A Multi-State Q-learning Approach for the Dynamic Load Balancing of Time Warp

Sina Meraji
School of Computer Science
McGill University
Montreal, Canada
Email: smeraj@cs.mcgill.ca

Wei Zhang
School of Computer Science
McGill University
Montreal, Canada
Email: weizhang@cs.mcgill.ca

Carl Tropper
School of Computer Science
McGill University
Montreal, Canada
Email: carl@cs.mcgill.ca

Abstract— **In this paper, we present a dynamic load-balancing algorithm for optimistic gate level simulation making use of a machine learning approach. We first introduce two dynamic load-balancing algorithms oriented towards balancing the computational and communication load respectively in a Time Warp simulator. In addition, we utilize a multi-state Q-learning approach to create an algorithm which is a combination of the first two algorithms. The Q-learning algorithm determines the value of three important parameters- the number of processors which participate in the algorithm, the load which is exchanged during its execution and the type of load-balancing algorithm. We investigate the algorithm on gate level simulations of several open source VLSI circuits.**

I. Introduction

According to Moore's law [13] the complexity of *Integrated Circuits* (IC) doubles every 18 months. A major part of the circuit design process is verification, in which the correctness and performance of the circuits are ascertained using hardware and software simulation. Hardware simulators are expensive to build and it is hard to probe the values of internal signals. Hence the verification process relies for the most part on software simulation.

Current digital circuits have millions of gates. As a result it is difficult to fit the simulation models of these circuits into a single processor's memory. In addition to the demand for memory, the need for decreased simulation time is a major challenge for the verification process. As a result, the sequential simulation of digital circuits has become a bottleneck in the design process. As a consequence, parallel discrete event simulation has emerged as a viable alternative to provide a fast, cost effective approach for the performance analysis of complex systems.

A parallel (or distributed) simulation is composed of a set of processes which are executed on different processors and which model different parts of a physical system. Each of these processes is referred as a *Logical Process* (LP). They communicate with each other via time stamped messages. It is necessary to make sure that the events in a parallel simulation are executed in the same order as they would be in a sequential simulation [7], i.e. causality must be maintained. In order to do so, the LPs must be synchronized. There are two main approaches to this synchronization: *conservative* [5] and *optimistic synchronization* [9]. Conservative simulations rely on process blocking. On the other hand, optimistic simulations process events in the order in which they arrive at an LP. No attempt is made to assure that events do not violate causality. Among the optimistic synchronization schemes Jefferson's Time Warp [9] is the most widely employed. Time Warp simulators for digital logic circuits were used in [3], [10], [11], [21]. In this paper we utilize Verilog XTW (VXTW) [11], which is XTW with a front end capable of parsing all Verilog files which are capable of being synthesized.

It is well known that in order to achieve good performance using a parallel or distributed program, it is necessary to equalize the load on the processors and to minimize the communication between the processors. Dynamic load-balancing during run-time has been studied in many literatures [1], [2], [19], [22]. The dynamic load-balancing of parallel digital simulation is examined in [4], [17] for small circuits (up to 25k gates). [4] selects clusters of LPs and moves them between processors in order to balance the load. A variation of this algorithm also minimizes the communication between processors. A central node is responsible for selecting the LPs which are transferred, a reasonable choice because the circuit sizes were no larger than 25 K gates. The algorithm was implemented on a shared memory multi-processor, resulting in a negligible communication cost for load transferring. This is not the case for current multi-computers which have distributed memory. In this paper, we introduce two new dynamic load-balancing algorithms which utilize a combination of a centralized and distributed approach to select the LPs which are to be transferred.

In this paper, we also present a protocol which selects a load-balancing algorithm and its associated parameters using a multi-state Q-learning approach. Q-learning is an area of machine learning [6]. In contrast to adaptive methods, Q-learning does not depend upon an analytical model of the system being simulated. Instead, it learns directly from experience with the system for which it is employed. In our case, the system is the parallel simulation. An attractive feature of the Q-learning algorithm is that the runtime and implementation overhead are low.

The rest of the paper is organized as follows. In section 2, we briefly discuss Verilog XTW (VXTW), our parallel Verilog simulator. In Section 3 we introduce two dynamic load-balancing algorithms for Time Warp. Section 4 introduces a multi-state Q-learning algorithm which builds upon the algorithms described in section 3. The performance analysis of the two dynamic load-balancing algorithms and the Q-learning algorithm is addressed in section 5. Finally, the last section contains our conclusion and our thoughts for future work.

II. Verilog XTW (VXTW)

XTW [21] is an object oriented simulation environment for Time Warp which makes use of a multi-queue event scheduling mechanism (XEQ) and a rollback mechanism (rb-message). XEQ has a time complexity of O(1) and rb-message uses only one anti-message for all of the messages with time-stamps which are larger than that of a straggler message. This eliminates sending individual anti-messages for all of these messages and also eliminates the need for an output queue at each LP. XTW has a superior performance to that of Clustered Time Warp.

Unfortunately, XTW can only read bench files. In order to benefit from the performance of XTW, in [11] we add a front-end to XTW which can change the data format and generates bench input data from the HDL descriptions. The architecture of VXTW is presented in figure 1. It takes a Verilog source file as the input and utilizes Synopsis DC to create GTECH [15] modules. These modules are created using GTECH library. In the next step, the Verilog Parser parses these modules and creates the bench files. Verilog has a database of rules to create the bench files.

These bench files can be input to a distributed simulator (VXTW in this paper). LPs (representing gates) are distributed among the processors. We initialize the primary inputs with a set of random input vectors in our experiments. After these steps the simulation itself can begin. The functions of the different digital gates are implemented in the simulation executive. XTW is used as the Time Warp engine. The bottom layer of VXTW is a communication layer which provides a communication interface for the processors involved in the simulation. We use *Message Passing Interface* (MPI) [14] for communication between different processors.

DC can synthesize both structural and behavioral descriptions of digital circuits into gate-level designs. The GTECH library is a standard cell library which contains over 100 basic modules. DC utilizes these basic modules in the GTECH library in order to describe the functionality of the original design. Once these GTECH modules are created, a Verilog parser is utilized to convert them to a flat bench file which is readable by XTW.

III. Dynamic Load-balancing

It has been widely observed that one of the most important factors affecting the performance of parallel programs is the distribution of load among the processors executing the program. In order to achieve the best speed up, different processors should have approximately the same load. As in [4], we define the load of a processor to be the number of events which are processed by its LPs since the last load-balance. During the course of our experiments we observed that the load on different processors in the simulation differ a lot during the simulation. Hence a dynamic load-balancing approach which can equalize the loads during a simulation is attractive.

As mentioned in the previous section, the communication time for transferring the load in [4] was negligible because a shared memory multi-processor was used as the experimental platform. As a result, we developed two new dynamic load-balancing approaches for distributed memory multiprocessor structures which we describe in the next section. We have a ring topology for the processors and utilize a combination of centralized and distributed approach to balance the load. In the following algorithms, each processor sends its load and communication information to a master node. The master node utilizes the information in order to select the processors which are to participate in the load-balancing algorithm. The selected processors determine which LPs they wish to transfer to other processors.

A. General Structure of the Algorithms

The algorithms which we introduce in this section are intended to balance the communication and computational load of the system respectively. Appropriately enough, we call them *computation* and the *communication* algorithms. In our algorithms each gate is represented by an LP. We make use of a Depth First Search (DFS) algorithm to initially distribute the LPs to processors for both of our algorithms.

We start our description by introducing four parameters which are made use of by the algorithms:

LP Computation Load (LpLoad): The computational load for each LP is defined as the number of processed events since the last load-balance of the simulation.

Processor Computation Load (PLoad): The computational load for each processor is defined as the sum of the computational loads of the LPs within that processor.

LP Communications Load (LpComm[]): The communication load for each LP is represented by an array of length $n-1$ where n is the number of processors in the system. Each element of this array is the number of messages that the LP sent to the other processors since the last load-balance of the simulation.

Processor Communication Load (PComm[]): The communication load of a processor is represented by an array of length $n-1$ where n is the number of processors. The elements of the array are the number of messages that the corresponding processor sent to other processors since the last load balance.

The load-balancing algorithm is initiated every C cycles. The type of the load-balancing algorithm (computation or communication) and the value of C are defined by the user at the beginning of the simulation. We use a combination of centralized and distributed control in the algorithms. The main structure of the algorithms is as follows: each processor sends the values of $PLoad$ and $PComm$ to a central node.

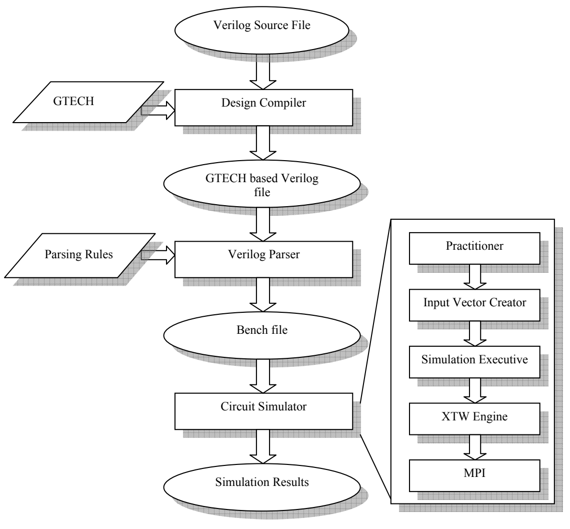


Fig. 1. The main structure of the simulator

This node matches the top $P\%$ of the over and under-loaded processors, where P is a user defined input parameter.

In the next step, for each pair of nodes which are matched together, the over-loaded node is informed about its corresponding under-loaded node. When an over-loaded node receives a notice, it selects up to L (an input parameter) of its LPs and sends them to the corresponding under-loaded processor. The next two subsections describe the details of the computation and communication load-balancing algorithms.

B. Computation Load-balancing Algorithm

The computation load-balancing algorithm utilizes $PLoad, LpLoad, PComm[]$ and $LpComm[]$ to balance the load. Each processor sends its $PLoad$ and $PComm$ values to a central node every C cycles. The central node selects the top $P\%$ of the processors which have the maximum $PLoad$ and puts them in O. The lowest $P\%$ of the processors which have the minimum $Pload$ are put in U. The processors of the sets U and O create a bipartite graph in which the weights of the edges are the values of $PComm[]$. This means that if P1 sent 1000 messages to P2 since last execution of the dynamic load-balancing algorithm, there will be a link from P1 to P2 with a weight of 1000. Basically, this graph shows the communication history of the top $P\%$ over-loaded and bottom $P\%$ under-loaded processors. We utilize a graph bipartite matching algorithm [8] to match the processors of these two sets. After this matching, the central node informs the over-loaded processors about their corresponding under-loaded processors with a Dynamic_Destination message. Whenever a processor P_i receives a dynamic destination message, it selects up to L LPs which have the most communication with the destination processor, packs them into messages and then sends them to the destination processor. It is possible that P_i later receives messages intended for LPs which were already transferred. In this case, P_i forwards these messages to their new processors.

C. Communication Load-balancing Algorithm

The communication load-balancing algorithm has the same structure as the computation load-balancing algorithm. The main difference is that it attempts to first balance the communication and then the computation. The algorithm uses $PLoad, LpLoad, PComm[]$ and $LpComm[]$ to balance the load. Every C cycles, each processor P_i sends its $PLoad_i$ and $PComm_i[]$ values to a central node. $PComm_i[j]$ contains the communication load between nodes i and j. The central node finds the maximum value of $PComm_i[j]$ among all of the values of $PComm_i[]$ that it received from different processors. If processors P_i and P_j had the most communication during last C cycles the algorithm attempts to transfer LPs between these two processors. In order to take into account the effect of the computation, the processor with the highest value of the $PLoad$ is chosen as the sender processor and a Dynamic_Destination message is sent to it. This process is continued until $2P\%$ ($P\%$ over-communicating and $P\%$ as under-communicating) of the processors are matched together.

Upon receipt of a dynamic destination message at processor P_i, it selects up to L LPs which have the most communication with the destination processor. These LPs are sent to the destination processor. As in the computation algorithm, if P_i later receives a message which belongs to the LPs which were already transferred, it forwards the message to their processor. Algorithm 2 summarizes the communication algorithm.

IV. Reinforcement Learning and the Multi-state Q-learning Algorithm

Reinforcement Learning (RL) is an area of the machine learning which is concerned with the interaction of an agent with its environment. At each interaction the agent senses the current state s of the environment, and chooses an action a to execute. This action causes changes in the environment and the environment, in its turn, sends a scalar reinforcement signal r (a reward or penalty) to the agent indicating the effectiveness of its actions. In this way, "The RL problem is meant to be a straightforward framing of the problem of learning from interaction to achieve a goal" [12]. The RL problem can be solved by dynamic programming and the optimal policy determined if the probability of rewards and state transitions are known. However, this is not often the case, and statistical sampling methods were developed. One such approach is Q-learning.

In Q-learning agents learn to act optimally in a Markovian domain by experiencing the consequences of their actions. An agent can utilize Q-learning to acquire an optimal policy using delayed rewards. The agent can find the optimal policy even when there is no prior knowledge of the effects of its actions on the environment [20]. Q-learning utilizes the reward and the best value of the current state to improve the estimate of the previous state-action pair. The Q-learning update rule is:

$$Q(s_t, a_t) \leftarrow (1-\alpha)Q(s_t, a_t) + \alpha \left[r_{t+1} + \gamma max_a Q(s_{t+1}, a)\right]. \tag{1}$$

where α and γ are the learning step and the discount rate, respectively. In order to select an action in a state, one simple solution is to select the action which has the best value. We call this method *exploitation* and the selected action the *greedy-action*. On the other hand, exploration means taking an action other than the greedy action. Exploration helps to avoid being trapped in local minima. A method to combine these two approaches is the ϵ-greedy approach which attempts to balance exploration and exploitation. The ϵ-greedy approach chooses a greedy action with probability $1-\epsilon$, where ϵ is a small number. It has been shown that the ϵ-greedy approach outperforms the greedy approach Q-learning algorithm [18]. Algorithm 3 shows the basic ϵ-greedy Q-learning algorithm.

Algorithm 1 The Q-learning

```
Repeat for each episode
  Initialize $s$
  Repeat for each step
    Choose $a$ from $s$ using policy derived from $Q(e.g., \epsilon - greedy)$
    Take action $a$, observe $r, s'$
    $Q(s_t, a_t) \leftarrow (1-\alpha)Q(s_t, a_t) + \alpha[r_{t+1} + \gamma max_a Q(s_{t+1}, a)]$
    $s \longleftarrow s'$
```

A. Q-learning and the Dynamic Load-balancing of Time Warp

As already alluded to, the advantages of using Q-learning for the dynamic load-balancing of Time Warp is that it does not need knowledge of the environment and does not need an analytical or statistical model for the environment. Instead, it develops a control policy based on a history of feedback from the environment. In addition, it does this with a low runtime overhead as well as a low implementation cost.

1) Single Agent vs. Multi-agent: RL can formulate the dynamic load-balancing of Time Warp as a single agent or multi- agent [16] problem. In a multi-agent problem, different nodes can have different values of the control parameters of a problem. In this approach the learning of one agent affects the learning of other agents-the agents have to cooperate with each other in order to find optimal values for the control parameters. In a single agent approach we have one agent and all of the nodes share the values of the control parameters. Basically, there is a central node which gathers the data from all of the nodes, runs the RL algorithm and informs the other nodes about the values of the control parameters. In this paper, we utilize the single agent approach for our RL algorithm. We leave the multi-agent approach for future work.

2) Dynamic Load-balancing Design: We make use of three control parameters for the multi-state Q-learning algorithm:

- A: The choice of dynamic load-balancing algorithm.
- P: The percentage of nodes which participate in the load-balancing algorithm.
- L: The number of LPs which are transferred from one node to another one in each cycle of the dynamic load-balancing algorithm.

Our reasoning for these choices were based upon several observations gleaned from preliminary experiments.

As to our choice of the type of load balancing algorithm note that we had implemented two dynamic load-balancing algorithms, the computation and communication algorithms. The main aim of the computation algorithm was to balance the computational load, while the communication algorithm tried to balance the communication load between the nodes. From our experimental results, we noted that the computation and communication algorithms produced different results for a different number of processors. In addition, different circuits required different algorithms.

In both the computation and communication algorithms, we make use of a parameter P, the percentage of nodes which participated in the algorithm. We noted that when we used a small number of processors (e.g. 2-6) in the computation algorithm we could not have a large value for P because the more nodes which participated in the algorithm the more LPs are transferred in each load-balancing cycle, thereby increasing the communication overhead in a small network. If this increase is more than the speed-up that we can achieve because of load-balancing, the total simulation time increases. For obvious reasons, different values of L had a significant impact on the simulation.

We define 4 states of the simulation. A state is determined by comparing the average load differences (computation and communication) between the processors to specific threshold values. If this difference is less than a certain threshold, the simulation is considered to be balanced.

- $BcompBcomm$(Balanced Computation and Balanced Communication): Both of the computation and the communication loads are balanced.
- $BcompUcomm$(Balanced Computation and Unbalanced Communication): The computation load is balanced but the communication load is unbalanced.

- $UcompBcomm$(Unbalanced Computation and Balanced Communication): The computation load is unbalanced but the communication load is balanced.
- $UcompUcomm$(Unbalanced Computation and Unbalanced Communication): Both the computation load and the communication load are unbalanced.

In the first state, both the computation and communication loads are balanced so we don't need to run the Q-learning algorithm. In the second and third states, the communication load and computation load are both unbalanced. Accordingly, we run the communication load-balancing algorithm in second state and the computation load-balancing algorithm in the third state. We also need to learn the values of L and P in these two states. If P and L have m and n different values respectively, then there are mn combinations for the control parameters and we define mn different actions in each of these two states. In the last state both the computation and communication loads are unbalanced, and as a result the algorithm should learn the values of the all of the control parameters. A has two values; it can be either communication or computation. If P and L have m and n different values respectively, then we can define $2mn$ different actions for this state.

The transition between states is performed by comparing the average difference in the computation and commutation loads between processors with two predefined threshold values, Tcomp and Tcomm. As an example, if in the first state both the computation and communication loads become higher than Tcomp and Tcomm, we transit to state 4. If just the communication load became higher than Tcomm we switch to the second state. Finally, in the event that the computation load became higher than Tcomp, we transit to the third state. This is shown in figure 2.

The RL algorithm is executed after each C cycles, where C is a user input parameter. After C cycles all of the nodes send their data to a central node which executes the RL algorithm. After computing new values for the control parameters, it broadcasts them to all of the nodes. C is not included in the learning algorithm because its value does not vary a great deal.

3) Q-learning and dynamic load balancing: The reward function is of fundamental importance to a Q-learning algorithm. If the reward function does not reflect the main goal of the system, the learning algorithm may fail to find the optimal policy. In Time Warp, the long-term goal is to reduce the simulation time. Hence the reward should be related to the wall-clock time of the simulation.

If t_i is the wall clock time at the ith cycle, C_i, the Event Commit Rate (ECR) of the ith interval (the interval from C_{i-1} to C_i) is defined as:

$$ECR_i = NC_i/(t_i - t_{i-1}), \tag{2}$$

where NC_i denotes the number of committed events in the i-th interval.

In order to define a reward, we use a reference point. We define ECR_{ref} as the average event commit rate since the beginning of the simulation:

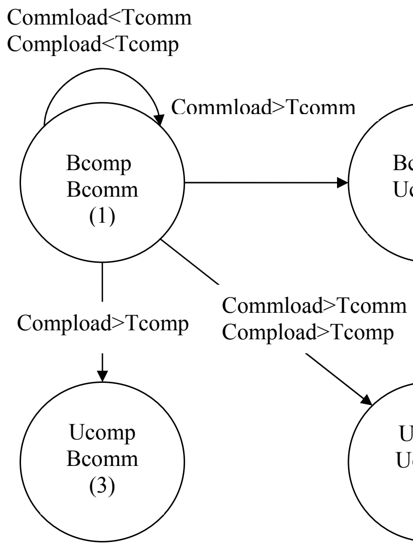


Fig. 2. The states and transition from the first state in Q-learning algorithm

$$ECR_{ref} = (\sum_{i=1}^{D} EC_i)/(t_D - t_0). \tag{3}$$

In the above formula, D is a small number between 10 and 20. This is a choice based on experimental evidence. The reward of the i-th cycle is then defined as:

$$R_i = ECR_i - ECR_{ref}. \tag{4}$$

From this definition, the reward is positive if the simulation is faster than the reference rate during the last cycle, otherwise the reward is negative. The event commit rate represents the speed of the simulation.

We use Q-learning to control the dynamic load balancing of Time Warp. As previously discussed, in different states we have a different number of actions to tune the control parameters. We utilize formula 3 as the value function. After calculating ECR_{ref}, the reward is calculated every C cycles and the value of the current state-action pair is updated. As mentioned in section 4-b, we use the best value of the new state, s_{t+1}, to update the value of the current state-action pair.

At each cycle of the dynamic load-balancing algorithm, we either select the state-action pair with the largest average reward or with a probability of ϵ we randomly pick a state-action. Algorithm 4 presents the general structure of the Q-learning algorithm.

V. Experimental Results

In this section we present performance results for the dynamic load-balancing algorithms. In all of the graphs, A=1 and 2 indicate whether the type of the load-balancing algorithm is computation or communication respectively. P is the percentage of nodes which participate in the load-balancing algorithm and L indicates the number of LPs transferred in each cycle of the algorithm. In the multi-state Q-learning

Algorithm 2 The Q-learning and Dynamic load-balancing

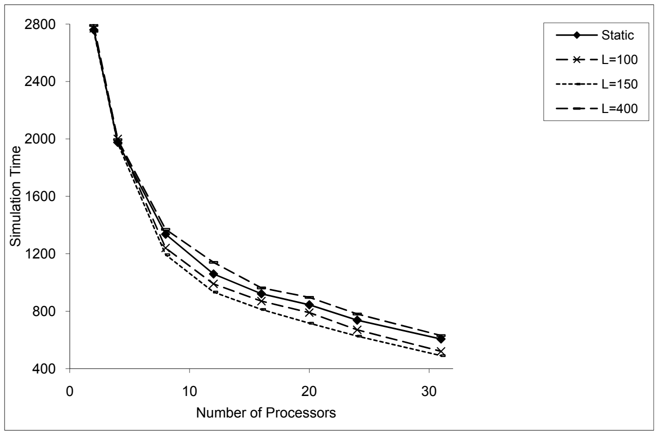
```
Master Node ($P_0$):
  {After each C cycles}
  Update the state using Tcomp and Tcomm thresholds
  if (STATE==1) then
    Skip and do the normal simulation
  else if (STATE==2) then
    A=Communication
    Run the Q-learning algorithm to select the values of $P$ and $L$
  else if (STATE==3) then
    A=Computation
    Run the Q-learning algorithm to select the values of $P$ and $L$
  else if (STATE==4) then
    Run the Q-learning algorithm to select the values of $A$, $P$ and $L$
  end if
  if (A == Computation) then
    Run the computation load-balancing algorithm
  else
    Run the communication load-balancing algorithm
  end if
```

algorithm, the values of ϵ, γ, α and C are 0.1, 0.9, 0.1 and 5 respectievly. L can have the following 4 values: 50, 100, 150,and 200 and P could be either 10% or 20%. Hence, considering the two possible values of A, 1 and 2, we will have 16 actions in state 4 and 8 actions in states 2 and 3. Each experiment result is the average of 10 simulation runs.

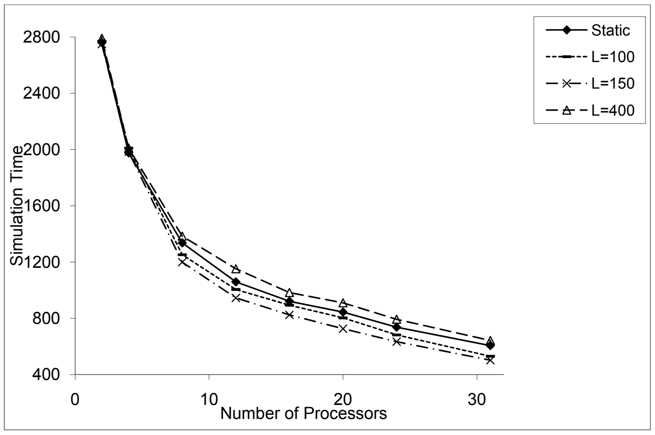
We utilized VXTW as our parallel Verilog simulator. The Verilog source files utilized in this simulation are the OpenSparc T2, the LEON processor and two Viterbi decoders designed at the Rennsalaer Polytechnic Institute (RPI). The OpenSparc and LEON designs are open source designs. LEON is a 32-bit microprocessor which is based on the SPARC-V8 RISC architecture and instruction set. It was originally designed by the European Space Research and Technology Center, part of the European Space Agency. One of the specifications of the LEON processor is its configurable core, making it suitable for System-on-Chip (SOC) designs. The LEON processor has around 200k gates. We used one core of the OpenSPARC T2 which is synthesizable by Synopsis DC and has 400k gates. The other circuits which were used in our simulations are two Viterbi decoders with 100k and 800k gates from Rensselaer Polytechnic Institute (RPI).

Our experimental platform consists of 32 dual core, 64 bit Intel processors. Each of these processors has 8 Gigabytes of internal memory. Load distribution between the two cores of a processor is automatically performed by the operating system. The processors are connected to each other by means of a 1 Gigabyte per second Ethernet. We utilized Message Passing Interface (MPI) as the communication platform between processors. MPI provides a reliable mechanism for sending and receiving messages between different processors.

Figure 3-a shows the performance of the computation dynamic load-balancing algorithm ($A = 1$) for different values of L when P =10% on the OpenSparc T2 processor. The average load difference between all of the processors is decreased by up to 60% and we achieved up to an 18% improvement in the



(a)



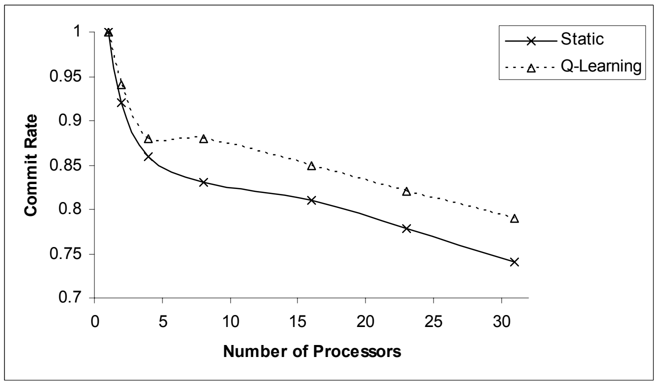
(b)

Fig. 3. The average simulation time of the computation load balancing algorithm for different values of L and P: a)P =20%, b)P =10%

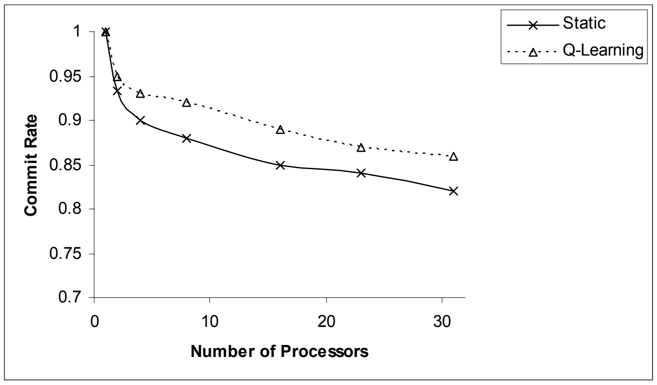
simulation time with L =150. As can be seen, increasing the number of LPs from 100 to 150 results in better performance of the algorithm. On the other hand, increasing the number of LPs to 400 worsens the situation. The reason for this is that when we transfer many LPs in each round, the communication time for transferring the LPs increases and overwhelms the performance gain which we achieved from balancing the load. Figure 3-b shows the same result when P is changed to 20%. The simulation time of the OpenSparc T2 processor is up to 4% better with P =10% than with P =20%. The reason is that when we select more nodes to send LPs the communication time for transferring the LPs increases. We do not put the results of the communication load balancing algorithm because of page limit. Different parameter values result in different simulation times for the communication load balancing algorithm as well.

The effect of changing A (type of the load-balancing algorithm), L and P on the other circuits is not shown here. However, we did many experiments on all of the circuits with different values of A, P and L. We found that for a different number of processors and for different circuits, we needed to utilize different load-balancing algorithms and their corresponding parameters to get the best performance. Hence, our major objective for the Q-learning algorithm was to learn the type of the dynamic load-balancing algorithm (A) for a specific configuration (different number of processors that participate in the load-balancing algorithm) and circuit and then to learn the corresponding parameters (P and L) for that algorithm.

Let us define the commit rate as the number of non-rollbacked messages divided by the total number of events. Figure 4-a and 4-b depicts the commit rate vs. the number of processors for the LEON and the Small RPI circuits with and without the Q-learning algorithm. The number of rollbacked messages increases with the number of processors. The reason for this is that spreading out more of the LPs among the processors results in a longer time for event cancellation. As can be seen, the learning algorithm has a better commit rate than the simulation with static load-balancing for different numbers of processors.



(a)



(b)

Fig. 4. The average commit rate for different number of processors a)LEON b)Small RPI

Figure 5-(a to d) shows the performance of different load-balancing algorithms and the Q-learning method on the large RPI circuit, the small RPI circuit, the OpenSparc T2 and LEON processors respectively. A =1, 2 correspond to computation and communication load-balancing algorithms respectively. In all of the graphs, we depict the best results which could be achieved by setting A (1 or 2), P and L. As can be seen, in almost all of the cases the Q-learning improves the simulation time more than other methods. If the Q-learning algorithm does not find a better result, its simulation time is at least as good as the best result of the other algorithms. An interesting point is the simulation time of the algorithms with two nodes. As can be seen, with two nodes the dynamic load-balancing algorithms not only cannot improve the simulation time but actually worsens the situation in some cases. The reason for this is that when we have two nodes, the communication overhead of transferring LPs is larger than the benefit we achieve from load-balancing. When we have more than four processors, the problem disappears in most of the cases and we can improve the simulation time. Using the Q-learning method, we can improve the simulation time up to 89%, 89%, 87% and 85% for large RPI, OpenSparc T2, LEON and small RPI circuits respectively.

VI. Conclusion

In this paper, we present a dynamic load balancing algorithm for Time Warp based upon reinforcement learning. A major advantage of reinforcement learning is that it does not rely upon a model; instead it depends upon learning from its environment. We introduce two dynamic load-balancing algorithms for balancing the computation and communication load during a Time Warp simulation of digital circuits described by Verilog. We utilized VXTW as our simulation engine (it is the only Time Warp simulator which can read all synthesisable Verilog circuits) and examined the performance of the two dynamic load-balancing algorithms for four large circuits. The results showed that for different circuits and different topologies (different number of processors) we needed to utilize different algorithms with different parameter values in order to obtain the best possible performance. As a result, we developed a reinforcement learning algorithm which learns to select the algorithm (communication or computation) and which adjusted the parameters of the algorithm. We utilized the multi-state Q-learning method in our implementation.

As for our future work, we plan to study the effect multi-agent technique in which we have more than one learning agent and in which the agents communicate with each other to learn the control parameters.

References

[1] Elie El Ajaltouni, Azzedine Boukerche, and Ming Zhang. An efficient dynamic load balancing scheme for distributed simulations on a grid infrastructure. In *DS-RT '08: Proceedings of the 2008 12th IEEE/ACM International Symposium on Distributed Simulation and Real-Time Applications*, pages 61–68, Washington, DC, USA, 2008. IEEE Computer Society.

[2] Shailendra S. Aote and M. U. Kharat. A game-theoretic model for dynamic load balancing in distributed systems. In *ICAC3 '09: Proceedings of the International Conference on Advances in Computing, Communication and Control*, pages 235–238, New York, NY, USA, 2009. ACM.

[3] Hervé Avril and Carl Tropper. Clustered time warp and logic simulation. *SIGSIM Simul. Dig.*, 25(1):112–119, 1995.

[4] Hervé Avril and Carl Tropper. The dynamic load balancing of clustered time warp for logic simulation. *SIGSIM Simul. Dig.*, 26(1):20–27, 1996.

[5] Yi bing Lin and Paul A. Fishwick. Asynchronous parallel discrete event simulation. *IEEE Transactions on Systems, Man and Cybernetics*, 26, 1996.

[6] J. G. Carbonell, editor. *Machine learning: paradigms and methods*. Elsevier North-Holland, Inc., New York, NY, USA, 1990.

[7] Richard M. Fujimoto. *Parallel and Distribution Simulation Systems*. John Wiley & Sons, Inc., New York, NY, USA, 1999.

[8] Harold Gabow and Robert Tarjan. Almost-optimum speed-ups of algorithms for bipartite matching and related problems. In *STOC '88: Proceedings of the twentieth annual ACM symposium on Theory of computing*, pages 514–527, New York, NY, USA, 1988. ACM.

[9] David R. Jefferson. Virtual time. *ACM Trans. Program. Lang. Syst.*, 7(3):404–425, 1985.

[10] Lijun Li, Hai Huang, and Carl Tropper. Dvs: An object-oriented framework for distributed verilog simulation. In *PADS '03: Proceedings of the seventeenth workshop on Parallel and distributed simulation*, page 173, Washington, DC, USA, 2003. IEEE Computer Society.

[11] Sina Meraji, Wei Zhang, and Carl Tropper. On the scalability of parallel verilog simulation. In *THE 38th INTERNATIONAL CONFERENCE ON PARALLEL PROCESSING (ICPP-2009)*, pages 1064–1069, 2005.

[12] Silvano Mignanti, Alessandro Di Giorgio, and Vincenzo Suraci. A model based rl admission control algorithm for next generation networks. *Next Generation Mobile Applications, Services and Technologies, International Conference on*, 0:303–308, 2008.

[13] Gordon E. Moore. Cramming more components onto integrated circuits. pages 56–59, 2000.

[14] mpi. *Message Passing Interface*. http://www-unix.mcs.anl.gov/mpi/, Accessed on January 2009.

[15] Samir Palnitkar. *Verilog®hdl: a guide to digital design and synthesis, second edition*. Prentice Hall Press, Upper Saddle River, NJ, USA, 2003.

[16] Liviu Panait and Sean Luke. Cooperative multi-agent learning: The state of the art. *Autonomous Agents and Multi-Agent Systems*, 11(3):387–434, 2005.

[17] Rolf Schlagenhaft, Martin Ruhwandl, Christian Sporrer, and Herbert Bauer. Dynamic load balancing of a multi-cluster simulator on a network of workstations. *SIGSIM Simul. Dig.*, 25(1):175–180, 1995.

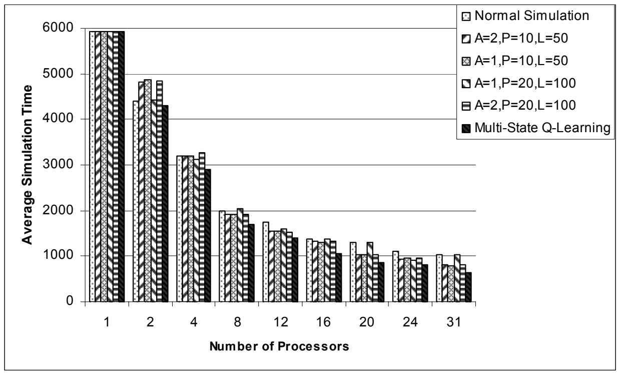
[18] R. Sutton and A. G. Barto. *Reinforcement Learning: an introduction*. The MIT Press, 2003.

[19] Xiaonian Tong and Wanneng Shu. An efficient dynamic load balancing scheme for heterogenous processing system. *Computational Intelligence and Natural Computing, International Conference on*, 2:319–322, 2009.

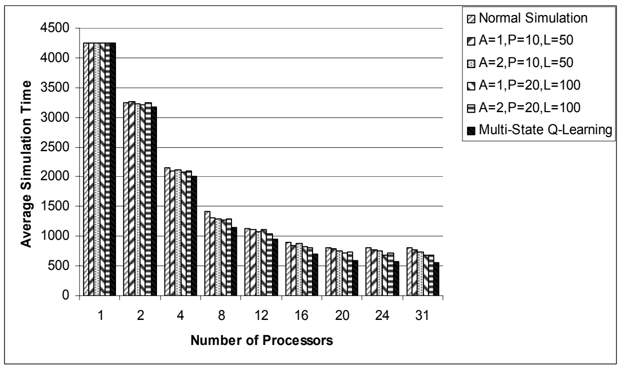
[20] Christopher J. C. H. Watkins and Peter Dayan. Q-learning. *Machine Learning*, 8(3-4):279–292, 1992.

[21] Qing XU and Carl Tropper. Xtw, a parallel and distributed logic simulator. In *ASP-DAC '05: Proceedings of the 2005 conference on Asia South Pacific design automation*, pages 1064–1069, New York, NY, USA, 2005. ACM.

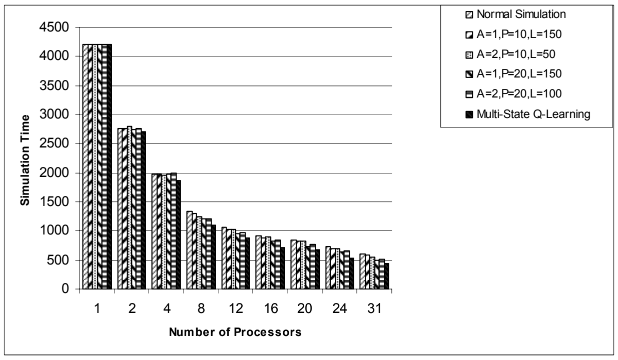
[22] BaoYin Zhang, ZeYao Mo, GuangWen Yang, and WeiMin Zheng. Dynamic load balancing efficiently in a large scale cluster. *Int. J. High Perform. Comput. Netw.*, 6(2):100–105, 2009.



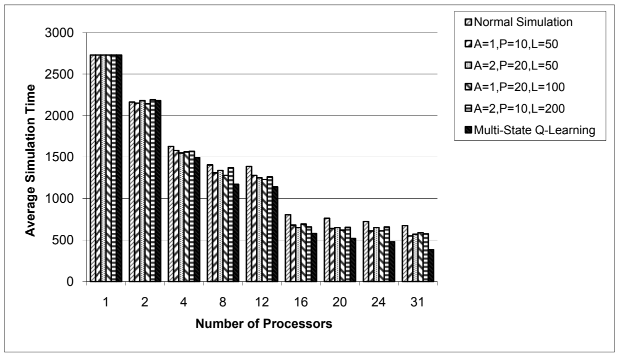
(a)



(b)



(c)



(d)

Fig. 5. The average simulation time of computation and communication load-balancing algorithms and the multi-state Q-learning method a) large RPI circuit, b) Small RPI circuit, c) OpenSparc T2 and d)LEON