A Genetic Algorithm for Optimistic Digital Logic Simulation

Sina Meraji, *Student Member, IEEE,* and Carl Tropper, *Member, IEEE*

***Abstract*—In this paper, we describe a distributed dynamic load balancing algorithm for parallel optimistic gate level simulation. Our optimistic simulator is based on Time Warp. The load balancing algorithm makes decisions based on the processing and communication loads at each processor. At the core of the algorithm is a genetic algorithm which is used to determine the values of the tuning parameters associated with the algorithm. It also determines the size of the time window of the simulator. The time window is a mechanism used to control the level of optimism of an optimistic simulator in order to avoid excessive rollbacks. An important feature of the genetic algorithm is that it is on-line, i.e. it is executed during the course of the simulation. The genetic algorithm is executed in one processor while the other processors execute the simulation and the load balancing algorithm. Experimental results have indicated a significant decrease in the execution time of the simulation- up to a 70% decrease in the simulation time of an optimistic simulator.**

***Index Terms*—Parallel Simulation, Dynamic Load Balancing, Genetic Algorithm, Logic Verification, Performance Evaluation.**

I. Introduction

An important part of the design process of digital circuits is gate level simulation, in which the correctness of the circuit is verified by simulation. Timing analysis is also performed to make sure that the circuit meets its timing constraints. In addition, the functionality of the circuit at the target frequency is also checked.

Moreover, discrete event simulation is a fundamental technique used for gate level simulation [12]. In a discrete event simulation the circuit is modeled as a graph, gates and flip-flops are represented by nodes and the wires connecting the gates and flip-flops are represented by edges of the graph. In a discrete event simulation, a change in the simulation (virtual) time occurs at discrete points in simulation time as a result of the execution of events. Changes on the wires between the gates are modeled by events. Associated with each event is the simulation time at which the event occurs. Each gate has a delay, which is the latency from changes in the gate's inputs to the corresponding changes in its outputs. In a sequential gate level simulation, events are processed in increasing time stamp order. The processing of an input event at gate g may result in a change in g's output, resulting in a new (output) event. The new event advances the simulation time from NOW to $NOW + D_g$ where D_g is g's delay.

Current integrated circuits have many millions of gates. As a result, it may take hours, days or even a week to execute the simulation [8]. On the other hand, parallel (or distributed) discrete event simulation [5], [25] runs on a collection of computers, making it possible to speed up the simulation. In a parallel gate level simulation, gates are modeled by logical processes (LPs) (Groups of gates may also be modeled by an LP). The LPs are allocated to different processors by utilizing a partitioning algorithm. Communication between LPs on different processors is accomplished by placing the events in messages which are sent between the processors. A number of simulators for parallel digital gate level simulation have been developed [3], [18], [22], [40], [19], [20].

Given that messages are processed concurrently at different LPs, it is possible to process events out of order, resulting in *causality errors*. In order to avoid these errors it is necessary to synchronize the LPs. There are two main approaches to accomplish this-*conservative synchronization* [5] and *optimistic synchronization* [14]. In the conservative approach, LPs are blocked until it is certain that no event in the system will be processed out of order [12]. The main drawback of conservative approaches is that they do not benefit from all of the parallelism which exists in the system. At the other extreme, optimistic approaches allow causality violations to occur but provide mechanisms to roll back the system to a safe state. In this paper, we utilize the Time Warp optimistic algorithm [14], [12].

In any parallel program it is advantageous to balance the computational load among the processors and to minimize the communication delay between them. Various static load balancing algorithms have been developed to accomplish this, including [1], [2], [37], [42]. In static load balancing the information with which load balancing decisions are made is known in advance while in dynamic approaches load balancing decisions are made according to the current state of the system. Large load imbalances were observed throughout the course of our experiments-hence the need for a load balancing algorithm [22].

Centralized algorithms for dynamic load balancing of parallel gate level simulations were employed in [4], [33], [23]. In these algorithms, the processors forward load information to a central processor which executes the algorithm and informs other processors how much load is to be transferred. As current circuits have many millions of gates, forwarding this load information to a central processor is very costly; it can easily create a bottleneck in the algorithm. In this paper, we propose a distributed load balancing algorithm in which each processor runs its own load balancing algorithm. We make use of an online genetic algorithm [13] to dynamically tune the parameters of the load balancing algorithm during the course

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

of the simulation.

The LPs in Time Warp can advance in simulation time at widely different rates. As a result the simulation can suffer from rollback explosions [10] and can use an excessive amount of memory. In order to avoid these problems a time window (an interval in simulation time) can be utilized to control the optimism of Time Warp [35]. Only events which have a time stamp within this interval can be processed by the LPs. When all of the LPs have executed the events within the time window, a new time window is created and events with a time-stamp within the new window can be executed. While it is possible to utilize a fixed window size [14], it is better to compute the window size dynamically, depending upon the state of the simulation [36]. Our genetic algorithm tunes the parameters of the load balancing algorithm dynamically and determines the size of the time window. Genetic algorithms are widely used to generate solutions for optimization and search problems. They use techniques inspired by natural evaluation such as inheritance, mutation, selection, and crossover. A genetic algorithm starts with a random population of candidates and tries to find an optimal solution by combining past and recent results. We design an online version of the algorithm.

The rest of this paper is organized as follows. Section 2 briefly summarizes previous work on parallel gate level simulation, dynamic load balancing algorithms for parallel gate level simulation and determining the window size. In section 3, we describe a distributed dynamic load balancing algorithm. Sections 4 and 5 discuss our genetic algorithm. Section 6 contains our experimental results and section 7 contains our conclusions and thoughts for future work.

II. Previous Work

A. Parallel Gate Level Simulation

The Verilog and VHDL languages are the two main Hardware Description Languages which have been used in computer aided design systems [9], [29]. One of the first attempts at parallel VHDL simulation is [17]. In this article, the authors describe the implementation of an object-oriented Time Warp simulator for VHDL in an actor-based environment. Other parallel VHDL simulators for gate level simulation are introduced in [39]. In [3] authors introduced Clustered Time Warp(CTW) in which LPs are grouped into a cluster and Time Warp is employed between the clusters. Based on CTW, the Distributed Verilog Simulation (DVS) [18] was the first parallel Time Warp Verilog simulator. In XTW [40], two optimization techniques were employed (RB-messages and XEQ) for parallel Time Warp gate level simulation. XTW was shown to outperform both Time Warp and CTW. The problem with XTW is that it can not parse Verilog files. In [22], a new parallel Time Warp simulator based on XTW which can parse all synthesizable Verilog files was introduced. In this paper, we make use of VXTW as our simulation environment.

B. Dynamic Load Balancing

Load balancing algorithms can be categorized as being either centralized or distributed, and being either dynamic or static [6]. A statistical approach to dynamically partition a circuit and map it to a set of processors is developed in [27]. In this work, a work graph is created to show the precedence relation between processors. The edges of the graph represent the communication information between processors. If two nodes are active at the same time they are assigned to different processors. While they achieved positive results, their test bench only had 64 gates. Moreover, the authors assumed that the network did not have direct cycles, something which is not true in general. The complexity of their algorithm is $O(E.(N-K) \times log_2(N-K))$ where N is the number of nodes in the graph, E is the number of edges and K is the number of partitions.

In Burdorf and Marti [7] objects are moved from processors which are far ahead in simulation time to processors which are far behind in simulation time.The objective of this algorithm was to decrease the number of rollbacks. Their results showed a five to ten times speed up over a simulation without load balancing. We note that their experiments were not oriented towards gate level simulation.

Schalgenhaft [33] introduces a dynamic load balancing algorithm which incorporates LPs into clusters. In order to balance the load, clusters are transferred between processors. A new metric, Virtual Time Progress(VTP), is defined which shows how fast an LP advances in virtual time. The goal of the algorithm described in this paper is to dynamically move clusters until all of the VTPs of the processors are approximately the same. The performance of this algorithm was evaluated for small circuits on just two processors. On a circuit with 20k gates a reduction in simulation time of 20% was achieved.

In some algorithms (e.g. [33], [7]), the decision to transfer load is based on the progress of an LP in virtual time. As the computational granularity is fine in gate level simulation, [4] emphasized the role of the processor load in dynamic load balancing. In [4] all of the processors send information about their load to a central processor, which makes the load balancing decisions. The performance of the algorithms was evaluated on small circuits (up to 25k gates). The results showed that the throughput improved by 40% to 100% relative to Time Warp with no load balancing. The throughput was defined as the number of non-rolled back messages per unit time. The simulation time was also reduced by up to 15%. However, Avril's approach is not applicable to current circuits with millions of gates because the (communication) cost of sending load information to a central processor is too great.

[23] describes two centralized dynamic load balancing algorithms which are used to balance the computation and the communication loads. In both of these algorithms, all of the processors in the simulation forward their computation and communication load to a central processor which, upon receipt of this information, categorizes the processors as being either over-loaded or under-loaded. The algorithm matches processors in these sets and informs the over-loaded processors about their matches, which then forward some of their LPs to their corresponding under-loaded processor. A reinforcement learning algorithm is used to tune several parameters in the load balancing algorithm. A load balancing algorithm which balances the computation and communication loads simulta-

neously is introduced in [21]. In this algorithm, all of the processors are arranged in a virtual ring and a token algorithm is utilized to distribute the load information to all of the processors. Each processor in the over-loaded set is matched with a single processor in the under-loaded set. A drawback of this algorithm lies in the one to one matching; it is more advantageous for a processor in the over-loaded set to be able to forward LPs to several processors in the under-loaded set.

C. Time Window

As the simulation progresses under Time Warp, more memory is consumed by the creation of new messages and also saving the states 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* [41]. 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 than this lower bound. This lower bound is called *Global Virtual Time (GVT)*:
Global Virtual Time (GVT): GVT(T) is defined as the minimum time-stamp of any unprocessed message or anti-message in the system at real time T [12].

As is well known, a large difference between the virtual times of the LPs in the simulation can lead to an excessive number of rollbacks. In order to control the optimism of the simulation it is possible to make use of a time window, an interval $[T, T + W]$ in virtual time. We use the size of the time window, W, and the GVT to define a bound on event execution at each LP; an event with time-stamp larger then $GVT + W$ is blocked until all of the events with time-stamp less then $GVT + W$ have been processed. A blocked LP can still receive events from other LPs but cannot process them or send messages to other LPs. The LP stays blocked until the GVT is updated. At this point, the time window is moved and the LP can execute the events within the new window. This algorithm is called Moving Time Window [34]. In this approach, the size of the window is determined in advance and remains fixed throughout the simulation. The main drawback is that the size of time window must be determined before the simulation starts.

In order to solve this problem, adaptive protocols were developed [30]. As its name indicates, an adaptive protocol changes parameters dynamically during the simulation based on "knowledge of selected aspects of the state of the simulation" . In [28] a moving bounded time window is defined such that just the events within the time window can be optimistically executed. in [38] an offline genetic algorithm is described which tunes the size of time window during the simulation. Using this algorithm the simulation time was reduced from 9.9% to 35.6% for the tested circuits. The cost of the algorithm was not considered in this paper.

D. Contributions

In this paper, we utilize a distributed load balancing algorithm and show that it has a better performance than the other dynamic load balancing algorithms for parallel gate level simulation. The new algorithm combines the computation and communication load balancing algorithms in [23]. In this algorithm, each processor in the overloaded set forwards messages to every processor in the under-loaded set. Experimental results indicated that the performance of this algorithm was highly dependent on tuning parameters-different circuits and platforms require different values for these parameters in order to get the best result. Hence, in this paper we present an online genetic algorithm which tunes the parameters of the dynamic load balancing algorithm. The genetic algorithm is also used to tune the size of the time window used in our Time Warp simulations.

III. A Distributed Dynamic Load Balancing Algorithm

This section contains a description of the dynamic load balancing algorithm. The algorithm we describe in this paper differs from the one described in [23] in that it uses a genetic algorithm to tune a different choice of parameters, and that it uses a combination of the communication and computational load balancing algorithm. The use of the current algorithm resulted in up to a 50% improvement in the simulation time when compared to the algorithm(s) described in [23]. A comparison of these approaches is described in the experimental results section.

A. Initialization

All of the gates in the circuit are mapped to Logical Processes(LPs) which are distributed among the processors in the system. We first distribute these LPs by means of a Depth First Search (DFS) partitioning algorithm. In the DFS algorithm, we impose the constraint that each processor should have the same number of LPs. The dynamic load balancing algorithm is initialized every C cycles, where C is a user input parameter. We make use of following quantities for the algorithm.

LP Computation Load (LpComp): The computation load of each LP is defined as the number of events processed since the last execution of the load balancing algorithm.

Processor Computation Load (PComp): The sum of the computation loads of the LPs within a processor is defined as the computation load of that processor.

LP Communications Load (LpComm[]): The communication load of an LP is defined to be the number of messages that the LP has sent to other processors since the last execution of the dynamic load balancing algorithm. If we have N processors, an array of size N-1 exhibits the communications load for each LP.

Processor to Processor Communication Load (PPComm[]): The number of messages that each processor has sent to the other processors since the last execution of the load balancing algorithm. For N processors, an array of size N-1 contains PPComm[] for each processor.

Processor Communication Load (PComm): This is the number of messages that each processor sent to the other processors.

The Computation-Communication weight (λ):. This represents the weight of the computation and communication loads for calculating the final load of the processor. The computation-communication weight has a value between 0 and 1.

Processor load (PLoad): This represents the load on the processor. PLoad is defined as the weighted sum of the computation and communication loads as follows:

$$PLoad = \lambda * PComp + (1 - \lambda)PComm \quad (1)$$

The dynamic load balancing algorithm is initiated every C cycles. Every C cycles, PComm is calculated as the sum of the all of the values in PPComm[] and PComp is calculated as the sum of all of the LpComps of the LPs within that processor. Finally, the processor load is calculated using formula 1 and each processor broadcasts its PLoad information to all of the other processors in the system. Each processor updates its local copy of the other processors load when it gets the new information. After receiving all of the updated information, each processor starts the dynamic load balancing algorithm. Algorithm 1 summarizes the initialization step.

Algorithm 1 Initializng the Dynamic Load Balancing algorithm

For each processor P_i:
- {**Every C cycles**}
- **for** each LP j which P_i hosts **do**
  - $PComp_i = PComp_i + LpLoad_j$
  - $PComm_i = PComm_i + LpComm_j[]$
- **end for**
- $PLoad_i = \lambda * PComp_i + (1 - \lambda)PComm_i$
- All-to-ALL-Broadcast($PLoad_i, PPComm_i[]$)
- {**Upon receiving of $PLoad_j$ from processor P_j**}
- Update the load information of P_j

B. The Dynamic Load Balancing Algorithm

Each processor utilizes the Pload and PPComm information of the other processors and its local PPComm and Pload for the load balancing algorithm. In the first step, the top $P\%$ (a user input parameter) of the over-loaded and under-loaded processors are selected and are placed in the over-loaded and under-loaded sets. The processors of these two sets constitute a bipartite graph in which the edges of the graph are the values of PPComm[]. In the next step, each processor in the over-loaded set forwards LPs to each processor in the under-loaded set. Each processor can forward up to L LPs to other processors, where L is a user input parameter. The number of LPs that a processor P_i in the over-loaded set sends to each processor P_j in the under-loaded set, $LPTransfer_{i,j}$, is:

$$LPTransfer_{i,j} = L * (PPComm_i[j])/(\sum_{i=1}^{N} PPComm_i[j]) \quad (2)$$

Assume that there are 4 processors in the under-loaded set and the PPComm[] values for a processor in the over-loaded set are 15, 25, 35 and 50. Then the number of LPs that P_i forwards to each of these processors is $0.12L$, $0.2L$, $0.28L$ and $0.4L$, respectively. The LPs which have the most communication with the destination processor are chosen for this transfer. LpComm[] is consulted for this information.

It is possible for a sending processor to receive messages intended for LPs which were already transferred. In this case the sending processor forwards these messages to the LP's destination processor. Algorithm 2 summarizes the load balancing algorithm.

Algorithm 2 The load balancing algorithm

For Each Processor P_i:
- {**Every C cycles**}
- **while** number of elements in $O < P\%$ **do**
  - $maxLoad = j$, where P_j has the $Max\{PLoad\}$ and P_j is not in O
  - $O = O \cup P_{maxLoad}$
- **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**
- **for** For each processor P_i in O **do**
  - $LPTransfer_{i,j} = L * (PPComm_i[j])/(\sum_{i=1}^{N} PPComm_i[j])$
  - Find the top $LPTransfer_{i,j}$ LPs which have the maximum value of $LpComm[j]$
  - Send the LPs to the Destination processor
- **end for**

IV. The Genetic Algorithm

It is clear that the tuning parameters of the dynamic load balancing algorithm have a significant effect on their performance. Determining values for these parameters may be viewed as an optimization problem. Because genetic algorithms have had great deal of success in solving a number of optimization problems [15], [16], [24], we develop one in order to determine values for these parameters.

A genetic algorithm is an evolutionary search algorithm in which generations of candidate solutions are iteratively computed [13]. Each generation is evaluated for its quality and a subset of solutions with the best quality is chosen as input for the next round of the algorithm. This is done until a search criterion is reached. The motivation for these algorithms can be found by observing a population of individuals competing for limited resources - only the fittest survive. A generic outline for a genetic algorithm is as follows:

1) Initialize the population with a random set of candidates.
2) Evaluate each candidate in the set to find its fitness.
3) Select parents and put them in a mating pool.
4) Recombine parents to produce new children.
5) Mutate the children.

6) Evaluate the children and find their fitness.
7) Select individuals from the set of parents and generate children to form the new generation.
8) If the termination condition has not satisfied, all steps from step 3 are executed again.

A. Fitness Function

In a genetic algorithm, the fitness of candidate solutions is determined by the objective function. In this section, we define our fitness function in terms of the load imbalance between processors and the event commit rate.

1) Event Commit Rate: Since the main reason of a the parallel simulation is decreasing the simulation time, the elapsed wall clock time of the simulation plays a central role in defining our function. If t_i is the elapsed wall clock time for the ith cycle, C_i, we define the event commit rate (ECR) of the ith interval ,from t_{i-1} to t_i, to be:

$$ECR_i = NC_i/(t_i - t_{i-1}), \tag{3}$$

where NC_i denotes the number of committed (i.e. non rolled-back) events in the i-th interval. With this definition, if $ECR_i < ECR_j$ represents the event commit rates for the ith and jth cycles, the simulation was faster in jth cycle than in the ith cycle.

2) Average Load Imbalance: As already mentioned, the distribution of load between different processors of the system has a significant effect on the performance of the simulation. If we have a load-imbalance we cannot benefit from the parallelism which exists within the system (some processors are over-loaded while some are idle). Communication load imbalance may result in a communication link becoming a bottleneck in the simulation too. Hence, both computation and communication loads are considered by our algorithm. The average computational load imbalance , $AvgComp$, is defined by

$$AveComp = (\sum_{j=1}^{N}\sum_{i=1}^{j}(i \neq j)(PComp_i - PComp_j)/C(2,n) \tag{4}$$

Similarly, the average communication load imbalance, AveComm, is defined as:

$$AveComm = (\sum_{i=1}^{N}\sum_{j=1}^{i}(i \neq j)PPComm_i[j])/C(2,n) \tag{5}$$

Finally, we define the average total load-imbalance, ATL, as the weighted sum of these two parameters.

$$ATL = \lambda * AveComp + (1-\lambda) * AveComm \tag{6}$$

where λ is a user defined parameter between 0 and 1.

3) Fitness Function: In our genetic algorithm, the fitness value of each gene, F_i, is defined by both the event commit rate and the load imbalance as follows:

$$F_i = ECR * 1/ATL \tag{7}$$

A larger event commit rate means that the percentage of rolled back events is decreased, leading to a better simulation time. Decreasing the average load imbalance of the system also improves the simulation time.

B. Tuning Parameters

Our Time Warp simulator makes use of a bounded window in order to decrease the number of rollbacks. The window size plays a critical role in the performance of our simulator. In addition to tuning the parameters which affect the performance of the dynamic load balancing algorithm, our algorithm determines the window size. In this section we discuss these parameters and the time window.

1) Load Balancing Parameters: The two parameters which determine the total amount of load transferred by the algorithm are P and L. $2P$ is the percentage of processors which participate in the load balancing algorithm ($P\%$ in over-loaded set and $P\%$ in under-loaded set). Our experiments indicated that for our experimental platform (32 dual core, 64 bit Intel processors on a fast Ethernet), we could not set P to values more than 40% because of the load-transferring overhead. Values less than 30% reduced the impact of dynamic load balancing algorithm. Hence, we set P=40% i.e. 24 processors participate in the load balancing algorithm.

The value of L represents the number of LPs that each processor forwards to other processors. If we set L to a large value it is probable that the communication cost of transferring these LPs may significantly decrease the speed-up obtained as a consequence of transferring the LPs. On the other hand, if L has a small value, the load balancing may not be effective.

The other parameter which has a significant effect on the performance of dynamic load balancing algorithm is C, the frequency of executing the algorithm. If we run the algorithm too frequently, the speed-up may be adversely affected. Our experiments indicate that for different circuits and different numbers of processors, we need to modify the values of both C and L. As a result, we embed both of these parameters in our candidates for the GA.

2) Window: Our experiments indicate that the value of W plays an important role. Increasing W worsens the simulation time because of increased rollback activity resulting from less restrained optimism. On the other hand, setting W to a small value (e.g. 1) prevents the simulation from benefitting from the parallelism in the simulation. As a result, we use the genetic algorithm to tune the size of the time window as well.

V. Implementation

A. Encoding Mechanism

Each candidate solution in a genetic algorithm is represented by a chromosome which is made up of genes. Various candidate representations have been introduced in the literature

[32], including binary, integer, floating-point phenotypic and tree representations. In this paper, we utilize a binary representation. Each candidate is represented by a string which is composed of three parameters: L, C and W. The maximum values of these parameters are 512, 16 and 16. The primary reason for this choice of values is that larger values resulted in poor performance. As a result, the string length is seventeen bits. For example, if L, C and W have the values of 123, 12 and 14 respectively, the encoded representation for the candidate is 111101111001110.

B. Candidate Initialization and Evaluation

We start the simulation with 8 candidates. While it is possible to select these candidates randomly, we chose the initial values of the parameters according to our experience-for various benchmark circuits, we utilize different initialization sets.

The genetic algorithm is run in a central processor which is used to compute the GVT as well. There is a master processor which determines the GVT value and which also runs the genetic algorithm. Each processor broadcasts its communication and computation load values to all of the other processors. The central processor uses these values to calculate the fitness value of each candidate. In order to evaluate the effect of the current value of C, the algorithm is run five times. Since the dynamic load balancing algorithm is run every C cycles, we need $5C$ cycles to determine the fitness value of the first candidates.

C. Parent Selection, Crossover and Mutation

After choosing an initial set of candidates, a subset of these candidates is chosen and utilized as the parents of the next generation of solutions. They are put into a "mating pool".

Various parent selection algorithms have been introduced [11]. We make use of the Stochastic Universal Sampling (SUS) [11] algorithm. In SUS, the fitness of each candidate is evaluated using formula 7. With this value, the probability of selecting a candidate as a parent can be calculated by:

$$P_i = f_i / \sum_{j=1}^{U} f_j \tag{8}$$

where U is the population size. With these values of P_i, an array a with Ucandidates can be constructed such that $a[j] = \sum_{i=1}^{j} P_i$, $1 \leq j \leq U$. Each cell (slot) of the array is constructed by adding the value of current element to the previous elements, i.e. it is the cumulative probability. Table 1 shows the slot values for 8 random initial candidates.

Starting with $i = 0$, a random number, rand, between 0 and $1/U$ is generated. U is the number of parents. If $rand <= a[i]$ and $rand > a[i-1]$ then candidate i is selected. After each candidate selection the values of rand and i are updated to $rand + 1/U$ and $i + 1$ respectively. This process is continued until U candidates are selected. Algorithm 4 contains the details of the SUS.

After selecting the parents, we recombine them in order to produce new children by means of an operation known as a crossover. As we have three parameters within each chromosome, we utilize the so-called three section cross-over. In this approach, one cross over point is selected within each section (L, C and W) and the cross over is performed at each section. Figure 2 illustrates our crossover approach. The three cross over points are selected by choosing random numbers between zero and the length of each section.

TABLE I
FITNESS VALUES, PROBABILITIES AND SLOT VALUES OF CANDIDATES

Candidate	Fitness Value	Probability	Slot Value
1	0.3321	0.1670	0.1670
2	0.2213	0.1113	0.2783
3	0.1156	0.0581	0.3364
4	0.4987	0.2508	0.5872
5	0.1137	0.0571	0.6443
6	0.2087	0.1049	0.7492
7	0.067	0.0337	0.7829
8	0.431	0.2167	1

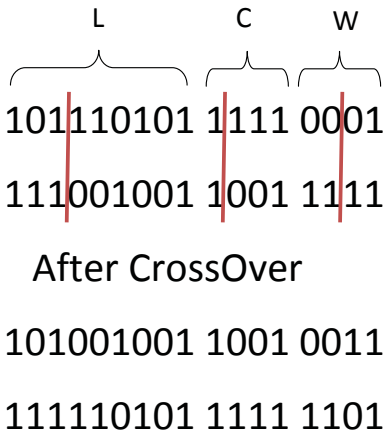


Fig. 1. The crossover operation

The last step in producing children is mutating the genes. In order to do so, we select a mutation rate, R_t, and produce a random number for each gene in the chromosome. If the random number is smaller than R_t, we do not do the mutation, otherwise we mutate the corresponding gene. The main goal of the mutation process is to increase the diversity of population. The purpose of mutation is to prevent the algorithm from being trapped in a local minimum.

D. Genetic Algorithm and Dynamic Load Balancing

In the first step of the genetic algorithm, the population is initialized with a random set of candidates. Afterwards, for each candidate, the parameters (W, L, C) of the candidate are broadcast to all the processors. Each processor runs the distributed load balancing algorithm in parallel with the other processors and forwards LPs to other processors to balance the computational and communication loads for 5C cycles. The frequency of running the load balancing algorithm, C, is one of the parameters that we tune on the fly. In order to see the effect of the current value of C, we run the load balancing algorithm with frequency C for a total of 5 times. At this point, each processor sends its load information to

Algorithm 3 Stochastic Universal Sampling

```
for each candidate j in the solution set do
  a[j] = ∑ p_i, 1 ≤ j ≤ U
end for
Generate a random number rand in the range [0, 1/U]
n = 0
i = 1
while n < U do
  while rand ≤ a[i] and n < U do
    Add candidate i to the mating pool
    rand=rand+1/U
    n=n+1
  end while
  i=i+1
end while
```

the central processor, which computes the fitness value of the candidate. This process is continued until all of the fitness values of the initial set are calculated. At this point, the genetic algorithm proceeds to generate children by parent selection, parent recombination and mutation. The fitness values of the new children are calculated as described above. Finally a set of surviving candidates are chosen. If the termination condition is not satisfied, the computation continues.

E. Survivor Selection

After the mutation step, we find the fitness of all of the new candidates. In order to do so, we run the load balancing algorithm with the corresponding values of each candidate. The size of the time window is also set to the window size of the current candidate.

After finding the fitness value of all the candidates in the new solution set, we have a pool of parents and children from which to choose the survivors. Hence we need a survivor selection algorithm. Various algorithms have been introduced, among which are age-based algorithms and fitness-based algorithms. In an age-based algorithm, a candidate exists in the solution set for a fixed number of iterations. This means that if we set the age to one, the entire population is replaced by children. On the other hand, in a fitness-base algorithm candidates with better fitness values are chosen. In this paper, we make use of a combined version in which we choose the top m candidates, according to their fitness values and then take into account the age of the these candidates. If their age is greater than a pre-determined value, we remove them from the set.

Eventually the GA algorithm must terminate. Some approaches for deciding on when it terminates are:

- The number of generations reaches a pre-determined limit.
- The number of fitness evaluations reaches a pre-set limit.
- The population converges to a single candidate.
- The best fitness value stays unchanged for a pre-determined number of generations.

We stop the genetic algorithm whenever one of these conditions is satisfied.

VI. Performance of the Genetic Algorithm

In order to evaluate the performance of the load balancing and genetic algorithms, we make use of VXTW [22] as our simulation environment. Our simulation platform has 32 dual core 64 bit Intel processors. Each processor has 8 GBytes of internal memory. We utilized Message Passing Interface (MPI) [26] for communication between the processors. Each simulation point in the graphs is the average of 10 simulation runs. We assume that the gate delay is one unit for all of the gates and that the wire transmission time is zero. Initially, we distribute the gates between the processors using a depth first search (DFS) algorithm with load balancing constraints.

The Verilog source files which we use in this simulation are the OpenSPARC T2, the LEON processor and the RPI circuit. The openSPARC T2 is an open source processor which was designed and released by Sun in 2007. We use part of the OpenSPARC T2 and synthesized it using Synopsis DC. It has 200k gates. The LEON processor is a 32-bit microprocessor which is based on the SPARC-V8 RISC architecture and instruction set. It was designed by the European Space Research and Technology Centre, and by Gaisler Research [31]. 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. The RPI circuit is a Viterbi decoder with 800k gates from Rensselaer Polytechnic Institute (RPI).

Recall that L represents the number of LPs which we transfer in each cycle of the load balancing algorithm, C is the frequency of running it, and λ represents the weight parameter used in computing the total load of a processor. We set λ to 0.6. The initial population size was set to 8. In order to determine the fitness of each candidate, we ran the simulation 5 times with values of that candidate. Each candidate is composed of three parameters itself, C, L and W. The values of C and L are numbers between 1 and 16 and the value of L is a number between 1 and 256. The reason that all of the max values are powers of 2 is that we use a bitwise crossover.

Two load balancing algorithms for parallel circuit simulation were described in [23], [21]. The first algorithm utilized a centralized dynamic load balancing approach while the second arranged processors in a virtual ring and used a token based algorithm to balance the load. In the centralized DLB algorithm, we employed both a computational and a communication load balancing algorithm. Tables II and III compare the distributed load balancing algorithm with the other two algorithms for the large and small RPI circuits; they display the improvement of the simulation time compared to a static Time Warp simulation. The simulation time for the centralized DLB is the best time of the computational and communication based algorithms. The distributed load balancing algorithm resulted in the greatest improvement for all parameter values. The results utilize the following parameter values- C=10, L=200 and W=5.

There are two main reasons for the better performance of the distributed load balancing algorithm. 1- it combines the computational and communication load balancing algorithms in the centralized DLB, so it can balance both the computation

and communication loads at the same time. 2- each processor in the over-loaded set is matched with all of the processors in the under-loaded set and forwards parts of its load to each of them.

TABLE II
PERCENT IMPROVEMENT IN SIMULATION TIME OVER TIME WARP (LARGE RPI CIRCUIT)

Number of Processors	Centralized DLB	Token DLB	Distributed DLB
8	13.2	21.5	30.6
12	14.1	22.7	31.2
20	17.9	24.9	29.3
31	15.2	21.5	28.4

TABLE III
PERCENT IMPROVEMENT IN SIMULATION TIME OVER TIME WARP (SMALL RPI CIRCUIT)

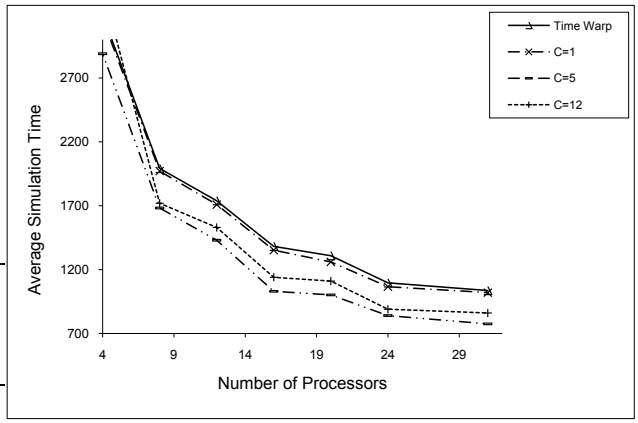
Number of Processors	Centralized DLB	Token DLB	Distributed DLB
8	11.1	21.2	29.2
12	12.1	20.3	30.1
20	9.2	18.7	28.2
31	10.5	19.1	31.2

Our experiments underlined the importance of tuning the parameters. Figures 2-a and 2-b show our results for different values of C for the large RPI circuit. $L = 150$ in figure 2-a and $L = 200$ in figure 2-b. As can be seen, the intermediate value of $C = 5$ yielded the best results for both values of L. The simulation time was improved by 25% and 30% for L= 150 and 200 respectively. The same results are achieved by changing the values of L and W.

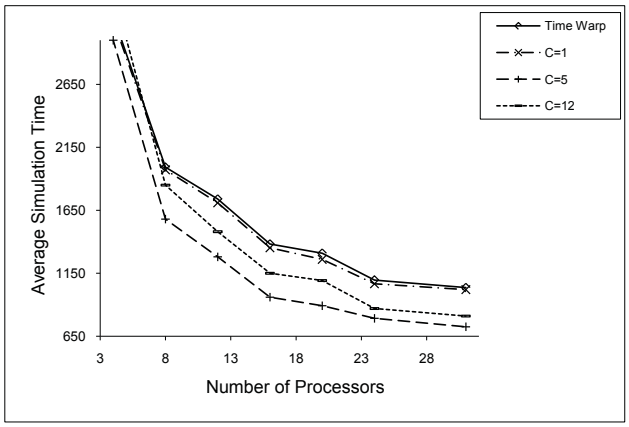
Figures 3 and 4 show the speed up for (1)the static Time Warp algorithm (2) the dynamic load balancing algorithm with different parameter values (3) the genetic algorithm on the large RPI circuit, the small RPI circuit, the OpenSparc T2 and the LEON processors respectively. The algorithm improves simulation times in comparision with static simulation up to 55%, 70%, 55%, and 68% for the OpenSparc T2, large RPI, small RPI, and LEON circuits respectively. Compared to a sequential simulation, total simulation times are decreased by 93%, 94%, 91.5%, and 92% for OpenSparc T2, large RPI, small RPI, and LEON circuits respectively.

The GVT computation in VXTW is done by a central processor. The genetic algorithm was implemented within the same central processor. This way, the genetic algorithm is run while the other processors are running the gate level simulation or the load balancing algorithm. In order to compute the fitness value of each candidate, the genetic algorithm needs to gather load information from all of the processors. This load information is piggy-backed to the central processor on GVT messages. We noted that implementing the load balancing algorithms in each processor can use up to 40% of total computation time of each processor. This happens when we have large values for C and L. Applying the genetic algorithm the running time of the load balancing algorithm can be reduced up to 10% for some circuits

The reasons that the genetic algorithm was implemented on-line and not off-line are (1)it would be necessary to somehow categorize similar circuits in order to make use of the same



(a)



(b)

Fig. 2. Simulation time using load balancing for the large RPI circuit for L: a)L =150, b)L =200

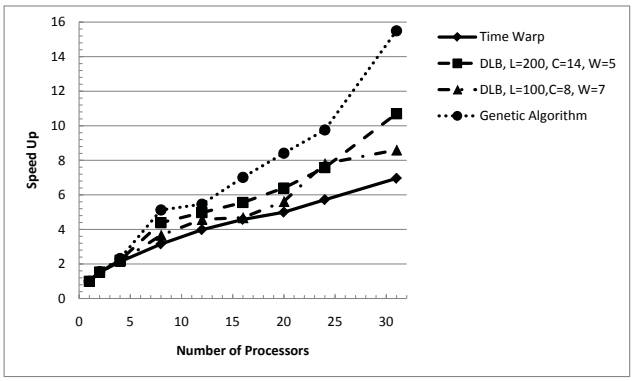
parameters for different circuits. It is also not at all clear how to define these categories (2) The on-line version is not costly to run.

We also measured the standard deviation for the genetic algorithm and static Time Warp algorithms. In all of the cases, the standard deviation of genetic algorithm is better than that of the Time Warp algorithm-we get a more stable result with the genetic algorithm. The reason for this is that the result of Time Warp algorithm depends on the quality of initial candidate solutions; the genetic algorithm can find a good solution by searching a large choice of possibilities. Figure 5 depicts the results of the standard deviation.

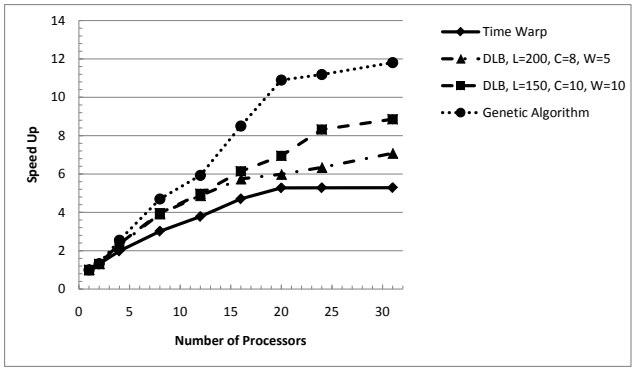
VII. CONCLUSION

One of the main approaches utilized for logic verification is discrete event simulation. Current circuits have millions of gates and simulation on a single processor has become a bottleneck for the design of these circuits.

In order to improve the performance of the simulator, a dynamic load balancing algorithm was introduced. This algorithm is a distributed algorithm in which each processor gathers the load information of the other processors. The processors which participate in the load balancing algorithm

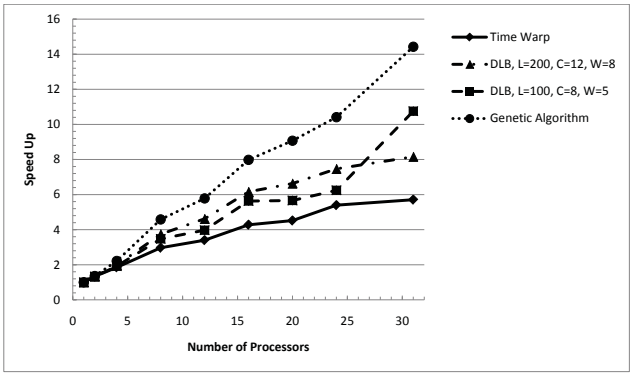


(a)

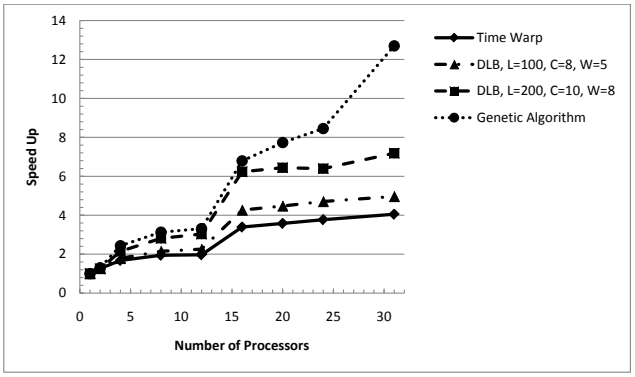


(b)

Fig. 3. Speed-up utilizing the genetic algorithm a)OpenSparc T2 processor b)Small RPI



(a)



(b)

Fig. 4. Speed-up utilizing the genetic algorithm a)Large RPI b)LEON

are separated into two sets of over-loaded and under-loaded processors. The over loaded processors select some LPs (gates) and forward them to the under-loaded processors. The parameters of the load balancing algorithm are L and C, which represent the number of LPs which are transferred in each run of the dynamic load balancing algorithm and the frequency of running the load balancing algorithm respectively.

We also make use of a time window in Time Warp to control the optimism so that we can avoid excessive rollbacks. The size of the window, W plays an important role in its success.

In order to find optimal values for L, C and W, we made use of an online genetic algorithm. Each candidate solution in the genetic algorithm consists of L, C and W. Using this algorithm, we improved the simulation time in comparision to static simulation up to 55%, 70%, 55%, and 68% for OpenSparc T2, large RPI, small RPI, and LEON circuits respectively. Compared with sequential simulation, the simulation time was decreased up to 93%, 94%, 91.5%, and 92% for OpenSparc T2, large RPI, small RPI, and LEON circuits respectively.

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] F. Bonomi and A. Kumar. Adaptive optimal load balancing in a nonhomogeneous multiserver system with a central job scheduler. *IEEE Trans. Comput.*, 39:1232–1250, October 1990.

[7] C. Burdorf and J. Marti. Load balancing strategies for time warp on multi-user workstations. *The Computer Journal*, 36:108–176, July 1993.

[8] Debapriya Chatterjee, Andrew DeOrio, and Valeria Bertacco. Event-driven gate-level simulation with gp-gpus. In *Proceedings of the 46th Annual Design Automation Conference*, DAC '09, pages 557–562, New York, NY, USA, 2009. ACM.

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

[10] 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.

[11] Agoston E. Eiben and J. E. Smith. *Introduction to Evolutionary Computing*. SpringerVerlag, 2003.

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

[13] David E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1989.

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

[15] Yosuke Kimura and Kenichi Ida. Improved genetic algorithm for vlsi floorplan design with non-slicing structure. *Comput. Ind. Eng.*, 50:528–540, August 2006.

[16] Shigenobu Kobayashi, Isao Ono, and Masayuki Yamamura. An efficient genetic algorithm for job shop scheduling problems. In *Proceedings of the 6th International Conference on Genetic Algorithms*, pages 506–511, San Francisco, CA, USA, 1995. Morgan Kaufmann Publishers Inc.

[17] Venkatram Krishnaswamy and Prithviraj Banerjee. Actor based parallel vhdl simulation using time warp. *SIGSIM Simul. Dig.*, 26:135–142, July 1996.

[18] 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.

[19] Dragos Lungeanu and C.-J. Richard Shi. Parallel and distributed vhdl simulation. In *Proceedings of the conference on Design, automation and test in Europe*, DATE '00, pages 658–662, New York, NY, USA, 2000. ACM.

[20] Dale E. Martin, Radharamanan Radhakrishnan, Dhananjai M. Rao, Malolan Chetlur, Krishnan Subramani, and Philip A. Wilsey. Analysis and simulation of mixed-technology vlsi systems. *J. Parallel Distrib. Comput.*, 62:468–493, March 2002.

[21] Sina Meraji and Carl Tropper. Towards optimizing parallel digital logic simulation. *Technical Report School of Computer Science, McGill University*, 2010.

[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] Sina Meraji, Wei Zhang, and Carl Tropper. On the scalability and dynamic load-balancing of optimistic gate level simulation. *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, 29:1368–1380, September 2010.

[24] Zbigniew Michalewicz. *Genetic algorithms + data structures = evolution programs (3rd ed.)*. Springer-Verlag, London, UK, 1996.

[25] Jayadev Misra. Distributed discrete-event simulation. *ACM Comput. Surv.*, 18(1):39–65, 1986.

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

[27] David M. Nicol and Paul F. Reynolds. A statistical approach to dynamic partitioning. In *Proceedings of Parallel and Discrete Event Simulation, PADS85*, pages 53–56, New York, NY, USA, 1985. ACM.

[28] A. Palaniswamy and P.A. Wilsey. Adaptive bounded time windows in an optimistically synchronized simulator. In *Proc. 3rd Great Lakes Symp. on VLSI*, pages 114–118, 1993.

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

[30] Kiran S. Panesar and Richard M. Fujimoto. Adaptive flow control in time warp. In *Proceedings of the eleventh workshop on Parallel and distributed simulation*, PADS '97, pages 108–115, Washington, DC, USA, 1997. IEEE Computer Society.

[31] LEON Processor. *Open Source Processor*. http://www.gaisler.com/cms/.

[32] Franz Rothlauf and David E. Goldberg. *Representations for Genetic and Evolutionary Algorithms*. Physica-Verlag, 2002.

[33] 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.

[34] D. Briscoe Sokol, L. and A. Wieland. Reinforcement learning: A survey. *MTW: a strategy fo scheduling discrete simulation events for concurrent execution*, 19:34–42, 1996.

[35] Lisa M. Sokol, Jon B. Weissman, and Paula A. Mutchler. Mtw: an empirical performance study. In *Proceedings of the 23rd conference on Winter simulation*, WSC '91, pages 557–563, Washington, DC, USA, 1991. IEEE Computer Society.

[36] 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.

[37] 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.

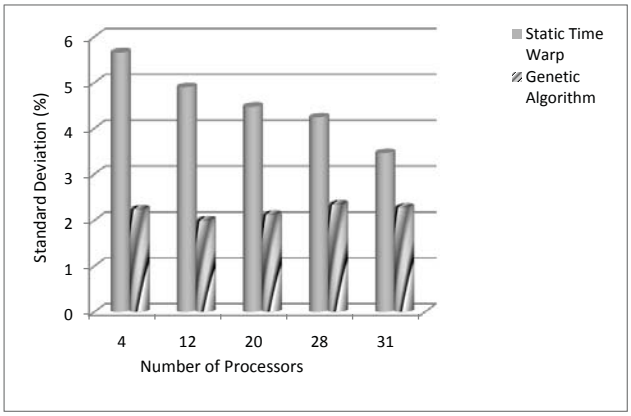
[38] Jun Wang and Carl Tropper. Optimizing time warp simulation with reinforcement learning techniques. In *WSC '07: Proceedings of the 39th conference on Winter simulation*, pages 577–584, Piscataway, NJ, USA, 2007. IEEE Press.

[39] Subramani Wilsey, Martin. Savant/tyvis/warped components for the analysis of vhdl. In *Proc. INt. Verilog HDL Cof. and Proc. VHDL User's Forum*, 1998.

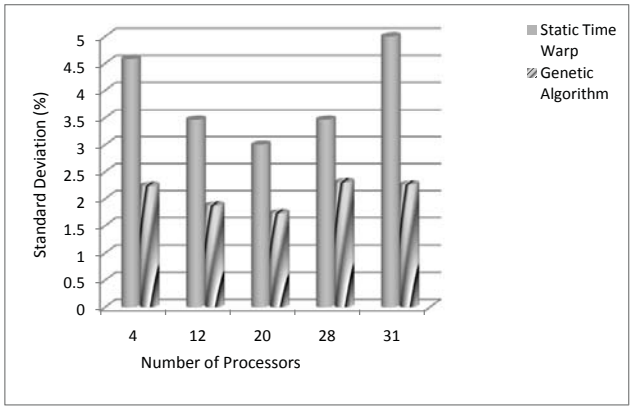
[40] Qing XU and Carl Tropper. Xtw, a parallel and distributed logic simulator. In *ASP-DAC '05: Proceedings of the 2005 conference on Asia South Pacific design automation*, pages 1064–1069, New York, NY, USA, 2005. ACM.

[41] 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.

[42] 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)

Fig. 5. The standard deviation of the dynamic load balancing and genetic algorithm a)Small RPI b)OpenSparc