the most recent feed-back; it essentially uses the whole history of feedbacks.
- The implementation and run-time overhead is low. The computation involved in updating the value functions or selecting the greedy action is minimal.

4.1 The Algorithms

To formulate optimizing Time Warp as an RL problem, we must answer the following questions.

- How many agents should there be?
- What is the control parameter?

- How do we define the reward function?
- Which RL algorithm do we use?

4.1.1 Single-agent vs. Multi-agent

Depending on the choice of control parameter, optimizing Time Warp can be formulated as a single-agent or multi-agent RL problem. If all of the nodes (a node is a group of one or more LPs running on a single CPU) always share the parameter, then there should only be one agent. On the other hand, if all of the nodes can have different values of the parameter at the same time, then multiple agents can be used.

Obviously, the multi-agent setting is much more complex than that of the single-agent, because each agent's learning process is affected by other agents, and usually some sort of coordination among the agents is required for the learning to be effective (Panait and Luke 2005).

So far in our research we have only implemented the simplest case in which there is only one agent and the system has only one state. We leave other more complex cases for future work.

4.1.2 Control Parameter

The control parameter we choose for our optimizing Time Warp is a bounded time window, similar to that in Palaniswamy and Wilsey (1993). The window size, W, together with the last GVT defines a limit for event execution. No events scheduled beyond $GVT+W$ are allowed to be executed. If the next scheduled event of an LP is beyond the window, then the LP must block. When blocked, an LP can still receive messages, but it cannot send any messages except for messages involved in GVT computation. When the GVT is updated, the window is moved forward, allowing blocked LPs to unblock if their next scheduled event falls inside of the updated window.

The purpose of the window is to control the optimism of the LPs. It prevents an LP from going too far ahead in virtual time, thereby reducing the risk of long rollbacks.

After deciding what parameter to control, we need to connect the parameter to the agent's actions. A straightforward design is for each action to represent a certain size of the window. We define an incremental unit, U, for the window, such that the window size is always a multiple of U. Then, each action a, $1 \leq a \leq N$, corresponds to adopting a window size of $a*U$.

The window size is updated right after each GVT computation. When the new GVT is obtained, all nodes send their state measurements to a single pre-designated node, which also hosts the RL agent. Based on the combined data, a new window size is determined by the agent and broadcast to all nodes.

4.1.3 Reward

The reward is the most important element in an RL system. As mentioned above, the reward should directly reflect the goal of the system. For optimizing Time Warp, since the ultimate goal is to reduce the elapsed wall-clock time of the simulation, the reward should be directly related to the speed of the simulation.

Let GVT_i denote the i-th GVT, t_i denote the wall-clock time when GVT_i is computed, and EC_i denote the number of events committed at GVT_i, We define the *event commit rate* (ECR) for the i-th GVT interval (the interval from GVT_{i-1} to GVT_i) as

$$ECR_i = EC_i/(t_i - t_{i-1}).$$

We use ECR_i as the measurement of the simulation speed during the i-th GVT interval.

To define reward, we define a reference event commit rate, ECR_{ref}, which is the average event committing rate of a period in the very beginning of the simulation with the window size set to its maximal value:

$$ECR_{ref} = (\sum_{i=1}^{D} EC_i)/(t_D - t_0),$$

where D is a small number(a value of 10 has shown to be good enough). In words, ECR_{ref} is the average number of committed events in the D GVT intervals at the beginning of the simulation.

The reward for the i-th GVT period is then defined as

$$r_i = ECR_i - ECR_{ref}.$$

Therefore, a positive reward is given if the simulation is progressing faster than the reference rate in the most recent GVT period, otherwise, a negative reward (or punishment) is given. This way, the speed of the simulation is directly reflected in the reward signal.

4.1.4 The Bandit Method

It is easy to see that in the single-agent setting the adaptive Time Warp control problem as described is quite similar to the n-armed bandit problem. The system has only one state, and each of the N values for the control parameter corresponds to an action the learning agent can take.

As in the n-armed bandit problem, we keep a running average of the rewards received for each action. After the initial period in which ECR_{ref} is established, the reward is calculated after each GVT computation, and the reward is used to update the running average for the action that was taken. Two arrays are used, $V[1..N]$ that records the current value of each action, that is, the average reward for each

action, and $K[1..N]$ records how many times each action has been taken. When an action a is taken and the reward obtained is r, the value $V[a]$ is updated using (1) as follows:

$$\begin{aligned} V[a] &= (V[a]*K[a]+r)/(K[a]+1) \\ &= V[a]+\frac{1}{K[a]+1}(r-V[a]) \end{aligned}$$

After each GVT computation, we pick a new value for the window size. Most of the time, we pick the one with the largest average reward, but with probability ε, we randomly pick a value from the N possible choices.

4.1.5 Q-learning

Currently in our implementation of Q-learning we only define one state for the system. One possible extension to multi-state is to define the states based on the difference between ECR_i and ECR_{ref}.

Using the formula in (2), after each GVT computation we compute the reward and update the value for the current state-action pair, $Q(s,a)$, using the best value of the new state s_{t+1} (since there is only one state, the new state is the same as old state). To ensure a good estimate for all of the state-action pairs, we need to allow each state-action pair to be selected enough times, and therefore again use the ε-greedy approach when selecting an action.

5 EXPERIMENTAL RESULTS

In this section we present test results of a distributed VLSI simulator on two benchmark circuits. The simulator is a Time Warp simulator with no optimizations. Aggressive cancellation is used for rollbacks.

For the bounded time window, we use the clock period as the incremental unit. The agent's action a can have a value between 1 and 8, corresponding to setting the window size to a times the clock period.

We have implemented both the Bandit method and Q-learning. The value of ε used for both cases was 0.1, and for Q-learning, γ was set to 0.9. The test circuits used were the two largest sequential circuits from the ISCAS-89 suite, with about 23,000 gates apiece. In the simulator each gate is represented by an LP.

We conducted all of the experiments on a network with 4 AMD Athlon 64 computers running the Linux operating system. The computers were connected by a fast Ethernet switch. MPICH was the underlying message system. Each circuit was given enough random test vectors for them to run for about 15 minutes.

Table 1 shows the simulation time of the two algorithms, and the speedup compared to the simulation time of pure Time Warp. Each of the simulation times was the average of 5 simulation runs. For both circuits, by using RL techniques to control the size of the bounded window we reduced the simulation time by about 15%, when compared with Time Warp.

Table 1: Running time in seconds.

Circuits	s38417		s38584	
TW	959.17		1144.68	
Bandit	808.07	15.8%	950.92	16.9%
Q-learning	823.73	14.1%	987.17	13.8%

Table 2: Action selection with Bandit method.

Action	s38417		s38584	
1	5068	82.89%	3983	78.46%
2	369	6.03%	686	13.52%
3	230	3.76%	73	1.44%
4	129	2.11%	86	1.68%
5	66	1.08%	58	1.13%
6	86	1.41%	66	1.30%
7	88	1.43%	62	1.22%
8	79	1.29%	63	1.24%

Table 3: State-action selection with Q-learning.

S-A	s38417		s38584	
0-1	4244	77.18%	2798	66.49%
0-2	404	7.35%	721	17.13%
0-3	238	4.32%	178	4.24%
0-4	174	3.16%	129	3.07%
0-5	111	2.02%	83	1.97%
0-6	117	2.12%	99	2.36%
0-7	115	2.09%	101	2.40%
0-8	96	1.75%	98	2.33%

Table 2 displays the number of times each action was selected using the single-state Bandit method, and the corresponding percentage. For s38417, action 1 was selected almost 83% of the time. From our results, this was the optimal action. The other actions were selected due to exploration with the ε-greedy method. For s38584, it seems that action 1 was optimal as well.

In Table 3 we show the number of times each state-action(S-A) pair was selected when Q-learning was used. Compared with the data in Table 2, it appears that Q-learning does slightly more exploration.

As action 1 was optimal for both circuits, we carried out experiments to find out the average *ECR* for action 1 by forcing the agent to always choose action 1. The results are 0.06527 and 0.06547 for s38417 and s38584 respectively. Figures 2 and 3 show the running average of ECR for both circuits. The average ECR converges to about 0.0625. This means that if we gradually reduce exploration, we can expect a further improvement of about 3% to 5%.

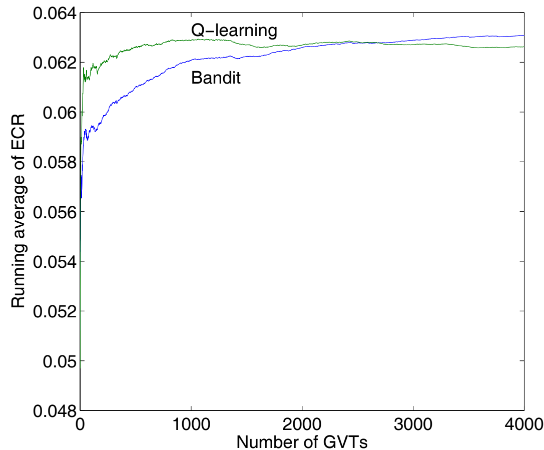

Figure 2: Average ECR with s38417.

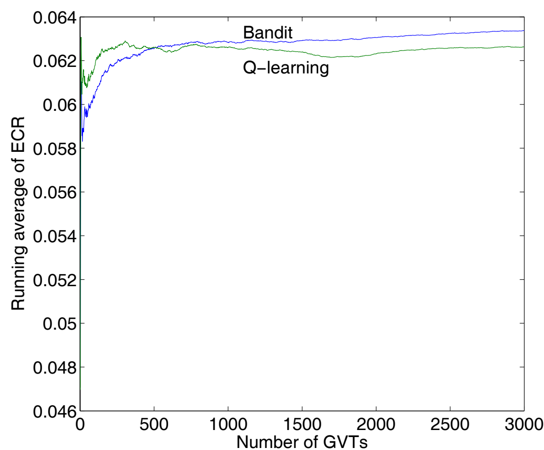

Figure 3: Average ECR with s38584.

6 CONCLUSIONS

Control techniques for adaptive Time Warp are often based on an analytic model and express the control parameter as a function of some system parameters, making their effectiveness highly dependent on the quality of the model. It is assumed that this pre-defined function provides optimal mapping from system state to the value of control parameter.

In this paper we have presented an alternative approach based on Reinforcement Learning techniques. Compared with other techniques, the RL methods require no model of the system at all. Instead of assuming an optimal control function, the very point of RL methods is to acquire one through learning. The agent has no knowledge of the system or of the control parameter it is supposed to tune. With the help of the reward signal, it learns how to behave in an optimal manner while the simulation executes. Feedback from the system is used not only as input to the mapping to the control parameter, but also in determining an optimal mapping. Furthermore, each action is taken based on the value function which encapsulates the whole history of feedbacks. Making use of Bounded Time Warp, we established that the agent could find the optimal window for simulations of two large logic circuits from the ISCAS89 benchmark suite.

For our future work, we intend to apply the RL techniques to other optimization problems for Time Warp. We also intend to explore ways in which to reduce the learning overhead. Yet another challenge is to solve the problem in a multi-agent setting.

REFERENCES

Das, S. 2000, April. Adaptive protocols for parallel discrete event simulation. *Journal of the operational research society (JORS)* 51 (4): 385–394.

Ferscha, A. 1995, July. Probabilistic adaptive direct optimism control in time warp. *Proceedings of the ninth workshop on Parallel and distributed simulation*:120–129.

Gosavi, A. 2003. *Simulation-based optimization: parametric optimization techniques and reinforcement learning*. Kluwer Academic Publishers.

Kaelbling, L., M. Littman, and A. Moore. 1996. Reinforcement learning: a survey. *Journal of artificial intelligence research* 4:237–285.

Lin, Y., B. Preiss, W. Loucks, and E. Lazowska. 1993, May. Selecting the checkpoint interval in time warp simulation. *Proceedings of the seventh workshop on Parallel and distributed simulation*:3–10.

Lubachevsky, B., A. Shwartz, and A. Weiss. 1991, April. An analysis of rollback-based simulation. *ACM transaction on modeling and computer simulation* 1 (2): 154–193.

Palaniswamy, A., and P. Wilsey. 1993, March. Adaptive bounded time windows in an optimistically synchronized simulator. *Great lakes VLSI conference*:114–118.

Panait, L., and S. Luke. 2005, November. Cooperative multi-agent learning: the state of the art. *Autonomous Agents and Multi-agent Systems* 11 (3): 387–434.

Panesar, K., and R. Fujimoto. 1997. Adaptive flow control in time warp. *Proceedings of the 11th workshop on parallel and distributed simulation*:108–115.

Parent, J., K. Verbeeck, and J. Lemeire. 2002. Adaptive load balancing of parallel applications with reinforcement learning on heterogeneous networks. *Proceedings of international symposium DCABES*.

Reynolds, P. 1988. A spectrum of options for paralle simulation. *Proceedings of the 1988 winter simulation conference*:325–332.

Russell, S., and P. Norvig. 2003. *Artificial intelligence: a modern approach*. Prentice Hall.

Schaerf, A., Y. Shoham, and M. Tennenholtz. 1995. Adaptive load balancing: a study in multi-agent learning. *Journal of artificial intelligence research* 2:475–500.

Sokol, L., D. Briscoe, and A. Wieland. 1988, July. Mtw: a strategy fo scheduling discrete simulation events for

concurrent execution. *Proceedings of the SCS multi-conference on distributed simulation* 19 (3): 34–42.

Sutton, R., and A. G. Barto. 1998. *Reinforcement learning: an introduction*. The MIT Press.

AUTHOR BIOGRAPHIES

JUN WANG is currently a PHD candidate in the School of Computer Science at McGill University, where he has been working in the area of distributed simulation of VLSI systems for the past few years. His other research interests include artificial intelligence and optimizing compilers.

CARL TROPPER is a Professor in the School of Computer Science at McGill University. His major research interest is in parallel and distributed computing. He has worked in the area of distributed discrete event simulation since the inception of the field. His focus over the past several years has been parallel VLSI simulation. His group has developed a distributed VLSI simulation environment which is being used for research in both the synchronization and performance issues associated with VLSI simulation. Another research direction is the integration of parallel continuous and discrete event simulation models.