On the Scalability and Dynamic Load-Balancing of Time Warp

Sina Meraji, Wei Zhang, *Member, IEEE,* and Carl Tropper, *Member, IEEE*

***Abstract*—As a consequence of Moore's law, the size of integrated circuits has grown extensively, resulting in simulation becoming the major bottleneck in the circuit design process. On the other hand, parallel or distributed simulations can be applied as fast, feasible and cost effective approaches for correctness analysis of current VLSI circuits. In this paper, we developed the first Time Warp simulator which can simulate in parallel all synthesizable Verilog circuits. We observed 4,000,000 events per second on 32 processors for the Viterbi decoder with 800k gates. We also observed that the load of different processors differ by up to 12M events during the course of the simulation. As a result, we first develop two new dynamic load balancing approach which balance the load during the simulation. Afterward, we utilize reinforcement learning to create an algorithm which is a combination of the first two algorithms. We investigate the algorithms on gate level simulations of several open source VLSI circuits. Our results show up to a 25% improvement in the simulation time using the reinforcement learning algorithm. To the best of our knowledge, this is the first time that reinforcement learning has been used for the dynamic load-balancing of Time Warp.**

***Index Terms*—Parallel Circuit Simulation, Verilog, Dynamic Load-balancing, Reinforcement Learning, Time Warp.**

I. Introduction

According to Moore's law [23] the complexity of *Integrated Circuits* (IC) will double every 18 months. *Hardware Description Languages* (HDL) such as Verilog [25] and VHDL [8] are commonly employed to design circuits. The use of an HDL speeds up the design process and the time-to-market of these circuits.

A major part of the design process is verification. Verification engineers check the correctness and performance of the circuits using hardware and software simulation. In hardware simulation, it is hard to probe the values of internal signals and expensive to build complex simulators. As a consequence, the verification process relies on software simulation.

Sequential simulation can be utilized as an accurate and inexpensive approach for the verification of digital circuits. However, as a consequence of increasing circuit size the execution time for sequential simulation is becoming prohibitive for large circuits. Current digital circuits have millions of gates and it is difficult to fit the simulation models of these circuits into a single processors' 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. At the same time, parallel discrete event simulation has emerged as a viable alternative to provide a fast, cost effective approach for the performance analysis of complex systems.

S. Meraji and Carl Tropper are with the School of Computer Science, McGill University, Montreal, QC., Canada, Their e-mails are: smeraj@cs.mcgill.ca and carl@cs.mcgill.ca

W. Zhang is with School of Computer Science, National University of Defense Technology, Changsha, China, His e-mail is: weizhang@nudt.edu.cn

Processing the events of a sequential simulation is accomplished by using a centralized priority queue of events. However, this approach cannot be extended in a straightforward manner to parallel simulation. Instead, a parallel (or distributed) simulation is composed of a set of processes which are executed on different processors and which model different parts of the physical system. Each of these processes is referred as a *Logical Process* (LP) and 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 [12], 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* [6] and *optimistic synchronization* [14]. Conservative simulations rely on process blocking, which by definition takes more time and results in deadlocks. At the other extreme, 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 [14] is the most widely employed.

Time Warp simulators for digital logic circuits were used in [4], [17], [22], [33], [21]. While some effort on distributed VHDL simulation had been made in [16], [18], [19], this was not the case for Verilog. DVS [17] was the first environment for parallel Verilog simulation. Performance analysis of XTW [33] has shown that it outperforms both Time Warp and Clustered Time Warp [4], which lay at the heart of DVS. However, XTW could not parse Verilog files. Hence we built a front-end for XTW rendering it capable of simulating any synthesizable Verilog design. We called the new simulator VXTW (Verilog XTW) [VXTW].

It is important to note that to date only small benchmark circuits and synthetic circuits were available for experimentation; our use of real designs in this paper provides more realistic benchmarks. It is also 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. As it is not possible to measure the load before a program starts, dynamic load-balancing during run-time has been employed for some time [1], [2], [31], [35]. The dynamic load-balancing of parallel digital simulation is examined in [5], [28] for small circuits (up to 25k gates). [5] 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 then 25 K gates. The algorithm was implemented on a shared memory multi-processor, resulting in a negligible communication cost for the 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 a distributed approach to select the LPs which are to be transferred.

Dynamic load-balancing is an adaptive protocol [9] for Time Warp. According to [9], an adaptive protocol is a protocol which dynamically changes its behaviour according to changes in simulation. Adaptive protocols are discussed in [10], [11], [29]. Some of these protocols control a time window in Time Warp by blocking overly optimistic execution. [5] introduced an adaptive dynamic load-balancing algorithm which improves the simulation time. While adaptive techniques improve the performance of pure Time Warp they have two main drawbacks: the usage of analytic models and the lack of evaluation of the effectiveness of the control mechanism. The quality of the protocol is highly dependent to the model and the control function is supposed to be optimal.

In this paper, we present a protocol which selects a load-balancing algorithm and its associated parameters using reinforcement learning [15]. Reinforcement learning is an area of machine learning [7]. In contrast to adaptive methods, reinforcement 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 reinforcement learning algorithms is that the runtime and implementation overhead are low. To the best of our knowledge, this is the first time that reinforcement learning has been used for the dynamic load-balancing of Time Warp.

The rest of the paper is organized as follows. In section 2, we briefly discuss Avril's [5] dynamic load-balancing algorithm and XTW. In Section 3 we detail our efforts in adding the front end parser to XTW. In Section 4 we introduce two new dynamic load-balancing algorithms for Time Warp. Section 5 introduces our learning algorithm. The performance analysis of the VXTW, dynamic load-balancing algorithms and the learning algorithm is addressed in section 6. Finally, the last section contains our conclusion and our thoughts for future work.

II. Background

A. Time Warp

Among the optimistic synchronization schemes Jefferson' Time Warp [14] is the first and the most well known one. Time Warp consists of two control mechanisms to guarantee the correctness of the simulation: Local Control and Global Control. While local controls are implemented within processors, global controls rely on a distributed computation performed by all of the processors in the system.

1) Local Control Mechanism: Each LP in Time Warp has an event list which includes all of the events which are processed or scheduled but not processed yet. This event list is referred as input queue. The events of input queue may be a result of the messages that are sent to the LP by other LPs. Each LP repeatedly removes the event with the smallest time stamp from its event list and executes it without verifying the safety of the event (optimistically). The LP may later receive a message with a time stamp smaller than the current time of the LP. This message is referred as *straggler message*. If a straggler message arrives, a rollback to the time stamp of the straggler is performed and all of the processed events with a greater time stamp than that of the straggler are re-executed.

The local control techniques try to retain the status of the system by rolling back all modified state variables and also messages sent to the other LPs. Two approaches for rolling back the state variables are copy state saving and incremental state saving [12]. Among these two, copy state saving is easier-it saves all state variables in a state queue before executing an event. The messages which were sent to other LPs are rolled back via the use of *anti-messages*. An anti-message is the exact copy of the original message with a flag which indicates it is an anti-message. Each LP saves the anti-messages in its output queue. Whenever a straggler message arrives at an LP, an anti-message for all the messages in output queue which have a time stamp larger than that of the straggler message is sent. The Anti-message annihilates its corresponding message in the input queue of the receiver LP. This is continued until all incorrect processes are rolled back.

2) Global Control Mechanism: The above discussion was an illustration of local control mechanisms. As the simulation progresses under Time Warp, more memory is consumed by the creation of new messages and also saving the status of the LPs in the state queues. As a result, we need a mechanism to reclaim the memory storage. This mechanism is referred as *fossil collection* [34]. By finding a lower bound on the time stamp of future rollbacks, we can delete the memory dedicated to events which have a smaller timestamp then this lower bound. This lower bound is called Global Virtual Time (GVT):

Global Virtual Time (GVT): $GVT(T)$ is defined as the minimum timestamp of any unprocessed message or anti-message in the system at real time T [12].

In order to compute the GVT each processor must calculate its local minimum time stamp among all its unprocessed messages and anti messages, which is referred as Local Virtual Time (LVT), and send it to a controller. The controller then computes the global minimum (GVT) among these values and broadcasts it to all the processors. It is clear that if one could take a snapshot of all unprocessed events and anti-messages in the system at simulation time T, computing $GVT(T)$ is trivial.

B. Clustered Time Warp and Dynamic Load-balancing

Avril et al. [4] designed an optimistic synchronization scheme for distributed simulation called *Clustered Time Warp* (CTW). In this approach LPs are grouped together to form a cluster. The main idea behind this is that in large scale digital circuits and network systems, LPs belonging to the same functional units can be grouped together. These LP groups are referred as a cluster. We can have any number of clusters in the model. The only restriction is that each cluster must be mapped onto one processor - it cannot be divided between processors. In CTW Time Warp is used between different processors while a sequential algorithm is employed within each cluster.

The goal of dynamic load-balancing algorithm is to evenly distribute the load between processors subject to a constraint on the communications between clusters. The load of a cluster is defined as the number of events processed by the LPs of the cluster since the last load balance. Extending this definition, the load of a processor is defined as the sum of the loads of the clusters in the processor. Load balancing is accomplished by transferring clusters from over-loaded processors to under-loaded ones. The algorithm iteratively chooses the processors with the largest and smallest loads and transfers half of the load difference from the most heavily loaded processor to the most lightly loaded processor. Clusters with approximately the same load as $(\delta load/2)$ are chosen for transfer. When the difference between the loads of the two clusters is less than a certain value, their load is assumed to be the same. For each of these clusters the change in the inter-processor communication caused by the transfer is estimated and the cluster with the lowest inter-processor communication is selected for this transfer. This algorithm is repeated iteratively until all of the processors have approximately the same load. The algorithm resulted in an improvement in processor throughput of 40% and 100% respectively on the two largest circuits from the ISCAS89 benchmark circuits.

As previously mentioned, a special node is responsible for selection of the LPs. This becomes impractical when the circuit has millions of gates. The algorithm was implemented on a shared memory multi-processor resulting in a negligible communications cost for transferring clusters between processors. Not surprisingly, the authors noted that taking inter-processor communications into account did not substantially improve the performance of the algorithm. In a distributed memory machine such as a large cluster the communications cost plays a much larger role in the performance of this algorithm, and different results should be expected.

C. XTW

In [33], Xu and Tropper developed a new event scheduling and rollback mechanism, XEQ and rb-message respectively, which improve the performance of the optimistic logic simulation. XEQ has an O(1) cost, while rb-message eliminates the computing cost of anti-messages and also reduces the memory cost by eliminating the output queue in each LP. These two techniques are incorporated in a new simulator, referred to as XTW. XTW is an object oriented simulation environment which makes it an extendable environment for Time Warp. XTW utilizes the characteristics of digital circuits and makes the following simplifying assumptions:

- Events are generated in chronological order.
- An LP receives message in chronological order.
- LPs are sparsely connected.
- The topology of LPs is static during the simulation.

The first two assumptions lead to a zero cost for sorting events while the latter two assumptions make it feasible to implement XEQ and rb-messages in large scale simulations [33].

XTW employs clusters of LPs and the LRLC [3] technique from CTW [4]. Each cluster has a multi-level event queue which is composed of three parts:

1) *Input channel* (InCh) which models a unique input of a circuit with the following rule:
   Rule 1: Each InCh can only have one unique incoming source. Each InCh itself contains one *input event queue* (ICEQ) and one *processed event queue* (ICPQ).
2) At the LP level, the event queue is referred as LPEQ, where events are sorted in increasing time stamp order.
3) At the cluster level, the event queue is referred as CLEQ where *time-buckets* are sorted in increasing time stamp order. A time-bucket is a set of events with equal time stamps.

There is a pointer at each input channel (referred as CIE) which points to the event which is de-queued from its ICEQ and is stored in the LPEQ or CLEQ. There is also a pointer at each LP (referred as CLE) which points to the event which is de-queued from its LPEQ and is stored in the CLEQ. These two pointers are used when rollback happens. According to [33] XEQ has the following rules:

Rule 2: An InCh can submit one event to its corresponding LP's LPEQ if and only if ICEQ is not empty. The pointer value of the event is assigned to CIE.

Rule 3: An LP can submit only one event to its corresponding cluster's CLEQ if and only if LPEQ is not empty. The pointer value of the event is assigned to CLE.

The steps for processing an event in XTW are as follows:

1) After an event is generated it will be sent to the corresponding InCh and appended to its ICEQ.
2) If ICEQ is not empty, the smallest time stamp event is submitted to LPEQ according to rule 2 at a cost of 1.
3) The ICEQ event is inserted to LPEQ. The cost of finding the correct position is n where n, in the worst case, is the number of InChs at an LP.
4) If LPEQ is not empty, the smallest time stamp event will be submitted to the CLEQ according to rule 3. The cost of finding the correct position is m where m is equal to the number of LPs in the cluster in the worst case.

The consequence of the above four steps is that the cost of event scheduling is constant. Using rb-messages instead of anti-messages in XTW removes the need to employ an output queue. Whenever a straggler arrives, an rb-message is send to other LPs. This message will cancel all the messages which have a time stamp larger than the time stamp of itself.

III. VERILOG XTW (VXTW)

Digital systems are now described by Hardware Description Languages (HDL) because it is easier to design, read and synthesize HDL files (making use of Electronic Design Automation (EDA) tools). In [33] the authors show that XTW has the best performance of the Time Warp based simulators which are employed for digital logic simulation. Unfortunately, XTW can only read bench files. In order to benefit from the performance of XTW, a front-end was added to it. This front-end changes the data format and generates the bench input data from the HDL descriptions. The Synopsys Design Compiler (DC) was used and a Verilog Parser was developed for the front-end. we called the new simulator Verilog XTW (VXTW) [22].

A. Synopsis Design Compiler

The Synopsis Design Compiler (DC) can synthesize structural descriptions and behavioral descriptions of digital circuits into technology-dependent, gate-level designs and also do some optimizations for both combinational and sequential logics. A wide range of design formats including Verilog description, VHDL description and EDIF netlist files are supported by DC. Here we use DC to read the Verilog source files and to generate the corresponding gate level Verilog descriptions based on the generic technology library (GTECH).

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. When these GTECH modules are created, a Verilog parser is utilized to convert them to a flat bench file which is readable by XTW.

B. Verilog Parser

The process of parsing consists of three steps: lexical analysis, syntax analysis and code generation. In the first step, we use LEX [20] as a lexical analyser. LEX uses a table of regular expressions to recognize all of the words in the input file and to generate an output stream to save these words and their types. As an example, the word ”wire” would be assumed as the beginning of a wire-type variable statement and the word ”GTECH_AO21” would be assumed as the beginning of a basic GTECH_AO21's instantiation gate.

After the lexical analysis, the words in the input file and their types are sent to a syntax analyzer and placed into different sentences. Finally, in the code generation step the corresponding code is produced from these GTECH gates according to the transformation rules. For example, a GTECH_AO21 gate would be replaced with an AND and OR gate.

C. Architecture

The main architecture of the current simulator is illustrated in Figure 2. It takes Verilog source file as the input and utilizes Synopsis DC to create GTECH 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 (XTW in this paper). The simulator itself has 5 steps. In the first step LPs (gates) are distributed between different processors. Load-balancing, concurrency and communication cost are the most important factors for partitioning. Here we use a Depth First Search (DFS) partitioning scheme which assigns the same number of LPs to different processors.

In the next step, we initialize the primary inputs with a set of random input vectors. These events generate other events. The three remaining steps perform the simulation. The functions of different digital gates are implemented in the simulation executive. XTW is used as the Time Warp engine. Finally, the bottom layer is a communication layer which provides a communication interface for the processors involved in the simulation. We use Message Passing Interface (MPI) [24] for communication between different processors.

IV. 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 on the different processors executing a program. In order to achieve the best speed up, different processors should have approximately the same load. As in [5], 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 can differ by up to 12M events 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 communications time for transferring the load in [5] 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.

A. General Structure of the Algorithms

In our algorithms each gate is represented by an LP. We make use of a DFS algorithm to initially distribute the LPs to processors for both the computation and communication algorithms.

The algorithms which we introduce in this section try to balance the communications and computational load of the system. Appropriately enough, we call them *computation* and the *communication* algorithms.

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

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

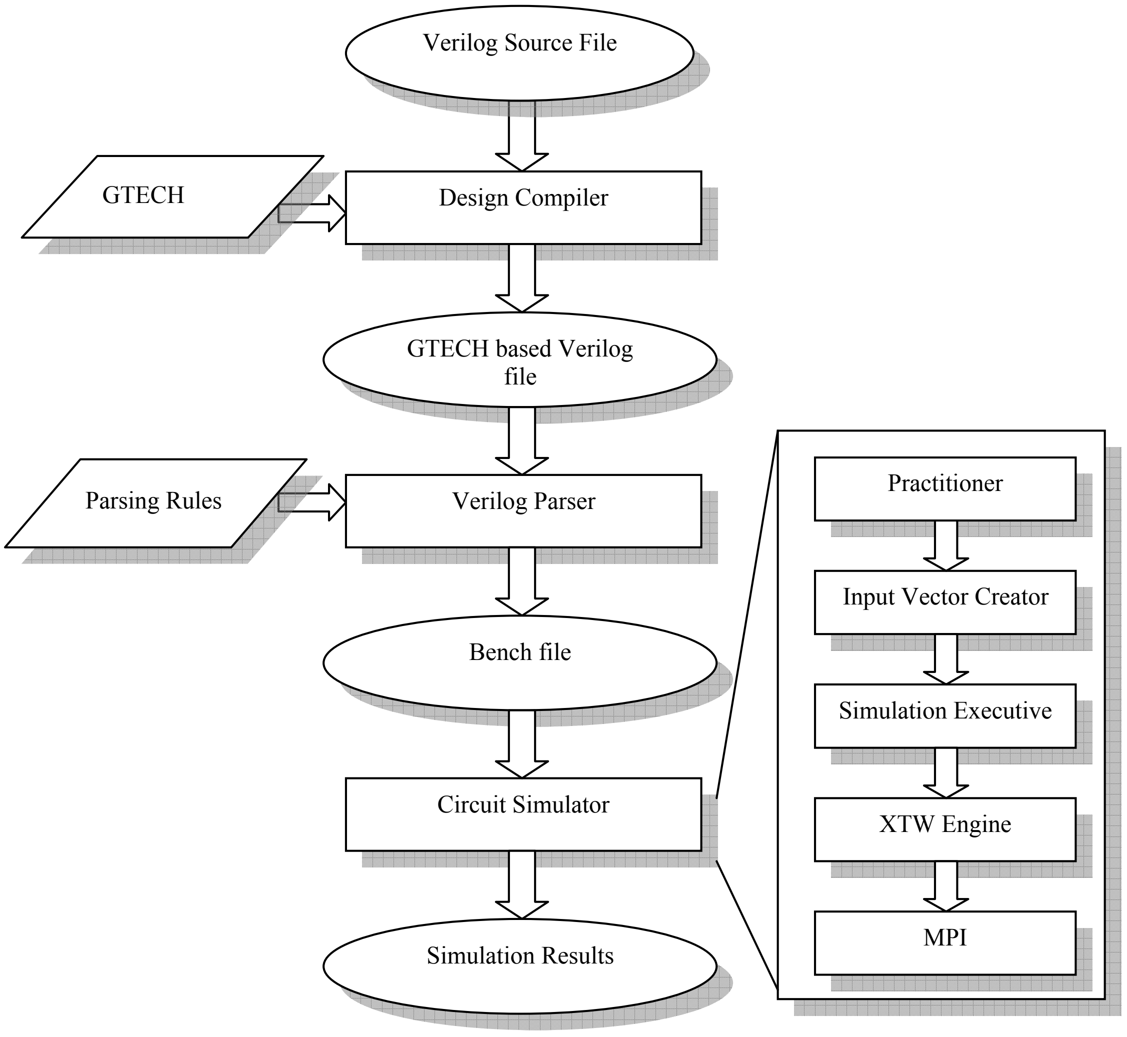


Fig. 1. The main structure of the simulator

LP Communications Load (LpComm[]): The communications load of 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 communications 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 GVT cycles. The type of the load-balancing algorithm (computation or communications) 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 value of $PLoad$ and $PComm$ to a central node. 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 communications 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 GVT 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 the FM [13] graph bipartite matching algorithm 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 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 the messages to their new processors. ALgorithm 1 summarizes the computation algorithm.

Algorithm 1 The computation load-balancing algorithm

```
Each Processor $P_i$:
  {Each C GVT cycle}
  for each LP j which $P_i$ hosts do
    $Pload_i = Pload_i + LpLoad_j$
    Send the $Pload_i$ and $PComm_i[]$ to the master node
  end for
  {Dynamic_Destination message}
  Find the top L LPs which have the maximum value of $LpComm[Destination]$
  Send the LPs to the destination processor
The Master Node:
  while number of elements in $O < P$% do
    $maxLoad = j$, where $P_j$ has the $Max\{PLoad\}$ and $P_j$ is not in $O$
    $O = O \cup maxLoad$
  end while
  while number of elements in $U < P$% do
    $minLoad = j$, where $P_j$ has the $Min\{PLoad\}$ and $P_j$ is not in $U$
    $U = U \cup P_minLoad$
  end while
  while $O! = Null$ do
    $maxLoad = j$, where $P_j$ has the $Max\{PLoad_i\}_O$
    for all the elements e in $U$ do
      if $Pcomm_{maxLoad}[e] > MaxCommiunication$ then
        $MaxCommunication = Pcomm_{maxLoad}[e]$
        $MaxCommNode = e$
      end if
    end for
    match $P_{maxLoad}$ and $P_e$
    $O = O - P_{maxLoad}$
    $U = U - P_e$
    send the Dynamic_Destination message to $P_{maxLoad}$
  end while
```

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 computation. The algorithm uses $PLoad, LpLoad, Pcomm[]$ and $LpComm[]$ to balance the load. Every C GVT 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 LPs which were already transferred, it forwards the message to their processor. Algorithm 2 summarizes the communication algorithm.

Algorithm 2 The communication load-balancing algorithm

```
Each Processor $P_i$:
  {Each C GVT cycle}
  for each LP j which $P_i$ hosts do
    $Pload_i = Pload_i + LpLoad_j$
    Send the $Pload_i$ and $PComm_i[]$ to the master node
  end for
  {Dynamic_Destination message}
  Find the top L LPs which have the maximum value of $LpComm[Destination]$
  Send the LPs to the destination processor
The Master Node:
  while $Selected < 2P$% do
    for $i = 1$ to $n - 1$ do do
      for $j = 1$ to $n - 1$ do do
        if ($MaxComm < PComm_i[j]$) and ($P_i$ and $P_j$ were not matched) then
          $Selected1 = i$
          $Selected2 = j$
          $MaxComm = PComm_i[j]$
        end if
      end for
    end for
    if $PLoad_{selected1} < Pload_{selected2}$ then
      $Sender = P_{selected1}$
    else
      $Sender = P_{selected2}$
    end if
    send the Dynamic_Destination message to the $Sender$
    $selected+ = 2$
  end while
```

V. Reinforcement Learning for Optimizing Dynamic Load-balancing

A. Introduction

Reinforcement learning is an area of the artificial intelligence which is concerned with the interaction of an agent with its environment. The agent takes actions which cause changes in the environment and the environment, in its turn, sends numerical responses to the agent indicating the effectiveness of its actions. The agent's objective is to maximize the long term sum of the numerical responses-it wants to become more competent than its initial knowledge might allow [27] it to be.

The agent and its environment can be represented by a finite-state Markov Decision Process (MDP) with state transition and reward probabilities. The MDP consists of:
1) S: a set of states of the environment.
2) A: a discrete set of actions that agent can take.
3) π: a policy which is a mapping from the environment to the action that agent takes.
4) RF: A Reward Function which maps the state (or state-action pair) of the environment to a set of scalar rewards R.
5) VF: A Value Function which defines the expected return an RL agent can receive for a given policy.

At time t, the agent chooses an action $a \in (As_t)$ depending on its current state $s_t \in S$ and the set of possible actions for that state, $A(s_t)$. The main aim of the agent is to develop an optimal policy π which maximizes the long-term reward. The reward function indicates the desirability of different states or state-action pairs. It is important for the reward function to reflect the main goal of the system and not a sub-goal. For example, in the dynamic load-balancing algorithm for Time Warp, if we give a reward for balancing the communication load instead of decreasing the simulation time, the simulation might wind up with a balanced communication load between processors and a bigger simulation time.

The concepts of return and reward are very close to each other. If the rewards received after step t are $r_{t+1}, r_{t+2}, r_{t+3}, \cdots$ then we can define R_t to be the long-term reward i from step t onwards to be:

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

where $\gamma, 0 \leq \gamma \leq 1$, is called the discount rate. The purpose of the discount rate is to give more weight to recent rewards instead of future rewards.

After defining the return function, we try to find an optimal policy which maximizes the reward. The straightforward approach is to do a brute force search which examines the returns of all possible policies and to choose the one with the maximal return value. The problem with this approach is that it becomes too costly if we have a large number of policies. The other problem is that the returns may be stochastic and, as a result, we may need a large number of samples to accurately estimate the return value of different policies.

The main two value functions are the *state-value function* and *action-value function*. The state-value function of a state s under a policy π indicates the expected return of policy π starting from state s as follows:

$$\begin{aligned} V^{\pi}(s) = \; & E_{\pi}\left\{R_t | s_t = s\right\} = \\ & = E_{\pi}\left\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} | s_t = s\right\}. \end{aligned} \tag{2}$$

The action-value function of taking an action a in a state s indicates the expected return under policy π, after taking action a in state s as follows:

$$Q_{k+1} = \frac{1}{k+1}\sum_{i=1}^{k+1} r_i = Q_k + \frac{1}{k+1}(r_{k+1} - Q_k). \tag{3}$$

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 more pragmatic methods were developed.

B. N-armed Bandit Method

In the simplest RL problem, the environment has just one state. A classic example of this problem is the N-armed bandit problem [27] in which we have a slot machine with more than one lever. The agent chooses one of the n levers of the slot machine at each step. Pulling a lever results in reward drawn from a distribution associated with that lever. The agent is assumed to have no initial knowledge about the levers. Its objective is to maximize the long-term reward.

One simple solution is to use the running average of the rewards for each action as the estimated value of that action. This is called the updating rule. The general form of the updating rule is as follows:

$$\begin{aligned} NewEstimate = \; & OldEstimate + \\ & Stepsize(Target - OldEstimate), \end{aligned} \tag{4}$$

where $Stepsize$ is inverse of the number of steps and target is the reward value of last step. If we use the action value function, when a state-action pair is selected for the $(k+1)st$ time, the value is updated as:

$$\begin{aligned} Q^{\pi}(s,a) & = E_{\pi}\{R_t | s_t = s, a_t = a\} \\ & = E_{\pi}\{\sum_{i=1}^{\infty} \gamma_k (r_{t+k+1} | s_t = s, a_t = a\}. \end{aligned} \tag{5}$$

The RL method which uses (5) as its updating rule is referred to as the bandit method. A greedy approach always chooses the best action at each step. This is the action which has the best estimated value and is known as an exploitation method. On the other hand, the exploration method chooses an action other than the greedy action with the aim of finding a better long-term reward then the one produced by a greedy policy.

One such approach 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 for the n-armed bandit problem [30].

C. Reinforcement Learning in dynamic load-balancing of Time Warp

There are a number of advantages of using RL for this problem, foremost of which are 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 single agent or multi-agent [26] problem. In a multi-agent problem, different nodes can have different values of the control parameters of the 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 to future work.

2) Control Parameters: We make use of three control parameters for our 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.

As described in section 3, we implemented two dynamic load-balancing algorithms, the computation and communication algorithms. While the main aim of the computation algorithm is to balance the computational load, the communication algorithm tries to balance the communication load between the nodes. From our experimental results, the computation and communication algorithms produce different results for a different number of processors. In addition, different circuits require different algorithms. As a result, one of the control parameters that the RL decides upon is the type of load-balancing algorithm.

In both the computation and communication algorithms, we make use of a parameter P, the percentage of nodes which participate in the algorithm. For example, when we have a small number of processors (e.g. 2-6) and we use the computation algorithm we cannot have a large value for P. Having a large P results in more nodes participating in load-balancing algorithm and more LPs being 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 will increase. Different values of L also have a significant impact on the simulation-we cannot simply make L a constant.

The above discussion indicates that the RL algorithm needs to decide about the values of these control parameters. These parameters should be connected to agent's actions. A already has two values; it could be either communication or computation. If P and L have m and n different values respectively, then there are $2mn$ combinations for the control parameters and we define $2mn$ different actions.

The RL algorithm is executed in each of the C cycles (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) The N-armed Bandit Method: The reward function is of fundamental importance to an RL algorithm. If the reward function does not reflect the main goal of the system, the RL 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 GVT, GVT_i, the Event Commit Rate (ECR) of the ith GVT interval (the interval from GVT_{i-1} to GVT_i) defined as:

$$ECR_i = NC_i/(t_i - t_{i-1}), \tag{6}$$

where NC_i denotes the number of committed events at GVT_i.

In order to define a reward, we use a reference point. As in [32], We define ECR_{ref} as the average event commit rate since the beginning of the simulation:

$$ECR_{ref} = (\sum_{i=1}^{D} EC_i)/(t_D - t_0). \tag{7}$$

In the above formula, D is a small number between 10 and 20. The reward of the i-th GVT interval is then defined as:

$$R_i = ECR_i - ECR_{ref}. \tag{8}$$

From this definition, the reward is positive if the simulation is faster than the reference rate during the last GVT interval, otherwise a punishment (negative reward) is awarded. The event commit rate represents the speed of the simulation.

We have modeled the dynamic load-balancing of Time Warp as the n-armed bandit problem. As previously discussed we have $2mn$ different combinations for the control parameters, each of which represents an action in the learning algorithm. After calculating ECR_{ref}, the reward is calculated every C GVT intervals. The running average of the action which is selected is updated with this reward and is saved. We utilize two arrays in our implementation:

- $R[N]$: The average reward for each action
- $C[N]$: The number of times that an action has been selected.

If an action a is selected and the resultant reward is r, then the value of $R[a]$ is updated as follows:

$$\begin{aligned} R[a] &= (R[a] \times C[a] + r)/(C[a] + 1) \\ &= R[a] + \frac{1}{C[a] + 1}(r - R[a]). \end{aligned} \tag{9}$$

At each cycle of dynamic load-balancing algorithm, we either pick an action with the largest average reward or with a probability of epsilon we randomly pick a value from the N actions. Algorithm 3 presents the general structure of the N-armed Bandit learning algorithm.

Algorithm 3 The N-armed Bandit learning method

```
Master Node ($P_0$):
  {After each C GVT cycles the N-armed Bandit learning method is run to select the following parameters:}
    1) A)The load balancing Algorithm: load or communication.
    2) P) Percentage of processors which participate in load balancing algorithm.
    3) L) Number of LPs that each overloaded processor should send to its corresponding under-loaded processor
  if the learning algorithm set A=Computation then
    Run the computation load-balancing algorithm (Algorithm 1)
  else
    Run the communication load-balancing algorithm (Algorithm 2)
  end if
```

VI. Experimental Results

In this section, we present performance results for the dynamic load-balancing algorithms. We study the performance of VXTW, which can parse all synthesizable Verilog files. 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).

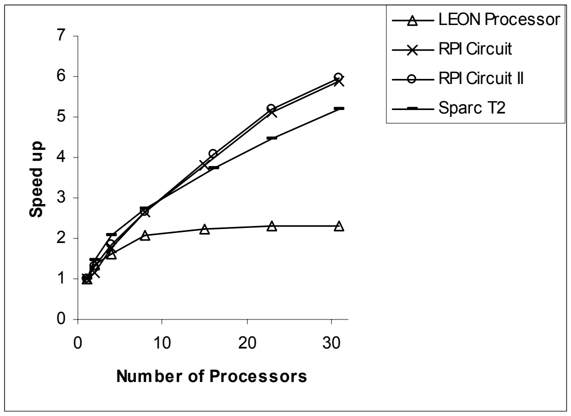
Our experimental platform consists of 32 dual core, 64 bit Intel processors. Each of these processors has 8 Gigabytes of internal memory. Load distribution between the two cores of a processor is automatically performed by the operating system. The processors are connected to each other by means of a 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.

In our simulations we assume a unit delay for gates and zero transmission time for the wires. We employ DFS partitioning with load balancing constraint for distributing the LPs (gates) between different processors. In each simulation 50 or 100 random vectors are input to the circuit. Each point in our graphs is the average of 10 simulation runs.

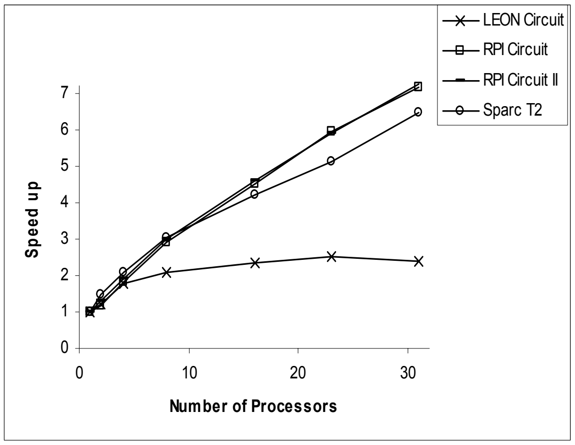
Figure 2 shows the speedup vs. the number of processors for 10 and 50 random input vectors. As we can see, the speedup increases when the circuit is larger since more events are generated. When the number of processors is increased to 10 the speedup of the LEON Processor starts to flatten out. The reason for this lies both in the structure of the LEON processor and in its size. To begin with the Leon processor is more complicated than the flat RPI designs. resulting in a larger number of rollbacks. In addition, when the number of processors increases the communication cost increases and the speed of message cancellation during a rollback decreases. The results for the RPI circuits and the OpenSparc T2 show that the speedup increases when the number of processors and the size of the circuit increases. We note that the smaller RPI circuit has the same speedup as the larger one. The reason for this is that the RPI circuits are simple flat circuits which do not exhibit much feedback. Hence there is a lower probability of receiving out of order messages, resulting in fewer rollbacks and ultimately a better speedup. A comparison between the graphs of Figure 2 (a,b) also reveals that increasing the number of vectors results in better speedup. The reason for this is that the processors are kept busy and the processor idle time decreases. The maximum speed up for the larger RPI circuit with 10 and 50 input vectors were 5.9 and 7.16, respectively.

The total number of processed events per processor is shown in Figure 3 (a,b) for 10 and 50 input vectors, respectively. The total number of processed events was 500M for the larger RPI circuit when we have 50 input vectors. If all of the 32 processors are used, the event processing rates are 1.1M, 2.1M, 2.8M and 4M events per second for the LEON, OpenSparc T2, RPI (smaller) and RPI (larger) circuits respectively when the number of input vectors is 10. For 50 input vectors, the event processing rates are 1.2M, 3M, 3.8M and 4M events per second for LEON, OpenSparc T2, RPI (smaller) and RPI (larger) circuits respectively. Increasing the number of vectors results in bigger event rates because the CPUs have a larger processing load. For the larger RPI circuit we achieve the same event processing rate when we use 10 or 50 vectors. The reason for this is that we have large number of LPs and even with fewer input vectors the processors are already busy.

Figure 4(a, b) depicts the commit rate vs the number of processors for 10 and 50 input vectors. We note that the number of rollback messages increases with the number of processors. The reason for this is that the LPs are more spread out among the processors and as a result event cancellation takes longer. We can see that the LEON processor has the smallest commit rate among the circuits. Once again, this is because of the structure and size of the circuit. More feedback results in a higher probability of receiving out of order events

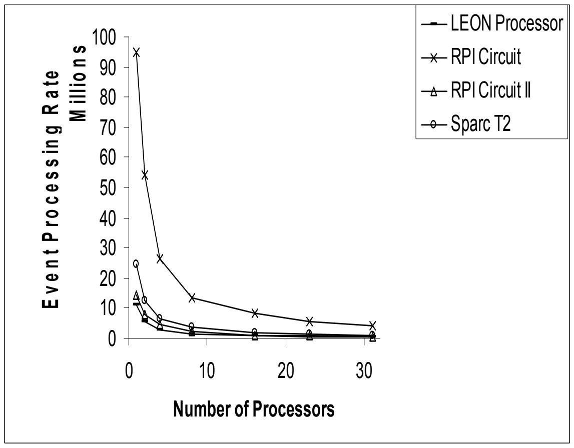


(a)

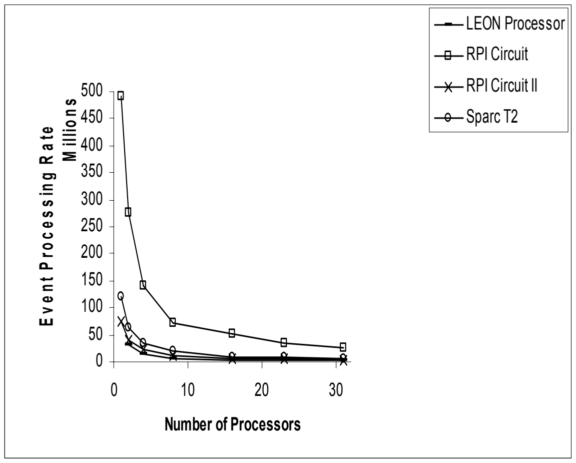


(b)

Fig. 2. Speedup vs. the number of processors (a) 10 input vectors, (b) 50 input vectors



(a)



(b)

Fig. 3. Average number of processed events per processor (a) 10 input vectors, (b) 50 input vectors

and, as a result the number of rollback messages increases and we need to send more anti-messages to cancel incorrect messages. The obvious effect of these rollback messages is the reduction of speedup as depicted in figure 2. Increasing the number of vectors also results in smaller commit rates because more events are created resulting in a larger number of out of order events.

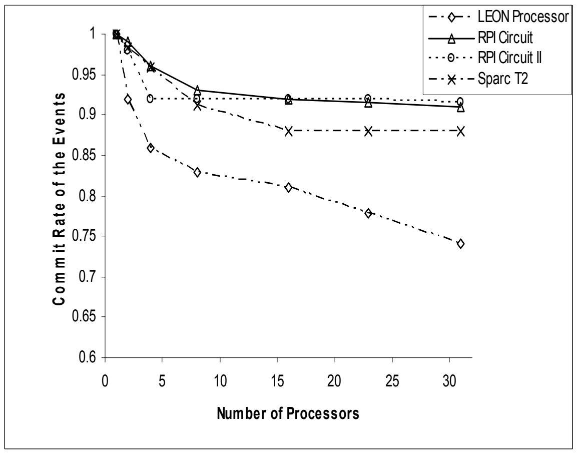
A. Dynamic Load Balancing

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. A =3 refers to the learning algorithm(Bndit method). 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 algorithm. In the bandit algorithm, the value of ϵ is 0.1. We have 8 actions, a, in each execution of the learning algorithm. Each experiment result is the average of 5 simulation runs.

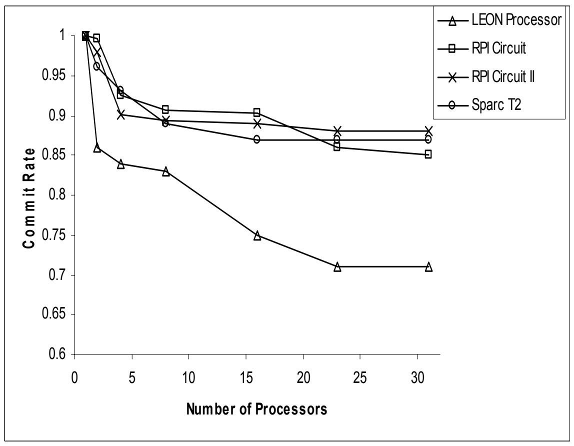
Figure 5-a shows the performance of the computation dynamic load-balancing algorithm ($A = 1$) for different values of L when P =20% on the large RPI circuit. We balanced the load up to 50% and achieved up to a 15% improvement in the simulation time with L =200. As can be seen, increasing the number of LPs from 150 to 200 results in better performance of the algorithm. On the other hand, increasing the number of LPs to 500 worsens the situation. The reason for this is that when we transfer many LPs in each round, the communication time of transferring the LPs increases and overwhelms the performance gain which we achieved from balancing the load. Figure 5-b shows the same result when P is changed to 10%. The simulation time of the large RPI circuit is up to 4% better with P =20% than with P =10%. Increasing P to more than 20% results in a worsened situation. The reason is that when we select more nodes to send LPs the communication time for transferring the LPs increases.

Figure 6-a shows the same result for the communication load-balancing ($A = 2$) algorithm with P =20% and different values of L on the small RPI circuit. We get better results by changing the value of L from 50 to 150, but if we increase L to 400 the result worsens. We can improve the simulation time by up to 17% with L =150. Figure 6-b shows the same result for P =10% and for different values of L. We improve the simulation time up to 20% with P =10% and L =150.

The effect of changing A (type of the load-balancing

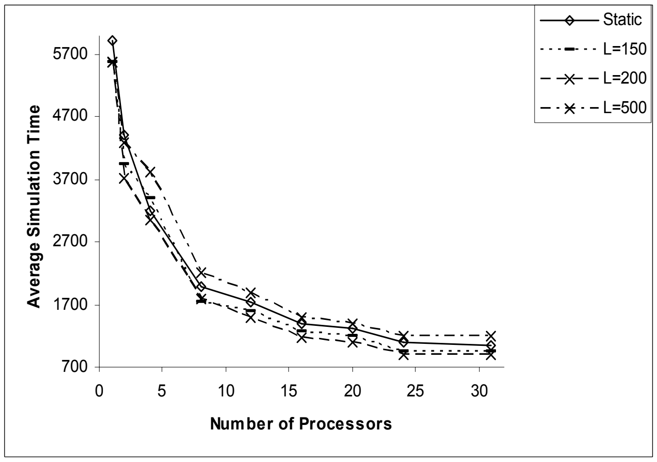


(a)

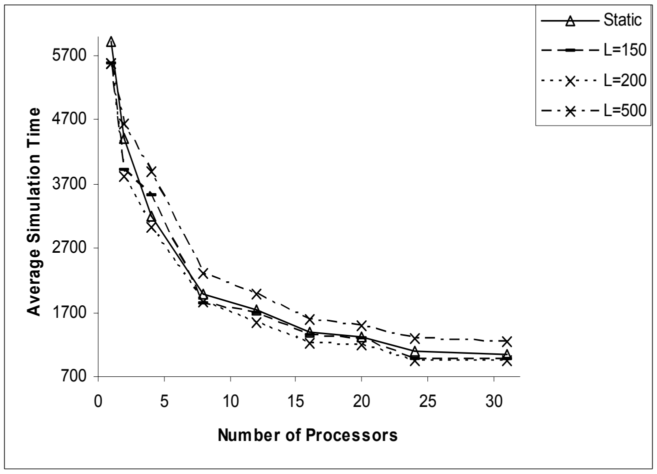


(b)

Fig. 4. Commit rate vs. different number of processors (a) 10 input vectors, (b) 50 input vectors



(a)



(b)

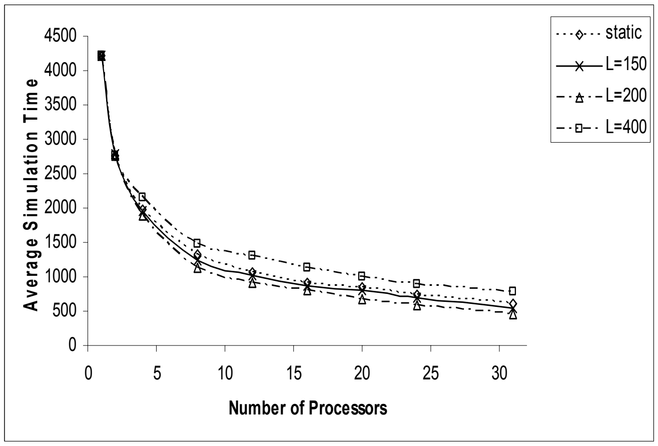
Fig. 5. The average simulation time of the computation load balancing algorithm for different values of L and P: a)P =20%, b)P =10%

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 Bandit 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) of that algorithm.

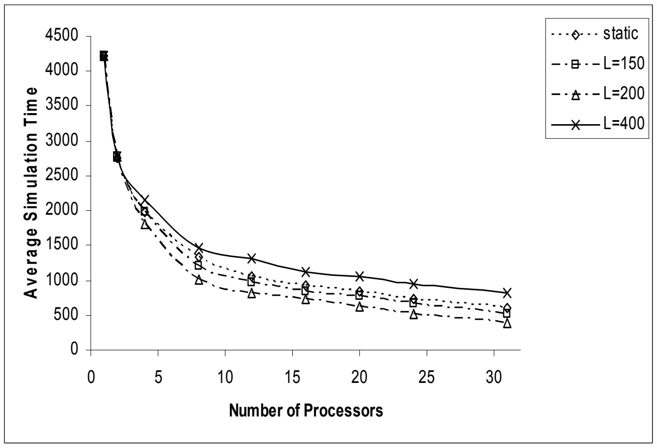
Figure 7-(a to d) shows the performance of different load-balancing algorithms and the Bandit method on the large RPI circuit, the small RPI circuit, the OpenSparc T2 and LEON processors respectively. A =1, 2, 3 correspond to computation load-balancing algorithm, communication load-balancing algorithm and the N-armed Bandit learning method 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 Bandit method (A =3) improves the simulation time more than other methods. If the Bandit method 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 Bandit method, we can improve the simulation time up to 25%, 21.5%, 24% and 21% for large RPI, OpenSparc T2, LEON and small RPI circuits respectively.

VII. Conclusion

In this paper, we developed the first parallel time warp simulator which can simulate all synthsizable Verilog circuits. We made use of XTW as the simulation engine because it has exhibited the best performance among the Time Warp based circuit simulators. We used the Synopsis Design Compiler



(a)



(b)

Fig. 6. The average simulation time of the computation load balancing algorithm for different values of L and P: a)P =20%, b)P =10%

to generate GTECH modules from Verilog source files and developed a Verilog parser to convert the GTECH modules into a flattened bench file. The performance of the simulator was evaluated with the LEON processor, the Open Sparc processor and with two Viterbi decoders designed at RPI. It is important to note that while previous work utilized small benchmark circuits and synthetic circuits, these circuits are real. We obtained an event rate of 4M events per second for the Viterbi decoder circuit on 32 processors. We noted that the speedup depended not just on the size of the circuit, but on its complexity as well-the flat RPI designs exhibited a better speedup then the more complicated open source designs. This observation may be put to use in the design of load balancing algorithms. In parallel circuit simulation the processor load changes its location throughout the course of the simulation. While some static partitioners have been shown to take effective advantage of circuit structure they do not have the ability to adjust to a change in processor load. We developed two new dynamic load balancing algorithm for parallel logic simulation. The algorithms utilize a combination of centralized and distributed approach for selecting the LPs which should be transferred.

The results of the dynamic load balancing algorithms 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 N-armed Bandit method in our implementation. The simulation results indicate that simulation time is reduced up to 25% using this approach. To the best of our knowledge, this is the first time reinforcement learning was applied to dynamic load balancing for Time Warp.

As for our future work, we plan to study the effect of another reinforcement learning method for dynamic load balancing known as Q-learning. Applying the 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 will be another focus of our research.

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] H. Avril and C. Tropper. on rolling back and checkpointing in time warp. *IEEE Transactions on Parallel and Distributed Systems*, 12(11):1105–1121, 2001.

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

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

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

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

[8] Ben Cohen. *VHDL Coding Styles and Methodologies*. Kluwer Academic Publishers, Norwell, MA, USA, 1995.

[9] Samir R. Das. Adaptive protocols for parallel discrete event simulation. In *WSC '96: Proceedings of the 28th conference on Winter simulation*, pages 186–193, Washington, DC, USA, 1996. IEEE Computer Society.

[10] Samir R. Das. Adaptive protocols for parallel discrete event simulation. In *WSC '96: Proceedings of the 28th conference on Winter simulation*, pages 186–193, Washington, DC, USA, 1996. IEEE Computer Society.

[11] Samir R. Das and Richard M. Fujimoto. An adaptive memory management protocol for time warp parallel simulation. In *SIGMETRICS '94: Proceedings of the 1994 ACM SIGMETRICS conference on Measurement and modeling of computer systems*, pages 201–210, New York, NY, USA, 1994. ACM.

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

[13] 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.

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

[15] Leslie Pack Kaelbling, Michael L. Littman, and Andrew W. Moore. Reinforcement learning: A survey. *Journal of Artificial Intelligence Research*, 4:237–285, 1996.

[16] V. Krishnaswamy and P. Banerjee. Design and implementation of an actor based parallel vhdl simulator. In *In 9th Workshop on parallel and distributed simulation(PADS95*, pages 135–143, 1995.

[17] 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.

[18] Dragos Lungeanu and C. J. Richard Shi. Parallel and distributed vhdl simulation. In *IEEE Design, Automation and Test in Europe (DATE 00*, page 658, 2000.

[19] Dale E. Martin, Radharamanan Radhakrishnan, Dhananjai M. Rao, Malolan Chetlur, Krishnan Subramani, and Philip A. Wilsey. Analysis and simulation of mixedtechnology vlsi systems. *Journal of Parallel and Distributed Computing*, 2002:468–493.

[20] Tony Mason and Doug Brown. *Lex & yacc*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1990.

[21] Sina Meraji, Wei Zhang, and Carl Tropper. On the scalability and dynamic load-balancing of parallel verilog simulations. In *Winter Simulation COnference (WSC09)*, 2009.

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

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

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

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

[26] 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.

[27] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Pearson Education, 2003.

[28] 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.

[29] Sudhir Srinivasan, Sudhir Srinivasan, Jr., Paul F. Reynolds, and Paul F. Reynolds. Npsi adaptive synchronization algorithms for pdes. In *In 1995 Winter Simulation Proceedings*, pages 658–665, 1995.

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

[31] 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.

[32] Jun Wang and Carl Tropper. Optimizing time warp simulation with reinforcement learning techniques. In *WSC '07: Proceedings of the 39th conference on Winter simulation*, pages 577–584, Piscataway, NJ, USA, 2007. IEEE Press.

[33] 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.

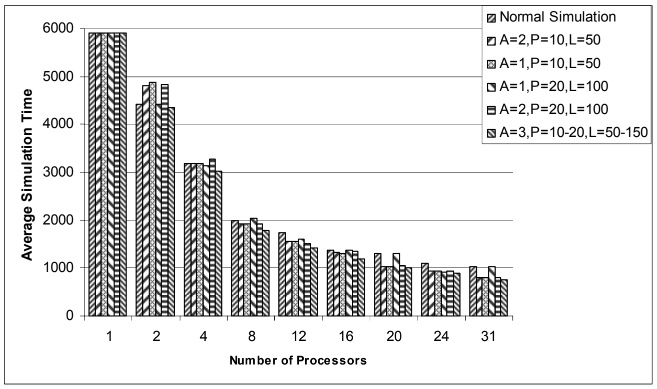
[34] Christopher H. Young and Philip A. Wilsey. Optimistic fossil collection for time warp simulation. In *Proceedings of the 29th Hawaii International Conference on System Sciences*, page 364, Washington, DC, USA, 1996. IEEE Computer Society.

[35] 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.

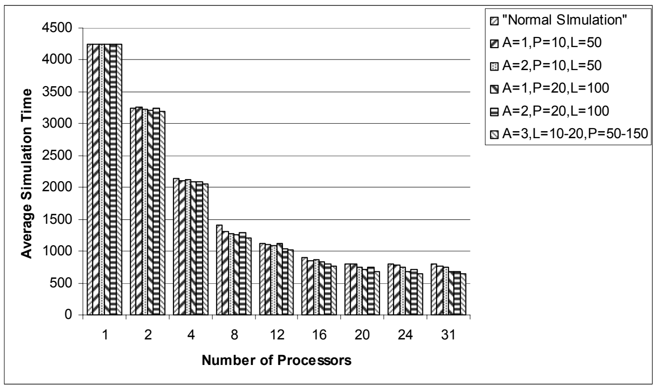
Sina Meraji is a PhD student in School of Computer Science of McGill University. His major research interest is parallel and distributed event simulation of Integrated circuits. He received his B.Sc. from computer engineering department of Amirkabir University, Iran. He also received his M.Sc. from computer engineering department of Sharif University, Iran. His email address is <smeraj@cs.mcgill.ca>.

Wei Zhang is currently visiting student in school of computer science of McGill University. he is also a Ph.D student in Computer Science, School of Computer Science, National University of Defense Technology, P.R. China. He received his B.Eng and M.Eng in Computer Science from School of Computer Science, National University of Defense Technology, P.R. China. His current research interests include Distributed Virtual Environments and Distributed Circuit Simulation. His email address is <weizhang@cs.mcgill.ca>.

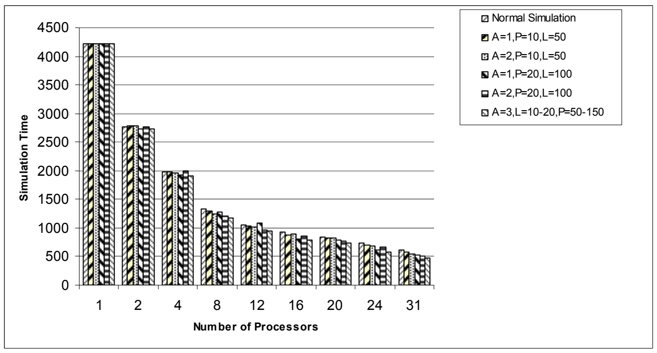
Carl Tropper is a Professor in the Department of Computer Science at McGill University. His major research interest is in parallel and distributed computing.He has worked in the area of distributed discrete event simulation since the inception of the field. His focus over the past several years has been 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. His email address is <carl@cs.mcgill.ca>.



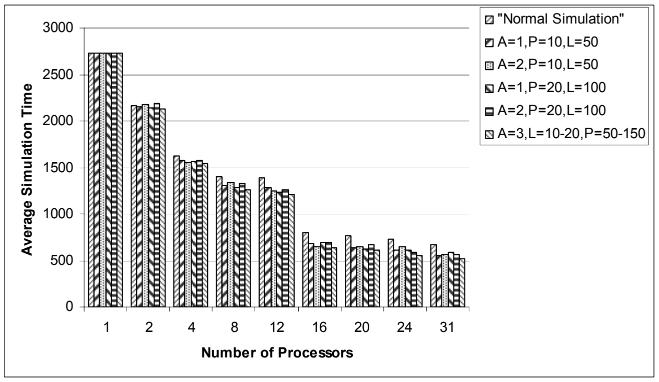
(a)



(b)



(c)



(d)

Fig. 7. The average simulation time of computation and communication load-balancing algorithms and the N-armed Bandit learning method a) large RPI circuit, b) small RPI circuit, c) OpenSparc T2 and d)LEON