Towards Optimization Techniques for Dynamic Load Balancing of Parallel Gate Level Simulation

Sina Meraji

Doctor of Philosophy

School of Computer Science

McGill University
Montreal, Quebec
2011-06-01

A thesis submitted to McGill University in partial fulfillment of the requirements for the degree of Doctor of Philosophy

Copyright (c) 2011 Sina Meraji
DEDICATION

To My Dear Parents Roya Meshkat and Mahmood Meraji
ACKNOWLEDGEMENTS

I would like to thank you my thesis advisor, Prof. Carl Tropper for his invaluable guidance during this research. Carl was more than an excellent supervisor, his care, moral support and encouragement were always very much appreciated. His financial assistance also made everything possible.

I would also like to thank you my parents, my brother and my sister for all their love and support over the years.

Special thanks to my committee members, Prof. Hans Vangheluwe, Prof. Doina Precup and Prof. Bettina Kemme who helped me a lot during the course of this thesis.

Thanks also to my colleagues Wei Zhang, David Xu and Jun Wang for all the intelligent discussions we had and also for their encouragement and support. They made it to be a lot of fun to work in Parallel and Distributed Simulation lab.

Thanks also to all the SOCS system staff and administrative who help me a lot during all these years: Diti Anastasopoulos, Andrew Bogecho, Ron Simpson, Kailesh Mussai, Tricia Bernier, Vanessa Bettencourt-Laberge, Heather Holowathy, Sheryl Morrissey and Ann Jack.

I would also like to thank all the great people I have met at McGill. They have made these years very enjoyable and unforgettable. Special thoughts go to: Amir Naghd, Arash Nouri, Mathieu Petitpas, Amin Ranjbar, Kamal Zellang, Yasi Sedaghat, Ladan Mahabadi and Shahin Rabbani.
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. Consequently, parallel simulation has emerged as an approach which can be both fast and cost effective. In this thesis, we examine the performance of a parallel Verilog simulator, VXTW, on four large, real designs using an optimistic synchronization scheme named Time Warp. As previous work has made use of either relatively small benchmarks or synthetic circuits, the use of these circuits is far more realistic. Because of the low computational granularity of a gate level simulation and because the computational and communication loads vary throughout the course of the simulation, the performance of Time Warp can be severely degraded or can even be unstable. Dynamic load balancing algorithms for balancing the computational and communication loads during the simulation are described in this thesis. Like all load balancing algorithms, the proposed algorithms have some tuning parameters which must be optimized. In addition, in order to avoid the simulation from being too optimistic, we make use of a time window. In the thesis, we make use of learning techniques from artificial intelligence (N-armed Bandit, Multi-state Q-learning) and heuristic searches (Genetic Algorithm, Simulated Annealing) to tune the parameters of the dynamic load balancing algorithms and to determine the size of the time window. we evaluated the performance of these algorithms on open source Sparc and Leon processor designs and on two Viterbi decoder designs and observed up to a 70% improvement in simulation time using these approaches.
Une des conséquences de la loi de Moore est la croissance significative de la taille des circuits intégrés; il en résulte que la simulation est devenue le goulot d'étranglement majeur dans le processus de conception de tels circuits. Conséquemment, la simulation parallèle se veut une approche qui a le potentiel d'être à la fois rapide et rentable. Dans cette thèse, nous examinerons la performance d'un simulateur Verilog parallèle appelé VXTW sur quatre conceptions de processeurs réelles de grande taille, en utilisant un algorithme de synchronisation optimiste appelé Time Warp. Puisque les travaux précédents ont utilisé des circuits synthétiques ou des tests de performance de taille relativement petite, l'utilisation de ces circuits est beaucoup plus réaliste. Puisque les simulations au niveau des portes logiques impliquent une granularité calculatoire peu élevée, et puisque les charges calculatoires et de communication varient au cours de la simulation, la performance de Time Warp peut se dégrader sévèrement ou devenir instable. Dans cette thèse, nous décrivons des algorithmes dynamiques d'équilibrage de charge visant à équilibrer les charges calculatoires et de communication durant la simulation. Comme tous les algorithmes d'équilibrage de charge, les algorithmes proposés comportent des paramètres de réglage qui doivent être optimisés. De plus, nous utilisons une fenêtre de temps pour éviter que la simulation ne soit trop optimiste. Dans cette thèse, nous utilisons des techniques d'apprentissage provenant du domaine de l'intelligence artificielle (machine à sous à leviers multiples, Q-learning avec plusieurs agents) et des recherches heuristiques (algorithmes génétiques, méthode du circuit simulé) pour régler les paramètres des algorithmes.
dynamiques d’équilibrage des charges, ainsi que pour déterminer la taille de la fenêtre
de temps. Nous évaluons la performance de ces algorithmes sur des conceptions de
processeurs Sparc et Leon libres de droits, ainsi que sur deux décodeurs Viterbi, et
nous avons pu observer une amélioration du temps de simulation de 70% en utilisant
ces approches.
# TABLE OF CONTENTS

DEDICATION ................................... ii
ACKNOWLEDGEMENTS ............................. iii
ABSTRACT ..................................... iv
ABRÉGÉ ....................................... v
LIST OF TABLES ................................. x
LIST OF FIGURES ................................. xi

1 Introduction ................................... 1
   1.1 Introduction ................................... 1
   1.2 Discrete Event Simulation ...................... 3
      1.2.1 Time-Stepped Simulation .................... 4
      1.2.2 Event-Driven Simulation .................... 5
   1.3 Digital Logic Simulation ....................... 6
   1.4 Parallel Discrete Event Simulation .......... 7
      1.4.1 Conservative Synchronization ............... 9
      1.4.2 Optimistic Synchronization ................. 14
      1.4.3 Parallel VHDL ............................. 19
   1.5 Time Warp for Digital Logic Simulation ...... 20
      1.5.1 Clustered Time Warp ....................... 20
      1.5.2 DVS ...................................... 22
   1.6 Dynamic Load Balancing ....................... 24
   1.7 Time Window .................................. 26
   1.8 Thesis Motivations and Structure ............ 27

2 Verilog XTW ................................... 30
   2.1 Introduction ................................... 30
   2.2 XTW ......................................... 30
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.3</td>
<td>Design and Implementation of the Verilog Parser</td>
<td>34</td>
</tr>
<tr>
<td>2.3.1</td>
<td>Synopsis Design Compiler</td>
<td>34</td>
</tr>
<tr>
<td>2.3.2</td>
<td>Verilog Parser</td>
<td>35</td>
</tr>
<tr>
<td>2.3.3</td>
<td>Architecture</td>
<td>35</td>
</tr>
<tr>
<td>2.4</td>
<td>Performance Evaluation of VXTW</td>
<td>37</td>
</tr>
<tr>
<td>3</td>
<td>Dynamic Load Balancing</td>
<td>44</td>
</tr>
<tr>
<td>3.1</td>
<td>Dynamic Load-balancing</td>
<td>44</td>
</tr>
<tr>
<td>3.1.1</td>
<td>General Structure of the Algorithms</td>
<td>45</td>
</tr>
<tr>
<td>3.1.2</td>
<td>Computation Load-balancing Algorithm</td>
<td>46</td>
</tr>
<tr>
<td>3.1.3</td>
<td>Communication Load-balancing Algorithm</td>
<td>47</td>
</tr>
<tr>
<td>3.2</td>
<td>Simulation Result</td>
<td>49</td>
</tr>
<tr>
<td>4</td>
<td>Reinforcement Learning for Optimizing Dynamic Load-balancing</td>
<td>57</td>
</tr>
<tr>
<td>4.1</td>
<td>Introduction</td>
<td>57</td>
</tr>
<tr>
<td>4.2</td>
<td>N-armed Bandit Method</td>
<td>59</td>
</tr>
<tr>
<td>4.2.1</td>
<td>Q-learning</td>
<td>61</td>
</tr>
<tr>
<td>4.3</td>
<td>Reinforcement Learning in Dynamic Load-balancing of Time Warp</td>
<td>62</td>
</tr>
<tr>
<td>4.3.1</td>
<td>Single Agent vs. Multi-agent</td>
<td>62</td>
</tr>
<tr>
<td>4.3.2</td>
<td>Control Parameters</td>
<td>62</td>
</tr>
<tr>
<td>4.4</td>
<td>The N-armed Bandit Method</td>
<td>64</td>
</tr>
<tr>
<td>4.5</td>
<td>Q-learning</td>
<td>66</td>
</tr>
<tr>
<td>4.6</td>
<td>Simulation Result</td>
<td>67</td>
</tr>
<tr>
<td>5</td>
<td>Simulated Annealing for Finding the Size of Time Window</td>
<td>74</td>
</tr>
<tr>
<td>5.1</td>
<td>Introduction</td>
<td>74</td>
</tr>
<tr>
<td>5.2</td>
<td>Simulated Annealing</td>
<td>75</td>
</tr>
<tr>
<td>5.2.1</td>
<td>The Simulated Annealing Algorithm</td>
<td>75</td>
</tr>
<tr>
<td>5.2.2</td>
<td>Simulated Annealing and Optimizing Time Warp</td>
<td>76</td>
</tr>
<tr>
<td>5.3</td>
<td>Experimental Results</td>
<td>78</td>
</tr>
<tr>
<td>6</td>
<td>A Multi-Agent Approach for Optimizing Parallel Digital Logic Simulation</td>
<td>85</td>
</tr>
<tr>
<td>6.1</td>
<td>Dynamic Load Balancing</td>
<td>85</td>
</tr>
<tr>
<td>6.1.1</td>
<td>Initiating Process</td>
<td>85</td>
</tr>
<tr>
<td>6.1.2</td>
<td>Dynamic Load Balancing Algorithm</td>
<td>87</td>
</tr>
<tr>
<td>6.2</td>
<td>Reinforcement Learning and Dynamic Load Balancing of Time Warp</td>
<td>89</td>
</tr>
<tr>
<td>Section</td>
<td>Title</td>
<td>Page</td>
</tr>
<tr>
<td>---------</td>
<td>-------</td>
<td>------</td>
</tr>
<tr>
<td>6.2.1</td>
<td>Multi-State Q-Learning</td>
<td>89</td>
</tr>
<tr>
<td>6.2.2</td>
<td>The Dynamic Load Balancing Agent</td>
<td>91</td>
</tr>
<tr>
<td>6.2.3</td>
<td>The Time Window Agent</td>
<td>95</td>
</tr>
<tr>
<td>6.2.4</td>
<td>Multi-state Q-Learning and the Dynamic Load Balancing of Time Warp</td>
<td>96</td>
</tr>
<tr>
<td>6.3</td>
<td>Simulated Annealing for Time Warp</td>
<td>97</td>
</tr>
<tr>
<td>6.4</td>
<td>Experimental Results</td>
<td>99</td>
</tr>
<tr>
<td>7</td>
<td>A Genetic Algorithm for Parallel Digital Logic Simulation</td>
<td>113</td>
</tr>
<tr>
<td>7.1</td>
<td>The Genetic Algorithm</td>
<td>113</td>
</tr>
<tr>
<td>7.1.1</td>
<td>Fitness Function</td>
<td>114</td>
</tr>
<tr>
<td>7.1.2</td>
<td>Tuning Parameters</td>
<td>116</td>
</tr>
<tr>
<td>7.2</td>
<td>Implementation</td>
<td>118</td>
</tr>
<tr>
<td>7.2.1</td>
<td>Encoding Mechanism</td>
<td>118</td>
</tr>
<tr>
<td>7.2.2</td>
<td>Candidate Initialization and Evaluation</td>
<td>118</td>
</tr>
<tr>
<td>7.2.3</td>
<td>Parent Selection, Crossover and Mutation</td>
<td>119</td>
</tr>
<tr>
<td>7.2.4</td>
<td>Survivor Selection</td>
<td>120</td>
</tr>
<tr>
<td>7.3</td>
<td>Performance of the Genetic Algorithm</td>
<td>123</td>
</tr>
<tr>
<td>8</td>
<td>Conclusion</td>
<td>128</td>
</tr>
<tr>
<td>8.1</td>
<td>Conclusion</td>
<td>128</td>
</tr>
<tr>
<td>8.2</td>
<td>Future Work</td>
<td>131</td>
</tr>
<tr>
<td>References</td>
<td></td>
<td>133</td>
</tr>
</tbody>
</table>
LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6–1</td>
<td>The percentage of improvement in simulation time over Time Warp for dynamic load balancing. Reinforcement Learning (RL) and Simulated Annealing (SA) used for the large RPI circuit</td>
<td>104</td>
</tr>
<tr>
<td>6–2</td>
<td>The percentage of improvement in simulation time over Time Warp for dynamic load balancing. Reinforcement Learning (RL) and Simulated Annealing (SA) used for the LEON processor</td>
<td>105</td>
</tr>
<tr>
<td>7–1</td>
<td>Fitness Values, Probabilities and Slot Values of Candidates</td>
<td>120</td>
</tr>
<tr>
<td>Figure</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>--------</td>
<td>-----------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>1–1</td>
<td>Time-stepped simulation</td>
<td>5</td>
</tr>
<tr>
<td>1–2</td>
<td>Event-driven simulation</td>
<td>6</td>
</tr>
<tr>
<td>1–3</td>
<td>The main structure of an event-driven simulation program</td>
<td>10</td>
</tr>
<tr>
<td>1–4</td>
<td>Local casualty control</td>
<td>10</td>
</tr>
<tr>
<td>1–5</td>
<td>The transient message example</td>
<td>16</td>
</tr>
<tr>
<td>1–6</td>
<td>A synchronous GVT computation algorithm</td>
<td>17</td>
</tr>
<tr>
<td>1–7</td>
<td>An example of synchronous GVT computation</td>
<td>18</td>
</tr>
<tr>
<td>1–8</td>
<td>The simultaneous reporting problem</td>
<td>19</td>
</tr>
<tr>
<td>1–9</td>
<td>DVS architecture</td>
<td>24</td>
</tr>
<tr>
<td>2–1</td>
<td>An example of Verilog structural and behavioural code</td>
<td>31</td>
</tr>
<tr>
<td>2–2</td>
<td>The main structure of the simulator</td>
<td>36</td>
</tr>
<tr>
<td>2–3</td>
<td>Speedup vs. the number of processors (a) 10 input vectors, (b) 50 input vectors</td>
<td>39</td>
</tr>
<tr>
<td>2–4</td>
<td>Total number of processed events per processor (a) 10 input vectors, (b) 50 input vectors</td>
<td>41</td>
</tr>
<tr>
<td>2–5</td>
<td>Commit rate vs. different number of processors (a) 10 input vectors, (b) 50 input vectors</td>
<td>43</td>
</tr>
<tr>
<td>3–1</td>
<td>The average simulation time of the computation load balancing algorithm on large RPI circuit for different values of $L$ and $P$: a) $P = 20%$, b) $P = 10%$</td>
<td>53</td>
</tr>
</tbody>
</table>
3–2 The average simulation time of the computation load balancing algorithm on OpenSparc T2 for different values of $L$ and $P$: 
a) $P = 20\%$, b) $P = 10\%$ .......................... 54

3–3 The average simulation time of the communication load balancing algorithm on small RPI for different values of $L$ and $P$: a) $P = 20\%$, b) $P = 10\%$ .......................... 55

3–4 The average simulation time of communication load-balancing algorithm on Leon for different values of $L$ and $P$: a) $P = 20\%$, b) $P = 10\%$ .......................... 56

4–1 The average simulation time of computation, communication and N-armed Bandit load-balancing algorithms a) large RPI circuit, b) small RPI circuit .......................... 70

4–2 The average simulation time of computation, communication and N-armed Bandit load-balancing algorithms a) OpenSparc T2 and b) LEON .......................... 71

4–3 The average simulation time of computation, communication and Q-Learning load-balancing algorithms a) large RPI circuit, b) small RPI circuit .......................... 72

4–4 The average simulation time of computation, communication and Q-Learning load-balancing algorithms a) OpenSparc T2 and b) LEON .......................... 73

5–1 The effect of distance parameter on simulation time .................. 80

5–2 The effect of distance parameter on acceptance rate .................. 81

5–3 Simulation results of circuit C38417 .......................... 82

5–4 Simulation results of circuit C38584 .......................... 82

5–5 Standard variance of circuit C38417 .......................... 83

5–6 Standard variance of circuit C38548 .......................... 84

6–1 The states and transitions from the first state in Multi-state Q-learning algorithm .......................... 95

6–2 The average simulation time of the load balancing algorithm for different values of $L$: a) $L = 50$, b) $L = 200$ .......................... 107
6–3 The average simulation time of the load balancing algorithm for different values of $C$: a) $C = 5$, b) $C = 10$ 

6–4 The average simulation time of the load balancing and Multi-agent Q-learning algorithms a) Large RPI and b) Small RPI 

6–5 The average simulation time of the load balancing and Multi-agent Q-learning algorithms a) OpenSparc T2 processor b) LEON Processor 

6–6 The average simulation time of the dynamic load balancing, multi-agent Q-learning and simulated annealing algorithms a) OpenSparc T2 processor b) Small RPI 

6–7 Commit Rate VS. the Number of Processors a) LEON processor b) Small RPI Circuit

7–1 The Crossover Operation

7–2 Average simulation times for different values of $W$

7–3 Speed-up utilizing the genetic algorithm

7–4 Speed-up utilizing the genetic algorithm
CHAPTER 1
Introduction

1.1 Introduction

A computer simulation is a computation that models the behavior of some real or imagined system over time [19]. Computer simulations have become an inseparable part of large system designs. They are used to evaluate the performance of complex systems when analytical solutions and prototyping are too difficult or costly to develop. A classification of simulations is referred as the time flow mechanism, which describes the way to change states when the simulation time is advanced. Continuous and Discrete simulations are the two main types of simulation in this classification. In a discrete simulation, the simulation model only considers the changes of the states at discrete points of simulation time and advances from one point to another. Discrete event simulation is an event driven simulation [19] where the simulation time is advanced from the time stamp of an event to the time stamp of the next event. An important application of computer simulation in engineering is the simulation of VLSI circuits.

Hardware Description Languages (HDL) are used for the design and simulation of current VLSI circuits because they both simplify and speed up the circuit design process. Verilog [46] and VHDL [13] both find widespread use, although Verilog is more prevalent in industrial settings. Simulation is used to verify the correctness of a circuit design. 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 of sequential simulation is becoming prohibitive for these large circuits. Because current digital circuits have millions of gates, it is difficult to fit the simulation models of these circuits into a single processors’s memory. In addition to the demand for memory, the need for decreased simulation time is also a major challenge. In short, 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 [32]. 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). LPs 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 [19], 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 [9] and optimistic synchronization [23]. Conservative simulations rely on processes 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 [23] is the most widely employed.

1.2 Discrete Event Simulation

Simulation may be viewed as the imitation of the dynamic behaviour of a real-
world physical system. In general, simulation depicts the change in a real system
over time. In a computer simulation a model of the system is executed by a program
which executes on a computer. The properties of a system can be represented by
states and the changes within the system can be modeled by changing these state
variables. For example, in a digital circuit, the logic values of the inputs to a logic
gate at a certain instant in time represent the state of that gate at that instant. In a
model that simulates a queue, the number of jobs in the queue represents the state of
the queue. In general, in order to study the dynamics of a real system, a simulation
must include the following elements:

1. A representation of the system’s state.

2. Some means of changing the system’s state.

3. A representation of real time.

The notion of time is essential to a simulation system. Time within a simulation
program (the simulation time) represents the casual relationship between events
in the physical system. The simulation time is often referred to as the logical or
virtual time. Simulation time should not be confused with the wall-clock time, or
the execution time of the simulation. For example, it might take an hour to simulate
hundred events in a system, while it just takes few seconds to simulate the same
number of events in another system.
Changes within the physical system are simulated by updating the values of the state variables. A classification of simulations is referred as the time flow mechanism [19]. According to this classification, there are continuous and discrete simulations. In a continuous simulation the state of the simulation changes continuously over time. Modeling of the weather and voltage changes on wires are examples of continuous simulations. In this thesis, we only deal with discrete simulation models, in which changes in the physical systems can only occur at discrete points in time. This type of simulation is known as discrete Event Simulation (DES). The time advance mechanism [33] within a DES represents the way time advances from one point to the next. Two main time advance mechanisms are: 1) time stepped simulation and 2) event driven simulation.

1.2.1 Time-Stepped Simulation

In a time-stepped simulation, the changes in the states variables are checked and the simulation time advances every $t$ cycles where $t$ is a predefined value. $t$ is referred as the size of the time step. This type of time advance mechanism is also called fixed-increment time advance [33]. When the simulation time advances from $t_i$ to $t_{i+1}$ all of the events within that interval are processed. All of these events are considered to be simultaneous events and can be executed in any order. In order to determine the correct order of these events, we have to reduce the size of the time step. Figure 1-1 depicts a time stepped simulation. The main drawback of this approach is that in order to determine the correct order of the events, we have to set the time step to a small value. We have to update all of the state variables at each advance of simulation time, a costly undertaking.
1.2.2 Event-Driven Simulation

In an event-driven simulation, the simulation time advances according to the time stamps of the events. In this type of the simulation, instead of advancing the simulation time by a fixed value and checking if any event has occurred, the events are sorted according to their time stamp and the simulation time increases from the time stamp of one event to that of another event. As various events have different time stamps, the advancing of simulation time is non uniform in general. Figure 1-2 shows how the event-driven simulation works. In order to implement an event-driven simulation, we need three main data structures.

1. An event list which contains all of the events in increasing time stamp order.

   The event list is usually implemented by the use of a priority queue [1].

2. A global clock which contains the simulation time.

3. A set of state variables which represents the state of the simulation and which models the physical system.

As mentioned, in an event-driven simulation only the processing of events advances the simulation time and changes the state variables. The events are sorted in an event list. The processing of an event might cause in creation of new events as

\[ T_1 \quad e_1 \quad e_2 \quad T_2 \quad ... \quad T_i \quad e_i \quad e_{i+1} \quad T_{i+1} \]

Figure 1–1: Time-stepped simulation
Events in the list are processed until it is either empty or until some termination condition is satisfied.

A common architecture for an event-driven simulation program consists of a simulation application combined with a simulation kernel. The application part is specific to the model, e.g. a digital logic simulation model or a model of airport traffic. State variables and the handlers for the events are implemented within the simulation application. The kernel’s job is to manage simulated time and to call the functions associated with the events (event handlers) at the right times. Figure 1-3 depicts the main structure of an event-driven simulation program. In the following section, we describe event driven simulation for digital logic simulation.

1.3 Digital Logic Simulation

In this section, we briefly describe important aspects of a digital logic simulator. We use graphs to model digital circuits. Gates and flip-flops are represented by nodes and the connections between the gates (wires) can be represented by edges. In a digital circuit, each gate has a delay, which is the latency from the changes in gate’s inputs to changes in the gate’s outputs. A vector of events is assigned to each gate’s input. The events are processed in increasing time stamp order.

Figure 1–2: Event-driven simulation
The processing of an input event may result in the change of the gate’s output which is modeled by means of a new event. The new event advances the simulation time from \( \text{NOW} \) to \( \text{NOW} + t_{\text{input}} \) where \( t_{\text{input}} \) is the event’s time stamp. If the (output) logic value of a gate, \( g \), changes, this change (event) must be propagated to all of the fanouts of \( g \), i.e. to all of the gates which are connected to \( g \)’s output. The new events may change the fanout gates outputs, resulting in the propagation of new events. The time stamp of the new event is \( \text{NOW} + D_g \) where \( D_g \) represents the gate’s delay.

The event handler function is depicted in algorithm 1.

Algorithm 1 Event handler of a simple event-driven circuit simulation

**Input event:**
- Read the next input vector;
- Schedule another input event at time \( \text{NOW} + t_{\text{input}} \)

  **for** each gate \( g \) that is a fanout of a primary input **do**
  - Evaluate \( g \)
    - **if** output of \( g \) changes **then**
      - Schedule another event at \( \text{NOW} + D_g \)
    - **end if**
  **end for**

1.4 Parallel Discrete Event Simulation

Parallel Discrete Event Simulation (PDES) is intended to execute a discrete event simulation on a computer system with more than one processing node. As already mentioned, because of the increase in the complexity of systems which we wish to simulate, we need more powerful computing platforms on which to execute simulations. VLSI circuits have experienced an explosive growth in size. Because of this growth simulation has become a bottleneck in the design process. Hence the
need for parallel discrete event simulation. While a basic goal of PDES is to reduce the simulation time, it is not the only goal. Being able to execute a simulation which cannot execute on a single processor by using an additional processor is of fundamental importance. According to [19], the following are other benefits of PDES:

1. Geographical distribution. PDES makes it possible to geographically distribute computers which perform a specific job.
2. Integrating simulators that execute on machines from different manufactures.
3. Fault tolerance. If one processor breaks down, it may be possible to move the work of the failed processor to other processors.

The components of a PDES include:

1. Logical Process: A logical process is a model of physical component which is to be simulated with the simulation program.
2. PDES Process: A PDES process is a running program that together with other PDES processes performs the simulation of the system. Each PDES process contains one or more LPs.
3. Node: A node is a processor in a multiprocessor system that runs a specific PDES process.

In a PDES system, all the physical components are modeled as LPs. LPs are assigned to different PDES processes and finally PDES processes are distributed among processor nodes. Each of the PDES processes performs part of the overall DES. One of the main differences between the PDES and a single processor DES system is that PDES processes at each node are not isolated from other PDES processes. In a single processor DES, there is one central list of events which are executed in increasing
time stamp order. In a PDES, there is a possibility that a process receives an event which has a time stamp less than the current time stamp of the process. This means that part of the current process has been based on possibly incorrect system state. As mentioned, we called these kinds of errors as causality errors. In order to avoid the occurrence of causality errors we need a synchronization mechanism. Two well-known synchronization mechanisms are conservative and optimistic synchronization. They will be discussed in the following two subsections.

1.4.1 Conservative Synchronization

In conservative synchronization, the Logical Processes (LPs) avoid out of order execution of events. In the following example we assume that event $E_1$ at $LP_A$ with time stamp 3 and event $E_2$ at $LP_B$ has time stamp 10, if $LP_A$ schedules an event $E_3$ in $LP_B$ with time stamp 8, the execution of $E_3$ may affect the execution of $E_2$ at $LP_B$. The local causality constraint avoids the occurrence of such problems.

**Local Causality Constraint:** A local causality constraint in a discrete event simulation with LPs that interact with each other via time stamped messages is satisfied if and only if each LP processes the events in non-decreasing time stamp order [19].

Figure 1-4 illustrates a situation that local causality control is not satisfied. If each LP adheres to the local causality constraint the results obtained from parallel/distributed execution of the simulation program are exactly the same as the results of the sequential execution. In the following, we study three well known conservative synchronization algorithms.
Figure 1–3: The main structure of an event-driven simulation program

Figure 1–4: Local causality control
Null Message Algorithm

Assume that we have N logical processes, \(LP_0, \cdots, LP_{N-1}\). Each LP has a simulation time \(C_i\). Whenever an event is processed, \(C_i\) is updated to the time stamp of that event. If \(LP_i\) can send a message to \(LP_j\), a link is established from \(LP_i\) to \(LP_j\).

The fundamental problem in conservative synchronization is to determine when it is safe to process an event. An event \(E_i\) with time stamp \(T_i\) at \(LP_i\) is safe to process if \(LP_i\) can guarantee that it will not receive an event with a time stamp smaller than \(T_i\). Processes which do not have safe events are blocked and wait until their events become safe. One of the basic algorithms which determines which events are safe to process is the Null message algorithm. In this algorithm it is assumed that events are received in a time stamp order on a link and there is a FIFO queue at each input link. The initial event processing loop for an LP is as follows:

\[
\text{While (simulation is in progress)}
\]

\[
\text{Wait until each input FIFO has at least one message}
\]

\[
\text{Remove the smallest time stamped message } M \text{ from its FIFO}
\]

\[
C = \text{Time stamp of } M
\]

\[
\text{Process}(M)
\]

While this algorithm satisfies the local causality constraint, a cycle of empty queues may lead to a situation where each process on the cycle is blocked and as a result a deadlock can occur. Deadlocks are avoided by the use of lookahead.

Lookahead: if an LP with time stamp \(T\) can schedule events with time stamp of at least \(T+L\) then \(L\) is referred as the lookahead of that LP.
The Null message algorithm avoids the formation of deadlocks as follows. Each \( LP_i \) with time stamp \( T_i \) sends a null message with time stamp \( T_i + L_i \) to each \( LP_j \) which is waiting for a message from \( LP_i \). This null message informs \( LP_j \) that \( LP_i \) will not send a message with a time stamp less than \( T_i + L_i \). As a result some of the messages in \( LP_j \) may be executed and deadlock can be avoided [11]. Null messages are sent after processing of each event. Efforts to reduce the number of null messages are described in [42]. The performance of the algorithm mainly depends on the value of lookahead. Small values of lookahead may result in poor performance of the algorithm.

**Deadlock Detection and Recovery**

The main problem for the Null message algorithm is that a large number of Null messages can be generated, particularly when we have a small lookahead. In order to solve this problem deadlock detection and recovery algorithms [19] can be used. One well known algorithm is based on diffusing computation. In a diffusing computation, the computation is initially blocked and a single controller starts the process. It sends messages to one or more processes. In our simulation, these messages contain events which are safe to process. The processes which receive these events, process them and forward new events to other processes in the system.

The spreading of the events to previously blocked processes can be seen as the construction of a tree. All of the processes in the tree are not blocked; we call them engaged processes. All of the processes outside the tree are blocked—we call them disengaged. Whenever a disengaged process receives an event, it changes its status to engaged and adds itself to the tree. It is also possible that some of the processes
in the tree become blocked and can not process events. If a process is blocked and that process is a leaf node in the tree, it will disengage itself from the tree and inform its parent that it is no longer in the tree. A process becomes a leaf node if all the processes to which it sent messages signaled that they have removed themselves from the tree. If the controller becomes a leaf node then the computation is deadlocked again.

Various algorithms can be employed to break a deadlock. One possibility is for the controller to query the processes in the system asking for their minimum time stamps and then compute the smallest of these time stamps $T$. Having $T$ and knowing the lookahead, $L$, all of the events in $[T, T + L]$ are safe to process and the deadlock is broken. In order to expedite the algorithm, we arrange the processes in a tree structure.

**Bounded Lag Algorithm**

We first define the distance between two processes to be the amount of simulated time which elapses before an event in one process can affect another process [19]. More precisely, if a path exists from $P_1$ to $P_n$ through $P_2, P_3, \ldots, P_{n-1}$, the path length is defined as $L_{12} + L_{23} + \cdots + L_{n-2n-1} + L_{n-1n}$ where $L$ represents the lookahead value. $D_{1n}$, the distance from $P_1$ to $P_n$, can then be defined as the minimum path over all paths from $P_1$ to $P_n$. We can say that an event in $P_m$ depends on an event in $P_n$ if: $T_n + D_{nm} < T_m$ where $T_m$ is the current time of $P_m$. An event $E$ at $P_m$ is safe if it is not possible for an event with a time stamp smaller than $E$ to be sent to $P_m$. Let $LBTS_i$ be the lower bound on the time stamp of any message that $P_i$
can receive in the future. All of the events at $P_i$ which have a time stamp less than $LBTS_i$ are safe to process. $LBTS_i$ can be defined as:

$$LBTS_i = \min(T_j + D_{ji}) / \text{all } J$$

A native use of this definition to construct an algorithm has a fatal flaw—each process has to communicate with the other processes in the system to find the safe messages, resulting in $N^2$ messages. In addition, the distance matrix has to be recomputed if the lookahead changes during the computation.

In order to improve things, a time window can be defined from the simulation time of smallest event in the simulation, $T_S$, to the $T_S + T_W$, where $T_W$ denotes the size of window. Events with a time stamp larger than $T_S + T_W$ are not processed, and as a result, $T_W$ is also called the bounded lag. $LBTS_i$ can be computed as follows

$$LBTS_i = \min(T_j + D_{ji}) / \text{all } J \text{ where } D_{ji} < T_W$$

### 1.4.2 Optimistic Synchronization

In optimistic synchronization of parallel simulation, Logical Processes ($LPs$) process the events optimistically and allow a violation of the local causality constraint while providing mechanisms for recovering from these violations. In this synchronization approach, $LPs$ do not send messages in time stamp order and messages can be delivered in a different order than the one in which they were sent.

Among the optimistic synchronization schemes, Jefferson’s Time Warp [23] is the first and most well known. 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.
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 the input queue. Events in input queue may have been sent to the LP by other LPs or they may have been produced by the LP itself. 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. LPs can be created and assigned to the processors dynamically and the connections between them are not necessarily predefined.

Local control techniques try to retain the state of the system by re-instantiating all of the state variables and by eliminating the effects of messages sent to other LPs. Two approaches for saving the state variables are copy state saving [12], and incremental state saving [51]. Copy state saving is conceptually simpler—it saves all of the 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 [23] is an exact copy of the original message with a flag setting which indicates that it is an anti-message and not a 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 of the processes which were affected by the original message are rolled back [19].

Global Control Mechanism

As the simulation progresses under Time Warp, memory is consumed by saving messages, anti-messages and LP states. As a result, we need a mechanism to minimize the amount of memory used for this storage. By finding a lower bound on the time stamp of future rollbacks, we can delete the memory dedicated to events and states which have a smaller time stamp than this lower bound via fossil collection [65]. This lower bound is called the 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 [19].

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) would be trivial. There are, however, two challenging problems in attempting to create such snapshot-the transient message problem and simultaneous reporting problem.

A transit message is a message which has been sent but not received yet. Transit messages are unprocessed messages which ”disappear” into the network and must be considered in the calculation of GVT. In theory, a controller could halt all of the processors in the system and have them report the minimum of their unprocessed
events and anti-messages. With this information it could compute the GVT. This algorithm is not correct because while all the processors are frozen, there may be some messages in the network. Figure 1-5 illustrates an example of a transient message. The message with time stamp 15 is a transient message whose time stamp is not accounted for by either sender or the receiver processor.

<table>
<thead>
<tr>
<th><strong>Controller</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Periodically:</td>
</tr>
<tr>
<td>- Broadcast a “Start-GVT” message to all the processors to initiate the GVT calculation</td>
</tr>
<tr>
<td>Upon receive “Receive-Start” from all the processors</td>
</tr>
<tr>
<td>- Broadcast a “Compute-GVT” to all the processors</td>
</tr>
<tr>
<td>Upon receive the local minimum from all the processors:</td>
</tr>
<tr>
<td>- Compute the global minimum and broadcasts this value to all the processors</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Processor</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Upon receiving the “Start-GVT” message from the controller:</td>
</tr>
<tr>
<td>- Stop processing the events</td>
</tr>
<tr>
<td>- Send the “Receive-start” message to the controller</td>
</tr>
<tr>
<td>- Wait to receive “Compute-GVT” from the controller</td>
</tr>
<tr>
<td>Upon receiving “Compute-GVT” from the controller:</td>
</tr>
<tr>
<td>- Compute the local minimum time stamp among:</td>
</tr>
<tr>
<td>- The time stamp of unprocessed events and anti-messages within the processor</td>
</tr>
<tr>
<td>- The time stamp of the any message the processor has sent but has not received an acknowledgement</td>
</tr>
<tr>
<td>- Send the minimum value to the controller</td>
</tr>
</tbody>
</table>

Figure 1–6: A synchronous GVT computation algorithm

A straightforward solution to the transient message problem is to use message acknowledgements. A simple synchronous GVT algorithm [54] which uses acknowledgements and a central controller is shown in figure 1-6. An example of this algorithm with two processors is illustrated in figure 1-7. The first two messages (Start-GVT and Receive-Start) are for synchronizing the processors. When each processor receives the Compute-GVT message, the processor computes its LVT and sends this value to the controller. Finally the controller computes the GVT and broadcasts it to all of the processors.
A serious drawback of this algorithm is the need to block every processor in the system in the interval between receiving the Start-GVT message and the Compute-GVT messages. An alternative approach would be to let the processors keep processing the events when the GVT computation is in progress. This implies using an asynchronous GVT computation algorithm without the need for global synchronization points. Having such an algorithm may lead to simultaneous reporting problem. The simultaneous reporting problem occurs because not all of the processors report their local minimum at the same point in real time. Hence one or more messages can slip between the cracks and not count towards the calculation of GVT. Figure 1-8 illustrates an example of the simultaneous reporting problem. The controller broadcasts the Compute-GVT message. $P_2$ receives the message and reports its local minimum which is 25. The Compute-GVT that is sent to $P_1$ is delayed in the network. Meanwhile, $P_1$ sends a message with time stamp 15 to $P_2$. At this time $P_1$ receives the Compute-GVT message and reports that its local minimum is 30. Finally the controller computes an incorrect GVT value of 25.
Samadi [54] solved the simultaneous reporting problem by having processors send acknowledgements and tagging these acknowledgements during the period starting from the time that the processor sends its local minimum until it receives the new GVT value.

Fujimoto and Hybinette [20] solve the transient message and simultaneous reporting problems for shared-memory multiprocessors. In a shared-memory system the transient message problem is prevented because the sender writes a message into a memory buffer which is readable by the receiver and as a result the messages never disappear. The simultaneous reporting problem is solved because different processors do not perceive a different ordering of memory references to a shared variable. Implementing the GVT request via a write on a shared global memory buffer solves the problem.

![Diagram of the simultaneous reporting problem](image1.png)

**Figure 1–8: The simultaneous reporting problem**

### 1.4.3 Parallel VHDL

VHDL has had a great impact on computer aided design systems [13]. One of the first attempts at parallel VHDL simulation is [31]. In this article, the authors describe the implementation of an object-oriented Time Warp simulator for VHDL in an actor-based environment. Several partitioning algorithms were implemented. The authors evaluated their algorithm on benchmark circuits from ISCAS, achieving a speed up of 5 for one of these circuits.
In [36] the authors applied PDES to improve the performance of parallel VHDL simulation. In this paper the authors use synchronization protocols which let the simultaneous events to process in any order. Their approach based on Lamparts’s logical clocks. Allowing the LPs to self-adapt to optimistic or conservative mode, without the lookahead requirement, the simulation results of VHDL designs with 5531 to 14704 LPs using this method obtained almost linear speedup.

With the growing trend of hardware designs that contain both analog and digital components, we need environments to integrate analog and digital circuits together. A lot of effort have been made on simulation of digital or analog circuits, but not that much on simulation analysis of mixed analog-digital circuit. In [37], the authors applied parallel discrete event simulation for parallel simulation of VHDL designs which contain both analog and digital components. Their simulator is called Savant/TyVIS/Warped. Various optimization techniques are also applied to improve the performance of the simulator. According to their statement, the Savant/TyVIS/Warped is the only one of its kind that allows the parallel discrete event simulation of mixed-technology systems.

1.5 Time Warp for Digital Logic Simulation

One of the applications of Time Warp is the distributed simulation of large scale digital circuits. In this section, we introduce some of the Time Warp simulators for simulation of large scale digital circuits.

1.5.1 Clustered Time Warp

Avril et al. [5] designed a new optimistic synchronization scheme for distributed simulation called Clustered Time Warp (CTW). In this approach LPs are grouped
together to form a cluster. The inspiration behind this is that in large scale digital circuits, LPs belonging to the same functional units can be grouped together. 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 and a sequential algorithm is employed inside each cluster.

Each cluster has a structure referred to as a Cluster Environment (CE) which acts is the manager of the LPs in the cluster. The CE manages a Cluster Input Queue (CIQ) and a Cluster Output Queue (COQ). When an LP inside a cluster sends a message to an LP belonging to another cluster, a copy of the message (anti-message) is saved in the COQ. The receiving process puts the messages in its CIQ. Messages which travel between clusters are referred as external events. When an LP sends a message to another LP in the same cluster, the message is enqueued in the input buffer of the receiver. These messages are referred as internal events. The LPs within a cluster are scheduled by the CE and the processor schedules all of its CEs [5].

State saving depends upon the use of so-called timezones. For each cluster, the simulation is composed of a set of non-overlapping time intervals called timezones. When the simulation is started each cluster has one time zone with interval $[0, +\infty]$. When a cluster receives an external event with time stamp $t$, it finds the timezone interval $[t_i, t_{i+1}]$ into which $t$ fits ($t_i < t < t_{i+1}$) and decomposes it into two new timezones with intervals $[t_i, t]$ and $[t, t_{i+1}]$ [5].

In CTW, LPs have input queues but have no output Queue. Each LP saves the time stamp of the last event (TLE) it processed. The state of the LP is saved whenever the LP changes its timezone. When a straggler with time $t_s$ arrives, the CE
creates a new timezone and rolls back all of the LPs in the cluster which have a TLE
greater than \( t_s \) to a checkpoint prior to \( t_s \). It also sends appropriate anti-messages
for the messages in COQ. The significant difference from Time Warp is that all of
the events with a timestamp greater then that of the stragglers can be removed from
input queues. This is because all of the LPs in the cluster are rolled back, and these
events will be created again later. This protocol is referred as Clustered Rollback,
Clustered Checkpoint (CRCC). The reason for this name is that both of the decisions
for rolling back and checkpointing are taken at the cluster level [5].

Two other approaches, Local Rollback, Clustered Checkpoint (LRCC) and Local rollback, Local Checkpoint (LRLC) were also developed. In the former approach, the
decision for rolling back is made at each LP and checkpointing is managed by the
CE, while in the latter both of the decisions are made by the LP. The performance of
the above approaches was examined on two digital circuits of ISCAS’89 with 18000
and 20000 gates. Empirical results show that each of them decreases the maximal
memory usage of Time Warp. The percentage decrease is 40 percent for CRCC,
22 percent for LRCC, and 15 percent for LRLC. In addition, the execution time of
CRCC is 60 percent slower than that of Time Warp, while LRCC is 10 percent slower
than Time Warp. LRLC has a comparable speed to Time Warp.

1.5.2 DVS

Verilog is a Hardware Design Language (HDL) which describes the structural
and behavioural characteristics of digital circuits and allows the processes to execute
concurrently. This makes Verilog a suitable platform for distributed simulation. Li
et al. [34] developed the first *Distributed Verilog Simulator (DVS)* using the ideas embedded in Clustered Time Warp [5].

Icarus Verilog$^1$ is an open-source Verilog simulator which is composed of two important parts: the Iverilog compiler and a *Verilog Virtual Processor (VVP)* simulator. These two are connected by VVP assembly code. The Iverilog compiler translates the input Verlog file into the VVP assembly code. The VVP simulator, which is an event-driven simulator, processes the events and produces the final results. DVS receives the generated VVP assembly code of Iverilog as its input. The Verilog language supports both the structural and behavioural description of a circuit. While the structural description models the circuit as a network of interconnecting gates, the behaviour description takes into account the changes in the signals. These two are translated to *functor* statements and *thread* statements in the VVP assembly code [34].

The main structure of DVS is depicted in figure 1-9. The VVP Parser parses the VVP assembly code and instantiates structural and behavioural statements into functors and Vthreads respectively. Each functor is a digital gate with four inputs and one output. When any of the inputs changes during the simulation the value of the output port is updated. On the other side, Vthreads are employed to derive functors with input vectors. The Partitioner module in DVS provides an infrastructure for testing different partitioning algorithms. There is an abstract class, called ParBase, from which all partitioning schemes are derived.

---

1 S. Williams, Icarus Verilog, http://icarus.com/eda/verilog
The bottom layer in DVS is a communications layer which is used by the upper layers for message passing. The middle layer is a distributed simulation layer which called Object Oriented Clustered Time Warp (OOCTW). This is an object oriented version of Clustered Time Warp [5] which manages all of the Time Warp operations. The top layer is the simulation engine which is a copy of the sequential simulator engine [34].

It was observed that if Vthreads and functors are placed in the same partition, more rollbacks occur. Therefore DVS put the Vthreads in one processor and the functors were distributed between the remaining processors. The main design goals of OOCTW were (1) Integrating with the original Verilog simulator to limit the changes imposed to the sequential simulator. (2) Making the Time Warp library more reusable, readable and understandable. (3) Making the Time Warp library flexible so it can be a test bed for new optimization algorithms.

1.6 Dynamic Load Balancing

Dynamic load-balancing is an adaptive protocol [14] for Time Warp, i.e. it dynamically changes its behaviour according to changes in simulation. Adaptive
protocols are discussed in [14] [15] [57]. Some of these protocols control a time window in Time Warp by blocking overly optimistic execution.

Using the idea of Clustered Time Warp [5], Avril [6] developed a dynamic load balancing algorithm which increases the throughput of the system. The main goal in this approach is to evenly distribute the load between processors. The load of a cluster is defined as the number of events which were processed by the LPs of the cluster since the last load balance in the system. Consequently, the load of a processor is defined as the sum of all of the loads of the clusters within that processor. Load balancing is performed by transferring clusters from overloaded processors to under-loaded ones. The algorithm iteratively chooses the processors with the largest and the smallest loads and transfers half of the load difference ($\delta load/2$) from the most loaded processor to the least loaded one. Another metric which is considered during the execution of load balancing is the inter-processor communication. When the difference between the loads of two clusters is less than a certain value, we say that they have approximately the same load. Clusters with approximately the same load as $\delta load/2$ are selected. For each of these clusters the change in the inter-processor communication is estimated and finally the cluster with the lowest inter-processor communication cost is selected for this transfer. This algorithm is repeated iteratively until all the processors have almost the same load [6].

The throughput in Time Warp is defined as the number of non rolled back events per unit time [5]. When a move is initiated in the system, the throughput is decreased due to the time required to transfer LPs. When the transfer is completed the throughput increases and (hopefully) reaches a value larger than the one before
the transfer. Empirical results show that throughput was improved by up to 60 percent compared to Time Warp [6].

Schalgenhaft [55] also introduced a dynamic load balancing algorithm which incorporates LPs into clusters. In order to balance the load, clusters are transferred between processors. The performance of this algorithm was evaluated for small circuits on just 2 processors. Both algorithms were implemented on shared memory multi-processors which have a very small communications cost. This is not the case for current multi-computers with distributed memory.

1.7 Time Window

Time Warp is prone to an explosive growth in the number of rollbacks and to an excessive usage of memory. One approach to avoiding these problems is to limit the optimism by allowing only those events whose timestamps are within a certain time window to be executed optimistically[45]. The time window is defined by the interval $[GVT, GVT + W]$, where $W$ is the size of the window. Events within this interval can be executed, but those which have a timestamp beyond $GVT + W$ are not allowed to be executed, i.e. the LP is blocked. A blocked LP can still receive messages but cannot send messages except for messages involved in the $GVT$ computation. After a $GVT$ update, the window itself is updated. Previously blocked LPs are unblocked if their next scheduled event falls within the newly updated window. Approaches to computing the size of the time window may be found in [45][49][61]. Jun[61] achieved excellent performance by utilizing reinforcement learning approach[8].
1.8 Thesis Motivations and Structure

Time Warp simulators for digital logic circuits were used in [5] [34] [39] [40] [64]. While some effort on distributed VHDL simulation had been made in [30] [36] [37], this was not the case for Verilog. DVS [34] was the first environment for parallel Verilog simulation. Performance analysis of XTW [64] has shown that it outperforms both Time Warp and Clustered Time Warp [5], 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) [40]. Small benchmark circuits and synthetic circuits were typically used for gate level simulation experiments; our use of real designs in this work 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 hard to measure the load before a program starts, dynamic load-balancing during run-time has been employed for some time [2] [3] [59] [66]. As mentioned the dynamic load-balancing of parallel digital simulation is examined in [6] [55] for small circuits (up to 25k gates). As a result, we developed load balancing algorithms for parallel verification of real large circuits on multiprocessors.

An algorithm is described which selects a load-balancing algorithm and its associated parameters using reinforcement learning [24], an area of machine learning [10]. Reinforcement learning 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.

From the very beginning, many optimizations of TimeWarp have been proposed [5, 14, 17, 48, 56, 64]. We shall refer to these variations as Optimizing Time Warp, in the absence of a generally accepted term. The basic theme of these optimizations is to limit the optimism of Time Warp in order to prevent excessive memory usage, to reduce the number of rollbacks and to prevent, or at least limit, long rollback cascades. Simulated Annealing (SA)[27] is a search method used to find a good approximation for the global minimum of a function in a large search space. Due to the excellent performance of SA in solving optimization problems, it has been applied in many different problem domains [25] [44] [63]. We also utilized an SA algorithm to find the size of a time window for Time Warp. Moreover, we combined a dynamic time window computation with a dynamic load balancing algorithm into a unified algorithm.

One of the other approaches which is widely use for solving optimization problems is the genetic algorithm [22]. A genetic algorithm (GA) is a heuristic search which is widely used to generate useful solutions for optimization and search problems. Genetic algorithms use techniques inspired by natural evaluation such as inheritance, mutation, selection, and crossover. The Genetic algorithm starts with a random population of candidates for the search/optimization problem and tries to
find the optimal candidate by combining the exploitation of past results and exploration of the new search spaces. In this thesis, we also used genetic algorithms to tune the parameters of the load balancing algorithms and time window.

The rest of the thesis is organized as follows. In chapter 2, we describe Verilog XTW (VXTW) and its performance for real large designs. In chapter 3, we introduce two new dynamic load-balancing algorithms for Time Warp simulation of large circuit and study their performances. In chapter 4, we utilize Reinforcement Learning (RL) to tune the parameters of the dynamic load balancing algorithm. Chapter 5 introduces a Simulated Annealing (SA) algorithm for optimizing the size of the time window. Chapter 6 introduces a combined (dynamic load balancing and time window) approach to improve the performance of the parallel digital logic simulation. In chapter 7, we use Genetic Algorithm (GA) to tune the parameters of the load balancing algorithm and time window. In all the chapters, we provide the performance graphs of corresponding algorithms. Finally, the last chapter contains my conclusion and my thoughts about future work.
2.1 Introduction

The Verilog Hardware Description Language is an IEEE standard for designing Integrated Circuits (IC). Verilog can model both the behavioral and structural description of a circuit. An example of the behavioral and structural description of a one bit full adder from [46] is depicted in figure 2-1. More information about Verilog can be found in [46].

The basic element of the Verilog language is the module which contain behavioural and/or structural code. Verilog models a digital circuit as a set of modules which are connected by input and output ports. Different connected modules implement various functional units of the circuit and Verilog allows the concurrent execution of these functional units. The available concurrency within the Verilog modules makes it an appropriate infrastructure for parallel simulation [7].

2.2 XTW

In [64], Xu and Tropper developed a new event scheduling and rollback mechanism, XEQ and rb-message respectively, which improve the performance of 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
module \( FA(A, B, Cin, Sum, Cout) \);
\[
\text{Input } A, B, Cin
\]
output Sum, Cout;
reg Sum, Cout;
reg T1, T2, T3;
always @ (A or B or Cin)
begin
\[ Sum = (A \sim B) \sim Cin \]
\[ T1 = A \& B; \]
\[ T2 = A \& Cin; \]
\[ T3 = B \& Cin; \]
\[ Cout = (T1 \mid T2) \mid T3; \]
end
endmodule

module \( FA(A, B, Cin, Sum, Cout) \);
\[
\text{input } A, B, Cin;
\]
output Sum, Cout;
wire SI, T1, T2, T3;
xor
\[
X1 \oplus SI, A, B;
X2 \oplus Sum, SI, Cin;
\]
and
\[
A1 (T1, A, B);
A2 (T2, A, Cin);
A3 (T3, B, Cin);
\]
or
\[
O1 (Cout, T1, T2, T3);
\]
endmodule

Figure 2–1: An example of Verilog structural and behavioural code
extendable environment for Time Warp. XTW utilizes the characteristics of digital circuits and makes the following simplifying assumptions:

- Events are generated in chronological order.
- LPs receive messages 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 [64].

XTW employs clusters of LPs and the LRLC [4] technique from Clustered Time Warp [5]. 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 [64] 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 is sent to the corresponding InCh and appended to its ICEQ.
2. If the ICEQ is not empty, the smallest time-stamped event is submitted to LPEQ according to rule 2 at a cost of 1.
3. The ICEQ event is inserted into the LPEQ. The cost of finding the correct position is n which, in the worst case is the number of InChs at an LP.
4. If the LPEQ is not empty, the smallest time-stamped event is submitted to the CLEQ according to rule 3. The cost of finding the correct position is m, where m is 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 sent to other LPs. This message will cancel all of the messages which have a time stamp larger than the time stamp contained in the rb-message.

2.3 Design and Implementation of the Verilog Parser

Digital systems are 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 [64] 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)\textsuperscript{1} was used and a Verilog Parser was developed for the front-end.

2.3.1 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 logic. A wide range of design formats including Verilog, VHDL 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

\textsuperscript{1} http://www.synopsys.com

34
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 readable by
XTW.

2.3.2 Verilog Parser

The process of parsing consists of three steps: lexical analysis, syntax analysis
and code generation. In the first step, we use LEX [38] as a lexical analyser. LEX
uses a table of regular expressions to recognize all of the words in the input file and
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 state-
ment 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.

2.3.3 Architecture

The main architecture of the current simulator is illustrated in figure 2-2. It
takes a Verilog source file as the input and utilizes Synopsis DC to create GTECH
modules. These modules are created using the 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.
Figure 2–2: The main structure of the simulator
These bench files are the inputs to a distributed simulator. 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) [43] for communication between different processors.

2.4 Performance Evaluation of VXTW

Our experimental platform consists of 32 dual core, 64 bit Intel processors. Each of these processors has 8 Gigabytes of internal memory. The distribution of the load 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. As previously mentioned, we employ Message Passing Interface (MPI) as the communication platform between processors. MPI provides a reliable mechanism for sending and receiving messages between different processors.

The Verilog source files used in this simulation are an OpenSPARC T2 [58], the LEON processor [50] and two Viterbi decoders designed at the Renssalaer Polytechnic Institute (RPI). All of these circuits are open source designs. We used one
core of the OpenSPARC T2 which is synthesizable by Synopsis DC and has 400k gates. 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 Centre, part of the European Space Agency, and after that by Gaisler Research [50]. One of the specifications of the LEON processor is its configurable core which makes it suitable for System-on-Chip (SOC) designs. The LEON processor has around 200k gates. The other circuit which used in our simulation are two Viterbi decoders with 100k and 800k gates from Rensselaer Polytechnic Institute (RPI).

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

Figure 2-3(a,b) 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
Figure 2–3: Speedup vs. the number of processors (a) 10 input vectors, (b) 50 input vectors
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 3 (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 2-4(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 1.2M, 3M, 3.8M and 4M events per second for
the LEON, OpenSparc T2, RPI (smaller) and RPI (larger) circuits respectively when
the number of input vectors is 50. For 10 input vectors, the event processing rates
are 1.1M, 2.1M, 2.8M 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.

Let us define the commit rate as the number of non-rollbacked messages divided
by the total number of events. Figure 2-5(a, b) depicts the commit rate vs the
Figure 2–4: Total number of processed events per processor (a) 10 input vectors, (b) 50 input vectors
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 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 2-3. 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.
Figure 2–5: Commit rate vs. different number of processors (a) 10 input vectors, (b) 50 input vectors
CHAPTER 3
Dynamic Load Balancing

3.1 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 processors which execute a program. In order to achieve the best speed up, different processors should have approximately the same load. As in [6], 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.

The communication time for transferring the load in [6] 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 platforms which we describe in the next section. We utilized a combination of a centralized and a distributed approach to balance the load. In the following algorithms each processor sends its load and communication information to a master node. The master node utilizes this information to select (and inform) the processors which participate in the load-balancing algorithm. These processors then utilize local data to transfer LPs to other processors. This is different from
Avril’s [6] algorithm which forwards all of the LPs information to a central node. In Avril’s algorithm, the master node can become bottleneck for the simulation. This does not happen in our algorithms because each processor only sends some general information about its state to the master node.

3.1.1 General Structure of the Algorithms

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

We first introduce 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.

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

The load-balancing algorithm is initiated every $C$ GVT cycles. The type of the load-balancing algorithm (computation or communication) and the value of $C$ are defined by the user at the beginning of the simulation. We use a combination of centralized and distributed control in the algorithms. The main structure of the algorithms is as follows: each processor sends the values of $PLoad$ and $PComm$ to a central node. The central 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 overloaded node is informed about its corresponding under-loaded node. When an overloaded node receives a notice, it selects up to $L$ (an input parameter) of its LPs and sends them to the corresponding under-loaded processor. The next two subsections describe the details of the computation and communication load-balancing algorithms.

3.1.2 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 $PComm[i]$. 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 [21] 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 2 summarizes the computation algorithm.

3.1.3 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 $PLoad$, $LpLoad$, $PComm[i]$ and $LpComm[i]$ 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
Algorithm 2 The computation load-balancing algorithm

Each Processor $P_i$:

{Each C GVT cycle}

for each LP $j$ which $P_i$ hosts do

$P_{load_i} = P_{load_i} + LpLoad_j$

Send the $P_{load_i}$ and $PComm_i[]$ to the master node

end for

{On Receipt of 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 $O$

$O = O \cup P_{maxLoad}$

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

while $O! = Null$ do

maxLoad = $j$, where $P_j$ has the Max $\{PLoads\}_O$

for all the elements $e$ in $U$ do

if $PComm_{maxLoad[e]} > MaxCommuation$ then

$MaxCommuation = PComm_{maxLoad[e]}$

$MaxCommNode = e$

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
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 3 summarizes the communication algorithm.

### 3.2 Simulation Result

In this section we present performance results for the dynamic load-balancing algorithms. $P$ 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. Each experiment result is the average of 5 simulation runs. Dynamic load balancing algorithms consider the computation and communication information to dynamically balance the load during the simulation. Using the dynamic load balancing algorithm, we improved the total simulation time of all circuits.

Figure 3-1-a shows the performance of the computation dynamic load-balancing algorithm 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
Algorithm 3 The communication load-balancing algorithm

Each Processor $P_i$:

{Each C GVT cycle}

for each LP $j$ which $P_i$ hosts do

$\text{Pload}_i = \text{Pload}_i + L\text{pLoad}_j$

Send the $\text{Pload}_i$ and $\text{PComm}_i[]$ to the master node
end for

{On Receipt of Dynamic_Destination message}

Find the top L LPs which have the maximum value of $Lp\text{Comm}[\text{Destination}]$

Send the LPs to the destination processor

The Master Node:

while $\text{Selected} < 2P\%$ do

for $i = 1$ to $n - 1$ do do

for $j = 1$ to $n - 1$ do

if $(\text{MaxComm} < \text{PComm}_i[j])$ and ($P_i$ and $P_j$ were not matched) then

$\text{Selected}_1 = i$

$\text{Selected}_2 = j$

$\text{MaxComm} = \text{PComm}_i[j]$

end if
end for
end for

if $\text{PLoad}_{\text{selected}_1} > \text{PLoad}_{\text{selected}_2}$ then

$\text{Sender} = P_{\text{selected}_1}$

else

$\text{Sender} = P_{\text{selected}_2}$
end if

send the Dynamic_Destination message to the $\text{Sender}$

$\text{selected}++ = 2$
end while
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 3-1-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.

The performance of the computation dynamic load-balancing algorithm for different values of $L$ when $P =10\%$ on the OpenSparc T2 processor is depicted in figure 3-2-a. The simulation time is improved up to 18% with $L =150$. As depicted in figure 3-2-a, increasing the number of LPs from 100 to 150 results in better simulation time. On the other hand, the simulation time worsens if we increase the number of LPs to 400. Figure 3-2-b shows the same result when $P$ is changed to 20%. The simulation time of the OpenSparc T2 processor is up to 4% better with $P =10\%$ than with $P =20\%$.

Figures 3-3 and 3-4 show the same result for the communication load-balancing for small RPI and Leon processor respectively. Figure 3-3-a shows the performance of the algorithm for $P =20\%$ and different values of $L$. We get better results by changing the value of $L$ from 50 to 150, but if we increase $L$ to 400 the result worsens. The reason is again the cost of LP transferring. We can improve the simulation time by up to 17% with $L =150$. Figure 3-3-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 $L = 150$. For Leon processor, as depicted in figure 3-4-a, we can improve the simulation time by up to 18% with $L = 150$ and $P = 20\%$. Figure 3-4-b shows the same result for $P = 10\%$ and for different values of L. We improve the simulation time up to 22% with $P = 10\%$ and $L = 150$. 

52
Figure 3–1: The average simulation time of the computation load balancing algorithm on large RPI circuit for different values of $L$ and $P$: a) $P = 20\%$, b) $P = 10\%$
Figure 3–2: The average simulation time of the computation load balancing algorithm on OpenSparc T2 for different values of $L$ and $P$: a) $P = 20\%$, b) $P = 10\%$
Figure 3–3: The average simulation time of the communication load balancing algorithm on small RPI for different values of $L$ and $P$: a) $P = 20\%$, b) $P = 10\%$
Figure 3–4: The average simulation time of communication load-balancing algorithm on Leon for different values of L and P. (a) P = 20%, (b) P = 10%
CHAPTER 4
Reinforcement Learning for Optimizing Dynamic Load-balancing

4.1 Introduction

Reinforcement Learning (RL) is an area of the Artificial Intelligence (AI) 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 [53] 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) \( \pi \): 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 $a \in (A_{s_t})$ 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 $\pi$ 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:

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

(4.1)

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 $\pi$ indicates the expected return of policy $\pi$ starting from state $s$ as follows:

$$V^\pi(s) = E_\pi \{ R_t | s_t = s \} = E_\pi \left\{ \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} | s_t = s \right\}. \quad (4.2)$$

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

$$Q^\pi(s, a) = E_\pi \{ R_t | s_t = s, a_t = a \} = E_\pi \left\{ \sum_{i=1}^{\infty} \gamma_i (r_{t+i+1} | s_t = s, a_t = a) \right\}. \quad (4.3)$$

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.

### 4.2 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 [53] 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:

\[ \text{NewEstimate} = \text{OldEstimate} + \text{Stepsize}(\text{Target} - \text{OldEstimate}), \] (4.4)

where \( \text{Stepsize} \) is inverse of the number of steps and \( \text{target} \) 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:

\[ Q_{k+1} = \frac{1}{k + 1} \sum_{i=1}^{k+1} r_i = Q_k + \frac{1}{k + 1} (r_{k+1} - Q_k). \] (4.5)

The RL method which uses (4-5) 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 than the one produced by a greedy policy.

In order to select an action in a state, a straightforward approach would be to select the action which has the best value. We call this method exploitation and the selected action as greedy action. On the other hand, exploration means taking an action other than the greedy action. Exploration helps to discover actions which might be better than the greedy action. A method which combines these
two approaches is the $\epsilon$-greedy approach which attempts to balance exploration and exploitation. The $\epsilon$-greedy approach chooses a greedy action with probability $1 - \epsilon$, where $\epsilon$ is a small number. It has been shown that the $\epsilon$-greedy approach outperforms the greedy approach for the Q-learning algorithm [53].

4.2.1 Q-learning

In Q-learning agents learn to act optimally in a Markovian domain by experiencing the consequences of their actions. An agent can utilize Q-learning to acquire an optimal policy using delayed rewards. The agent can find the optimal policy even when there is no prior knowledge of the effects of its actions on the environment [62]. Q-learning utilizes the reward and the best value of the current state to improve the estimate of the previous state-action pair. The Q-learning update rule is:

$$Q(s, a_t) \leftarrow (1 - \alpha)Q(s, a_t) + \alpha [r_{t+1} + \gamma \max_a Q(s, a)].$$ (4.6)

where $\alpha$ and $\gamma$ are the learning step and the discount rate, respectively. We utilized the $\epsilon$-greedy approach to combine exploration and exploitation. Algorithm 4 shows the basic $\epsilon$-greedy Q-learning algorithm.

Algorithm 4 The Q-learning

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

61
4.3 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.

4.3.1 Single Agent vs. Multi-agent

RL can formulate the dynamic load-balancing of Time Warp as single agent or multi-agent [47] 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.

4.3.2 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 chapter 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.

4.4 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 $i$th GVT, $GVT_i$, the Event Commit Rate (ECR) of the $i$th GVT interval (the interval from $GVT_{i-1}$ to $GVT_i$) is defined as:

$$ECR_i = \frac{NC_i}{(t_i - t_{i-1})}, \quad (4.7)$$

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 [61], We define $ECR_{ref}$ as the average event commit rate since the beginning of the simulation:

$$ECR_{ref} = \frac{\sum_{i=1}^{D} NC_i}{(t_D - t_0)}. \quad (4.8)$$

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:
\[ R_i = ECR_i - ECR_{ref}. \] (4.9)

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 \(2^{mn}\) 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:

\[
R[a] \leftarrow (R[a] \times C[a] + r)/(C[a] + 1) \\
= R[a] + \frac{1}{C[a] + 1}(r - R[a]). \tag{4.10}
\]

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 5 presents the general structure of the N-armed Bandit learning algorithm.
Algorithm 5 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)
endif

4.5 Q-learning

We use Q-learning to control the dynamic load balancing of Time Warp. We utilize single state Q-learning techniques in this chapter. In the single state approach, as previously discussed, we have $2^{mn}$ different combinations for the control parameters, each of which represents an action in the learning algorithm. We utilize formula 4-5 as the value function. After calculating $ECR_{ref}$, the reward is calculated every $C$ cycles and the value of the current state-action pair is updated. As mentioned in section 4-2, we use the best value of the new state, $s_{t+1}$, to update the value of the current state-action pair.

At each cycle of the dynamic load-balancing algorithm, we either select the state-action pair with the largest average reward or with a probability of epsilon we randomly pick a state-action. Algorithm 6 presents the general structure of the single state Q-learning dynamic load-balancing algorithm.
Algorithm 6 The Q-learning and Dynamic load-balancing

Master Node ($P_0$):

{After each $C$ cycles the Q-learning method is run to select the following parameters:}

1. A) The type of the load balancing algorithm
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 to Computation then
    Run the computation load-balancing algorithm
else
    Run the communication load-balancing algorithm
end if

4.6 Simulation Result

In this section, we study the performance of the learning algorithms. The Verilog source files utilized in this simulation are the OpenSparc T2, the LEON processor and two Viterbi decoders designed at the Renssalaer Polytechnic Institute (RPI). Our experimental platform consists of 32 dual core, 64 bit Intel processors. In all of the graphs, $A=1$ and 2 indicate whether the type of the load-balancing algorithm is computation or communication respectively. $P$ is the percentage of nodes which participate in the load-balancing algorithm and $L$ indicates the number of LPs transferred in each cycle of algorithm. In the learning algorithms, the values of $\epsilon$, $\gamma$, $\alpha$ and $C$ are 0.1, 0.9, 0.1 and 5 respectively. $L$ can have the following 4 values: 50, 100, 150, and 200 and $P$ could be either 10% or 20%. Each experiment result is the average of 10 simulation runs.
We did many experiments on all of the circuits with different values of $P$ and $L$ and various load balancing algorithms. The effect of changing the type of the load-balancing algorithm, $L$ and $P$ on all circuits are depicted in previous chapter. 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 reinforcement learning algorithms was to learn the type of the dynamic load-balancing algorithm 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.

Figures 4-1(a,b) and 4-2(a,b) show the performance of different load-balancing algorithms and the N-armed Bandit method on the large RPI circuit, the small RPI circuit, the OpenSparc T2 and LEON processors respectively. $A = 1, 2$ correspond to computation load-balancing algorithm and communication load-balancing algorithm. 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 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.

Figure 4-3(a,b) and 4-4(a,b) shows the performance of different load-balancing algorithms and the Q-algorithm method on the large RPI circuit, the small RPI circuit, the OpenSparc T2 and LEON processors respectively. Using the Q-learning method, we can improve the simulation time up to 26%, 20.5%, 25.2% and 22.3% for OpenSparc T2, large RPI, small RPI, and LEON circuits respectively.
Figure 4–1: The average simulation time of computation, communication and N-armed Bandit load-balancing algorithms a) large RPI circuit, b) small RPI circuit
Figure 4–2: The average simulation time of computation, communication and N-armed Bandit load-balancing algorithms a) OpenSparc T2 and b) LEON
Figure 4-3: The average simulation time of computation, communication and Q-Learning load-balancing algorithms a) large RPI circuit, b) small RPI circuit
Figure 4–4: The average simulation time of computation, communication and Q-Learning load-balancing algorithms a) OpenSparc T2 and b) LEON
CHAPTER 5
Simulated Annealing for Finding the Size of Time Window

5.1 Introduction

While Time Warp potentially makes better use of the system’s parallelism, its over-optimism may lead to instability. In the worst case, the LPs spend most of their time rolling back in order to maintain causality, making it impossible for the simulation to progress[35]. Another major problem with Time Warp is its memory consumption, resulting from the need to save the states of LPs as well as events. One solution to this problem is to force an LP to block for a short period of time if its local time exceeds a certain virtual time (GVT). The virtual time constitutes the upper bound of a virtual time window, whose lower bound is the GVT. Approaches to computing the size of the time window may be found in [45][49][61]. Jun[61] achieved excellent performance by utilizing reinforcement learning approach[8].

Simulated Annealing (SA)[27] is a stochastic search method to find a good approximation to the global minimum of a function in a large search space. Due to the excellent performance of SA in solving combinatorial optimization problems, it has been applied in different applications, such as convex optimization[25], manufacturing cell formation[63] and the robust spanning tree problem[44]. In this chapter we utilize an SA algorithm in order to find the size of a time window for Time Warp protocol. Our simulation results show up to 30% improvement of simulation time in comparison to reinforcement learning[61].
5.2 Simulated Annealing

Simulated Annealing (SA) is a probabilistic heuristic approach originally proposed in [27] for global optimization problem of applied mathematics. For a given objective function $f$, SA can find a good approximation for the global optimum in a large search space. SA starts from an initial solution and attempts to move to a neighboring solution in order to improve the result. SA is grounded in an analogy pertaining to the thermal mobility of molecules, e.g. water changes from a liquid state to a pure crystalline form when the temperature is decreased.

5.2.1 The Simulated Annealing Algorithm

At each step of the algorithm, SA decides between moving the system to a new state $s'$ and staying in the current state $s$ where $s'$ is a neighbor of the current state. This decision is made probabilistically. The probabilities are chosen so that SA tends to move to states with a better objective function value. States are, of course, specified in an application-specific way. This step is repeated until a "good enough" solution has been determined or a computational budget has been exhausted.

The acceptance probability function $P(\Delta E, T)$ specifies the probability of making the transition from the current state $s$ to a candidate new state $s'$ according to the changes in the $\Delta E$ value of the function. If a new state results in a better value, it is accepted. However, if the new solution yields a worse value, there is a probability that it will still be accepted. This is accomplished by first randomly selecting a number from $(0,1)$. If the value is less than or equal to the value of the probability function, the new state is accepted; otherwise, it is rejected. This check prevents the algorithm from being stuck in a local minimum-a state that has a value which is
larger than the global minimum, but which smaller than any of its neighbors. The probability function is shown in formula 1[27]:

\[ P(\Delta E, T) = e^{-\Delta E / T}, \]

where \( T \) is the current temperature. This is known as the Boltzmann distribution. When \( T \) goes to zero, the probability \( P(\Delta E, T) \) goes to zero if \( \Delta E > 0 \), and to a positive value if \( \Delta E < 0 \). On the other hand, for small values of \( T \), the system will increasingly favor moves to state in which the objective function is improved and avoid those which worsen the objective function. When \( T \) is 0, the SA changes the state only if it improves the objective function.

Another feature of SA is the annealing schedule - the temperature is gradually reduced as the simulation proceeds. Initially, \( T \) is set to a high value in order to escape local minima. After a few steps the temperature \( T \) is reduced gradually according to some annealing schedule. The annealing schedule can be specified by the user but should end with a low value temperature by the end of the allotted computational budget. The evolution of the temperature \( T \) during the optimization process is also called the cooling schedule.

5.2.2 Simulated Annealing and Optimizing Time Warp

Time Warp is prone to an explosive growth in the number of rollbacks and to an excessive usage of memory. One approach to avoiding these problems is to limit the optimism by allowing only those events whose timestamps are within a certain time window to be executed optimistically[45]. The time window is defined by the interval \([GVT, GVT+W]\), where \( W \) is the size of the window. Events within this interval can
be executed, but those which have a timestamp beyond $GVT + W$ are not allowed to be executed, i.e. the LP is blocked. A blocked LP can still receive messages, but cannot send messages except for messages involved in the GVT computation. After a GVT update, the window itself is updated. Previously blocked LPs are unblocked if their next scheduled event falls within the newly updated window.

Note that if an objective function does not adequately reflect the main goal of the system, the SA algorithm may well fail to find an optimal policy. In Time Warp, the basic goal is to reduce the simulation time; hence the objective function should be related to the wall-clock time of the simulation. As chapter four, we select the Event Commit Rate (ECR), formula 4-7, as the objective function.

**Algorithm 7 SA\_initialize()**

1: $old\_wins[1..n] = random(1, MAX\_WIN)$; %set initial windows randomly
2: $bcast(old\_wins)$; %broadcast the initial windows to all nodes
3: $std\_rate = 0$;

The main process of our SA algorithm is described in algorithm 7 and algorithm 8. Since the result of the SA algorithm is independent of the initial choice, we set the window sizes of different nodes to a random value at the beginning of the simulation. Then, we use the first $C$ GVT cycles to run the SA algorithm and find a good choice for the windows. After that, we will use the result of SA algorithm as the window sizes in the remaining simulation.

During the SA running period, we calculate the commit rate in every $n$ GVT cycles, afterward we move to a neighboring window and try to find a better choice. In order to move to a new neighbor, we select some elements of the windows randomly and increase or decrease its size by a dynamic value, $D$, according to the acceptance
rate. $D$ is the distance between neighbors. It is an important parameter to adjust the SA process. We will show the effect of the distance in the next section. After finding a neighbor, we wait for $n$ GVT cycles and calculate the new event commit rate $ECR_{new}$. If $ECR_{new}$ is greater than old (previous) Event Commit Rate $ECR_{old}$, we accept the new neighbor as the current windows. Otherwise a random value between 0 and 1 is generated, if this random value is less than the probability function value in 5-1, we again select the new neighbor as the current windows. As the cooling process, $T$ is multiplied by a cooling factor, $F_c$, after each $M$ GVT cycles, where $F_c$ and $M$ are some user input parameters.

5.3 Experimental Results

In this section we present performance results for the simulated annealing algorithm. We utilized DVS[34] as our simulation engine and the C38417 and C38584 circuits from ISCAS-89 suite as our test benches. Our experimental platform consists
of 12 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 the Message Passing Interface (MPI) as the communication platform between processors.

Among the different parameters which affect the performance of the SA algorithm, the distance parameter is very important. The distance parameter is the Euclidean distance between neighboring window-vectors. In order to find a neighbor window-vector of a current vector, we specify a radius and randomly select one within an n-dimension sphere. If the distance is too small, the searching process advances too slowly and we cannot find the optimal choice within a limited number of steps. If the distance is too big, we may jump too far and not obtain the optimal choice between neighbors.

Figure 5-1 shows the effect of different distances on the performance of Time Warp for circuit C38584. In this figure, the X-axis represents the distance between neighbors and the Y-axis represents the simulation time. The numbers in the legend represent the number of nodes which participate in the SA algorithm. As the results indicate, the best distances when we have 4, 8 and 16 nodes are 1, 4 and 8, respectively. The best distance becomes bigger with more nodes because the choice becomes greater with more nodes, and we need a larger distance to find a better choice.

There is a relationship between the simulation time and the acceptance rate. Fig. 5-2 shows the acceptance rates for different distance values. In this figure,
the X-axis represents the distance between neighbors and the Y-axis represents the acceptance rate. The numbers in the legend represent the number of nodes which participate in the SA algorithm. As figures 5-1 and 5-2 imply, if the acceptance rate is close to 50%, the simulation time will be reduced by using the SA algorithm. The reason for this is that if the acceptance rate is too high, the neighboring window is too close to the original window and we need to increase the distance. On the other hand, if the acceptance rate is too low, the neighboring window is too far from the original window and we need to decrease the distance. Therefore, we applied an adaptive method to adjust the distance. If the acceptance rate is greater than 50%, we increase the distance; otherwise we decrease it.

Figures 5-3 and 5-4 show the simulation results for the C38417 and C38584 circuits with different algorithms. In these figures the X-axis represents the number of nodes that participate in the simulation and the Y-axis represents the simulation
time. The labels in the legend represent different methods which we utilize in order to find the size of the time window. “No-Win” represents the results without using a time window (the window is set to infinity); “avg-Win” represents the average results for all of the values of time windows; “RL” represents the results of a Reinforcement Learning algorithm with random candidates, in this work we implement a Q-Learning algorithm with one state; “SA” represents our simulated annealing algorithm. Both “RL” and “SA” methods are implemented on-line. As these figures indicate, the SA algorithm outperforms the other algorithms. The results show that simulation time is reduced by up to 52% when compared to the “No-Win” approach for the C38417 circuit. The best simulation time in [61] was achieved by applying the reinforcement learning approach. As we can see, our SA algorithm reduced the simulation time by up to 36% compared to the approach in [61] for the C38417 circuit with 24 nodes.
Figure 5–3: Simulation results of circuit C38417

Figure 5–4: Simulation results of circuit C38584
Figures 5-5 and 5-6 show the standard variance for the experiments for 100 repetitions with different methods. In the figures, the X-axis represents the number of nodes and the Y-axis represents the standard variance. The labels in the legend represent the different methods. From these figures, we can see that our SA algorithm gets more stable result than the RL algorithm. The reason for this is that the result of RL algorithm is decided by the quality of candidates which are input to the algorithm, while the SA algorithm can find a good solution given a rather large choice of possibilities. As for the overhead, the overhead of both RL and SA is very low (no more than 1% of the computation time), so we do not portray detailed results for the overhead in this section. The above experimental results clearly show that our SA algorithm can accelerate the simulation effectively as the speed-up obtained by our algorithm is better than any other existing method for choosing a time window.

Figure 5–5: Standard variance of circuit C38417
Figure 5–6: Standard variance of circuit C38548
CHAPTER 6
A Multi-Agent Approach for Optimizing Parallel Digital Logic Simulation

6.1 Dynamic Load Balancing

Dynamic load balancing for balancing either the computation or the communication loads in optimistic simulations were described in chapter 3. In the computation algorithm, processors forward their computation information to a central node. A central node determines the sending and the receiving processors. The senders select LPs and forward them to receiver processors. The communication load balancing algorithm considers communication information to match the processors. In this chapter, we utilize a combined version of these two algorithms which takes into account both communication and computation information in order to match the processors. In the following, we introduce a distributed load balancing algorithm which balances both the computation and communication loads of the processors.

6.1.1 Initiating Process

The load balancing algorithm is initiated every C cycles, where C is a predefined value. Each gate in the circuit is mapped to an LP. LPs are initially allocated among the processors before the simulation starts by a depth first search partitioning algorithm. We used the depth first search algorithm because of its low cost and good performance. The algorithm made use of the following parameters.
**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 this parameter for each LP.

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

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

**The Computation-Communication weight (λ):** This parameter 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 parameter represents the final load of the processor. PLoad is defined as the weighted sum of the computation and communication loads as follows:

\[
PLoad = \lambda \times PComp + (1 - \lambda)PComm
\]  

(6.1)

We employ a virtual ring topology for the processors. The load balancing algorithm is initiated every \( C \) cycles by a master node which sends a `START_LOAD – BALANCING` message to the first node in the ring. When the first node, \( N1 \), receives the message, it computes its \( PComp \) and \( PComm \). \( PComp \) and \( PComm \) are the sums of the \( LpComps \) and \( LPComms[ ]\)s of the LPs within that processor respectively. Using \( PComp \) and \( PComm \), \( PLoad \) of the processor is computed using formula 6-1. Afterwards, \( N1 \) adds its \( PLoad \) to the `START_LOAD – BALANCING` message and forwards the message to the next processor in the ring. The second node piggybacks the message to the next node in the ring with its own \( PLoad \) information. This process is continued until the first node receives the message again. At this point \( N1 \) extracts all of the \( PLoad \) information, passes the message to the next node in the ring and runs the load balancing algorithm. This is continued until all of the nodes get the load information. \( N1 \) destroys the message when it receives it again. Algorithm 9 shows the initialization step for the dynamic load balancing algorithm.

### 6.1.2 Dynamic Load Balancing Algorithm

All of the processors in the simulation run the algorithm, which utilizes \( PLoad \), \( PComm[] \) and \( LPcomm[] \) to balance the load. In the first step, the processor selects...
Algorithm 9 Initializing the Dynamic Load Balancing algorithm

The Master Node:
Send the START_LOAD-BALANCING message to the first processor in the ring

Each Processor $P_i$:

{After receiving the START_LOAD-BALANCING for the first time}

for each LP $j$ which $P_i$ hosts do

$P\text{Comp}_i = P\text{Comp}_i + Lp\text{Load}_j$

$P\text{Comm}_i = P\text{Comm}_i + Lp\text{Comm}[j]$

end for

$P\text{Load}_i = \lambda \ast P\text{Comp}_i + (1 - \lambda)P\text{Comm}_i$

Piggyback START_LOAD-BALANCING to the next processor in the ring

{After receiving the START_LOAD-BALANCING for the second time}

Extract all $P\text{Load}$ information from the message and pass it to the next node

the most over-loaded and under-loaded processors and puts them in two different sets: $O$ and $U$. Afterwards, each processor checks to see if it is in $O$ or $U$. If the processor is not in $O$, it ignores the rest of the algorithm.

Using the two sets of over-loaded and under-loaded nodes, a bipartite graph is constructed. The edges of the graph are the values of $P\text{PComm}[]$ which show the communication information between the processors of these two sets. As an example, if $P1$ and $P4$ are in $O$ and $U$ respectively, and $P1$ sent 10000 messages to $P4$ since the last run of the load balancing algorithm, an edge with the weight of 10000 exists between $P1$ and $P4$. Basically, this graph shows the communication information between the most over and under-loaded processors. We utilize the FM [21] bipartite graph matching algorithm to match the processors of these two sets.

After this matching, the processors in the $U$ set select up to $L$ LPs and forward them to their corresponding match in set $O$. The LPs which had the most communication with the destination processor are chosen for this transfer. $L$ is a
user-defined value. It is possible that the sender processor later receives messages intended for LPs which were already transferred. In this case, the sender processor forwards the messages to the new processors. Algorithm 10 summarizes the load balancing algorithm.

6.2 Reinforcement Learning and Dynamic Load Balancing of Time Warp

6.2.1 Multi-State Q-Learning

As mentioned in chapter 4, the main goal of the agent in the reinforcement learning problem is to develop a policy which maximizes the long term reward. The simplest approach would be a brute force search among all of the possible policies in an offline mode, selecting the one with maximum reward. This is not possible if we have a large number of policies. In addition, 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 state-action-value function, as the name implies, indicates the expected return under policy $\pi$, after taking action $a$ in state $s$. As mentioned in chapter 4, more formally we define it as:

$$Q^\pi(s, a) = E_\pi \{R_t | s_t = s, a_t = a\} = E_\pi \{\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} | s_t = s, a_t = a\}. \quad (6.2)$$

This is the total expected return under policy $\pi$ starting from action $a$ in state $s$. The RL problem can be solved by dynamic programming and the optimal policy
Algorithm 10 The load balancing algorithm

Each Processor $P_i$:

{After Receiving the START_LOAD-BALANCING Message for the Second Time}

while number of elements in $O < P\%$ do
    $maxLoad = j$, where $P_j$ has the Max $\{P_{load}\}$ 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 $\{P_{load}\}$ and $P_j$ is not in $U$
    $U = U \cup P_{\minLoad}$
end while

Copy$U = U$

while $O! = Null$ do
    $maxLoad = j$, where $P_j$ has the Max $\{P_{loads}\}_O$
    for all the elements $e$ in $U$ do
        if $PComm[maxLoad][e] > MaxCommunication$ then
            $MaxCommunication = PComm[maxLoad][e]$
            $MaxCommNode = e$
        end if
    end for
    match $P_{\maxLoad}$ and $P_e$
    $O = O - P_{\maxLoad}$
    $U = U - P_e$
end while

if $P_i \in CopyU$ then
    Destination = $P_i$'s match
end if

Find the top $L$ LPs which have the maximum value of $LpComm[Destination]$
Send the LPs to the Destination processor
determined if all the probability of rewards and state transitions are known. However, this is not often the case, and as a result statistical sampling methods are employed.

Having more than one state and different possible actions in each state, the main goal of a policy is to maximize the expected return. In order to do this, we wish to determine which action is "best" for each state. Different approaches for solving the reinforcement learning problem are proposed in [24]. Among them Multi-state Q-learning [62] achieves a good performance. Using the Multi-state Q-learning algorithm, agents learn to act optimally in a Markovian domain by experiencing the consequences of their actions. An agent can utilize Multi-state Q-learning to acquire an optimal policy using delayed rewards. A delayed reward is a reward that an agent receives because of an action it performed in the past. A significant feature of Q-learning is that if the agent does not know the effects of the action on its environment, it can still find the optimal policy. Multi-state Q-learning utilizes the reward and the best value of the current state to update the estimate of the previous state-action pair as follows:

$$Q(s_t, a_t) \leftarrow (1 - \alpha)Q(s_t, a_t) + \alpha [r_{t+1} + \gamma \max_a Q(s_{t+1}, a)]$$  \hspace{1cm} (6.3)

$\alpha$ and $\gamma$ are the learning step and the discount rate, respectively. In order to select the actions, we applied the $\epsilon - greedy$ approach.

6.2.2 The Dynamic Load Balancing Agent

We use a single-agent multiple-state approach to formulate the dynamic load balancing algorithm. In this approach, we have a central node which gathers information from all the nodes, runs the Multi-state Q-learning algorithm, finds new
values of the control parameters and informs other nodes about these values. Our single agent is run in the central node.

**Actions**

As mentioned, the dynamic load balancing algorithm runs every $C$ cycles. In each run, a matching between over-loaded and under-loaded processors is performed. Afterwards, the over-loaded processors select up to $L$ LPs which have the maximum communication with other nodes and sends them to the under-loaded processors. Hence, two important parameters of the dynamic load balancing algorithm are $L$ and $C$.

$L$ has an important effect on the performance of the dynamic load balancing algorithm. For different circuits and a different number of processors we need different values of $L$ to get the best result. If $L$ is large, each node sends a large number of LPs to the other nodes at each cycle of the algorithm. If the cost of this communication is more than the speed up that we achieve because of load balancing, the overall simulation time will increase. For a small number of processors and a large value of $L$, this is a possibility. On the other hand, if $L$ has a too small value, the load may not be balanced. Consequently, tuning the value of $L$ is important. $C$ also has a significant impact on the simulation result. If we run the algorithm too frequently, we increase the communications cost, while if we run it too infrequently we may fail to balance the load.

Experiments showed that for different circuits and different numbers of processors we needed different values for these parameters. Changing the value of these
parameters corresponds to actions in our algorithm. Note that if we have $m$ different values for $L$ and $n$ different values for $C$ the total number of actions for the Multi-state Q-Learning agent is $nm$.

**States**

We have defined four states for our dynamic load balancing algorithm utilizing the $\lambda$ parameter introduced in section 6-1-1. As mentioned, the $\lambda$ parameter shows the effect of the computation and communication loads on the dynamic load balancing algorithm. Larger values of $\lambda$ increases the effect of computational load and smaller values increases the effect of the communications load. Basically, with a large $\lambda$ (close to 1), the dynamic load balancing algorithm tends to balance the computational load and vice versa. The states are determined by comparing the average computation and communication load differences of the processors against specific threshold values. As mentioned, the computation and communication loads of a processor are defined by $P_{\text{Comp}}$ and $P_{\text{Comm}}$. If the average load difference of all the processors is less than a threshold, the state is considered to be unbalanced.

- $B_{\text{comp}}B_{\text{comm}}$ (Balanced Computation and Balanced Communication): In this state both the computation and the communication loads are balanced.
- $B_{\text{comp}}U_{\text{comm}}$ (Balanced Computation and Unbalanced Communication): In this state the computation load is balanced but the communication load is unbalanced.
• *UcompBcomm*(Unbalanced Computation and Balanced Communication): In this state the computation load is unbalanced but the communication load is balanced.

• *UcompUcomm*(Unbalanced Computation and Unbalanced Communication): In this state both the computation load and the communication load are unbalanced.

We consider these four states to be the states of the Multi-state Q-learning algorithm. In the first state, both the computation and communication loads are balanced, so we do not need to run the Multi-state Q-learning and load balancing algorithms. In the second state the communication load is unbalanced. In order to increase the effect of communication load on load balancing algorithm we decrease $\lambda$ to a value between 0 and 0.2. In the third states, the computational load is unbalanced so we increase the value of $\lambda$ to a number between 0.8 and 1. In the last state both the computational and communication loads are unbalanced. In order to balance both we set $\lambda$ to 0.5. We also need to learn the values of $L$ and $C$ in these three states. Hence, we run the Multi-state Q-learning algorithm in the last three states. If $C$ and $L$ have $m$ and $n$ different values respectively, then there are $mn$ combinations for the control parameters and we define $mn$ different actions in the last three states. As mentioned, we do not run the load balancing algorithm in the first state.
The transition between states is performed by comparing the average computation and communication load differences of processors with two predefined threshold values: \( T_{comp} \) and \( T_{comm} \), as shown in figure 6-1.

\[
\begin{align*}
B_{comp} &> T_{comp} \\
B_{comm} &> T_{comm} \\
U_{comp} &> T_{comp} \\
U_{comm} &> T_{comm}
\end{align*}
\]

Figure 6–1: The states and transitions from the first state in Multi-state Q-learning algorithm

The RL algorithm is executed in each of the \( C \) cycles. After \( C \) cycles all of the nodes send their data to a central node which executes the Multi-state Q-learning algorithm. After computing new values for the control parameters, it broadcasts them to all of the nodes.

### 6.2.3 The Time Window Agent

As mentioned, in order to prevent the number of rollbacks from becoming excessive, it is necessary to prevent some of the LPs from advancing too far in front of the other LPs, i.e. we need to control their optimism. The time window is a mechanism for doing this. The window is simply an interval \([T, T + W]\) in virtual time. We
use the size of the time window, $W$, and the $GVT$ again to define a bound on event execution at each LP-no event scheduled beyond $W + GVT$ is allowed to execute. Given this limit, if the next event at an LP has a time stamp beyond $W + GVT$, the LP is blocked. A blocked LP can still receive events from other LPs but cannot execute 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.

The main goal of the time window is to prevent an LP from going too far ahead of other LPs in simulation time, thereby causing rollbacks. The simplest approach to accomplish this is to have one action for each size of the time window. We defined an incremental unit, $U$, for the time window. If the agent has $k$ actions, $a$, then each action corresponds to a window size of $a * U$.

### 6.2.4 Multi-state Q-Learning and the Dynamic Load Balancing of Time Warp

As mentioned, the reward function is of fundamental importance to a Q-learning algorithm. The reward function should reflect the main goal of the system. We make use of the same reward function as chapter 4. If the $ECR_i$ represents the event commit rate of the $ith$ cycle, the reward of that cycle can be defined as:

\[ R_i = ECR_i - ECR_{ref}. \]  

(6.4)

where the $ECR_{ref}$ is the reference event commit rate. From this definition, the reward is positive if the simulation is faster than the reference rate during the last
cycle, otherwise a punishment (negative reward) is awarded. The event commit rate represents the speed of the simulation.

With \( m \) different values for \( L \) and \( n \) different values for \( C \), we have \( mn \) different actions for the dynamic load balancing agent. The time window agent also has \( k \) actions. We initially set the values of all of these actions to 0 and randomly select an action in the first step.

The Multi-state Q-learning algorithm is run every \( C \) cycles. The value of \( C \) itself is updated during the execution of the dynamic load balancing agent. The load balancing agent and time window agent are run iteratively. This means that after first \( C \) cycles we run the dynamic load balancing agent, it computes new values for the control parameters and broadcasts them to all of the nodes. After the next \( C \) cycles, we run the time window agent. This iteration continues throughout the course of the simulation.

The value of the current action for the agents is updated using formula 6–3. As mentioned in section 6-2-1, we use the best value to update the value of the current action. In the following steps, with probability \( 1 - \epsilon \) we select the greedy action and with probability \( \epsilon \) we randomly select an action (\( \epsilon - \text{greedy} \) approach). The detailed algorithm is demonstrated in algorithm 11.

6.3 Simulated Annealing for Time Warp

In this section, we describe a simulated annealing approach to tune the parameters of the dynamic load balancing algorithm and the size of the time window. The main drawback of the Q-learning approach is that it is necessary to specify the number and the values of the actions prior to executing the algorithm. Having a large
Algorithm 11 The Multi-state Q-learning for optimizing Time Warp

Master Node ($R_0$):

{After each $C$ cycles}

if ($i \mod 2 == 0$) then

{Run the dynamic load balancing agent}
Update the state using $T_{\text{comp}}$ and $T_{\text{comm}}$ thresholds

if (STATE==1) then
  Execute simulation only
else if (STATE==2) then
  Run the Multi-state Q-learning algorithm to update the values of $C$ and $L$
  Initiate the dynamic load balancing algorithm $\lambda = \text{random}(0, 0.2)$
else if (STATE==3) then
  Run the Multi-state Q-learning algorithm to update the values of $C$ and $L$
  Initiate the dynamic load balancing algorithm $\lambda = \text{random}(0, 0.8)$
else if (STATE==4) then
  Run the Multi-state Q-learning algorithm to update the values of $C$ and $L$
  Initiate the dynamic load balancing algorithm with $\lambda = 0.5$
end if
else
  {Run the time window agent to find the size of the time window}
end if
set of actions, the agent might not be able to find the best actions. In fact, one of the factors that affects the performance of the Q-learning algorithm is the time for exploring all of the actions (i.e. determining their performance). If the total simulation time allocated to a circuit is not large enough, the agent cannot explore all of the actions and the optimum policy may not be found. Another drawback of the Q-learning approach is finding suitable initial choices for the actions. SA, on the other hand, can start the search with any initial choice and can improve it by moving to the neighbors.

We utilize a vector of size three for our simulated annealing algorithm. The three elements of the vector are the same parameters as in the Q-learning algorithm:

1. C) The frequency of invoking the dynamic load balancing algorithm
2. L) The number of LPs that each overloaded processor sends to its corresponding underloaded processor
3. W) The size of the time window

We make use of the same objective function we used in Q-learning, the event commit rate. Because the SA algorithm is independent of the initial choice of states, we randomly initialize the state vector at the beginning of the simulation. The SA algorithm uses the first $K$ (a user input) cycles to determine the values of the vector $(C, L$ and $W)$. After finding these values, this vector is used for the rest of the simulation. Algorithm 12 contains the SA algorithm.

6.4 Experimental Results

In this section, we discuss our experimental results. We utilize VXTW [40] as our parallel simulation environment. Our experimental platform consists of 32 dual
Algorithm 12 Simulated Annealing Algorithm for Dynamic Load Balancing

\begin{itemize}
\item SA\_initialize()
\begin{itemize}
\item old\_vector[0] = random(1, MAX_C); \%set initial value for C randomly
\item old\_vector[1] = random(1, MAX_L); \%set initial value for L randomly
\item old\_vector[2] = random(1, MAX_W); \%set initial value for L randomly
\item broadcast(old\_vector); \%broadcast the initial vector to all nodes
\end{itemize}
\end{itemize}

\begin{itemize}
\item SA\_body()
\begin{itemize}
\item gather(Total\_Committed\_Events); \%gather committed events number
\item new\_rate = Total\_Committed\_Events/dt; \%calculate the event committed rate
\item if new\_rate > std\_rate then
\begin{itemize}
\item old\_vector = new\_vector;
\item std\_rate = new\_rate;
\item new\_vector = get\_a\_neighbor(old\_vector);
\end{itemize}
\item else
\begin{itemize}
\item \( \epsilon = e^{((new\_rate - std\_rate)/T)} \);
\item \( x = \text{random}(0, 1) \);
\item if \( x < \epsilon \) then
\begin{itemize}
\item old\_vector = new\_vector;
\item std\_rate = new\_rate;
\item new\_vector = get\_a\_neighbor(old\_vector);
\end{itemize}
\item else
\begin{itemize}
\item new\_vector = get\_a\_neighbor(old\_vector);
\end{itemize}
\item end if
\end{itemize}
\item end if
\item broadcast(new\_vector); \%broadcast the new vector
\item T\_counter \( + + \);
\item \( T = T_0 \times F_c(T_0^{T\_counter/M}) \);
\end{itemize}
\end{itemize}
core, 64 bit Intel processors. Each of these processors has 8 Gigabytes of internal memory. Each simulation point in our graphs is the average of 10 simulation runs. We assume a unit delay for gates and zero transmission time for the wires. The Verilog source files utilized in this simulation are the OpenSparc T2 processor, the LEON processor and two Viterbi decoders designed at the Renssalaer Polytechnic Institute (RPI). Initially, we distribute the gates between the processors using the depth first search (DFS) [28] algorithm. Message Passing Interface (MPI) is utilized for communications between processors.

As mentioned, in the dynamic load balancing algorithm $L$ is the number of LPs we transfer in each run of the dynamic load balancing algorithm and $C$ is the frequency of running the dynamic load balancing. In the Multi-state Q-learning algorithm, the values of $\epsilon$, $\gamma$ and $\alpha$ are 0.1, 0.9 and 0.1 respectively. In the dynamic load balancing agent, $L$ can have the following 4 values: 50, 100, 150, and 200 and $C$ can be 1, 5, 8 or 10. Hence, we have 16 actions at each execution of the Multi-state Q-learning algorithm for the dynamic load balancing agent. The time window agent can have a value between 1 and 8 for the window size, $W$. $U$ is the clock period.

In the simulated annealing algorithm, we set $n$, $\alpha$ and $F_c$ to 4, 0.8 and 10 respectively. $n$ is the frequency of running SA. Our experiments revealed that the best results achieved when $n$ was between 3 and 5.

Figure 6-2-a shows the simulation time for the dynamic load balancing algorithm for different values of $L$ when $C = 5$ on the OpenSparc T2 processor. The average load difference between all of the processors is decreased by up to 60% and we achieved up to a 29% improvement in the simulation time with $L = 150$. As can
be seen, increasing the number of LPs to 400 worsens the situation, while increasing the number of LPs from 100 to 150 results in better performance. The reason for this is that when we transfer many LPs in each round, the communication time for transferring the LPs increases and overwhelms the performance gain which we achieved from balancing the load. Figure 6-2-b shows the same result when $C$ is changed to 10. The simulation time of the OpenSparc T2 processor is up to 4% better with $C = 5$ than with $C = 10$. The reason for this is that we ran the load balancing algorithm less frequently and a load imbalance occurred in the simulation. The result of this load-imbalance worsened the simulation time.

Figures 6-3-a and 6-3-b depict the simulation results for the dynamic load balancing algorithm for different values of $C$ compared to Time Warp on the large RPI circuit. We ran the algorithm on the large RPI circuit and set $L$ to 50 and 200 respectively. As can be seen, increasing $C$ from 5 to 8 leads to better simulation results. This indicates that a frequency of 5 for dynamic load balancing algorithm is too large. Using a higher frequency for the dynamic load balancing algorithm results in more LP transfers. If this cost is more than the increase in simulation speed that we achieve because of the load balancing, the total simulation time worsens. As can be seen, when $C$ is 1 the simulation time is large. If we set $C$ to larger values (e.g. 15), the result is the same as the one obtained by Time Warp. Using the dynamic load balancing algorithm, we improve the simulation time by up to 30% and 25% with $L = 200$ and 5, respectively.

Figures 6-4(a-b) and 6-5(a-b) show the performance of the load balancing algorithms with different parameters and the multi-agent Q-learning algorithm method.
on the large RPI circuit, the small RPI circuit, the OpenSparc T2 and LEON processors, respectively. As mentioned, the learning algorithm optimizes the load-balancing and the time window at the same time. In all of the graphs, we depict the best results which could be achieved by choosing values for \( C \) and \( L \). As can be seen, in all of the cases, the multi-agent Q-learning method improves the simulation time more than other methods. An interesting point pertains to the simulation time of the algorithms with two nodes. Using the Multi-agent Q-learning method, we can improve the simulation time up to 45\%, 56\%, 42\%, and 54\% for the OpenSparc T2, large RPI, small RPI, and LEON circuits respectively. Compared to sequential simulation (i.e. one processor), the total simulation time is decreased up to 92.1\%, 92\%, 89.1\%, and 88\% for the OpenSparc T2, large RPI, small RPI, and LEON circuits respectively.

The distance parameter \((D)\) is the Euclidean distance between neighboring solution vectors in the SA algorithm. In order to find a neighbor solution-vector of a current vector, we specify a radius and randomly select a vector within an n-dimensional sphere. If the distance is too small, the search process advances too slowly and we cannot find the optimal choice within a limited number of steps. If the distance is too large, we may jump too far and not obtain the optimal choice. Our vectors have 3 parameters, \( C \), \( L \) and \( W \). Hence, we also need 3 values for \( D \). For the \( C \) parameter, we set \( D \) to 5. Each time a random number between 1 and 5 is generated and we add (subtract) it to the current value of \( C \). We do the same thing for \( L \), using a random number between 25 and 70 for \( D \). We set \( D \) to 1 for the \( W \) parameter.
The SA algorithm stops working when $T$ gets close to zero. However, if $k$ cycles have passed, we stop the SA even if $T$ is not close to zero. $K$ is equal to 70%-80% of total cycles. In most cases we have noted that $T$ goes to zero before the number of cycles reaches $k$.

Tables 6-1 and 6-2 show the percentage improvement in the simulation time (compared to Time Warp with static partitioning) resulting from the use of the dynamic load balancing, multi-agent Q-learning and SA algorithms. The results are for the LEON processor and for the large RPI circuit. Both the multi-agent Q-learning and SA algorithms result in substantial improvements, as much as 25-60% for 20 and 31 processors. Both algorithms outperform the dynamic load balancing algorithm by significant margins. An important observation is that the performance of the algorithms improves with the number of processors, a clear indication of their importance.

Table 6–1: The percentage of improvement in simulation time over Time Warp for dynamic load balancing. Reinforcement Learning (RL) and Simulated Annealing (SA) used for the large RPI circuit

<table>
<thead>
<tr>
<th>Number of Processors</th>
<th>Dynamic load balancing</th>
<th>RL</th>
<th>SA</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>16.1</td>
<td>28.7</td>
<td>35.6</td>
</tr>
<tr>
<td>12</td>
<td>21.9</td>
<td>33.8</td>
<td>40.2</td>
</tr>
<tr>
<td>20</td>
<td>31.31</td>
<td>43.4</td>
<td>50.3</td>
</tr>
<tr>
<td>31</td>
<td>46.9</td>
<td>56.5</td>
<td>60.4</td>
</tr>
</tbody>
</table>

Figures 6-6–(a,b) show the percentage improvement in simulation time over Time Warp (with static partitioning) obtained by using the multi-agent Q-learning and SA algorithms for the Opensparc processor and small RPI circuit respectively. The results are similar to those depicted in tables 6-1 and 6-2. In assessing these results, it
Table 6–2: The percentage of improvement in simulation time over Time Warp for dynamic load balancing. Reinforcement Learning (RL) and Simulated Annealing (SA) used for the LEON processor

<table>
<thead>
<tr>
<th>Number of Processors</th>
<th>dynamic load balancing</th>
<th>RL</th>
<th>SA</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>14.5</td>
<td>25.1</td>
<td>22.2</td>
</tr>
<tr>
<td>12</td>
<td>18.5</td>
<td>26.3</td>
<td>23.4</td>
</tr>
<tr>
<td>20</td>
<td>27</td>
<td>42.3</td>
<td>35.3</td>
</tr>
<tr>
<td>31</td>
<td>21.4</td>
<td>54</td>
<td>43.2</td>
</tr>
</tbody>
</table>

is important to understand that the results obtained by the Q-learning are dependent upon the choice of initial solutions. A different choice can easily result in worse results. On the other hand, the SA algorithm starts with a random solution and obtains comparable results.

We can conclude from the tables 6-1 and 6-2 and also figures 6-6(a-b) that for larger circuits such as the large RPI circuit, simulated annealing performs better than Q-learning. The reason for this is that the simulation time of this circuit is larger and as a result the simulated annealing algorithm has more time to adjust the parameters and find the best values for them. On the other hand for smaller circuits like small RPI circuit, it does not have enough time to find the optimum values.

Our final set of results pertains to the commit rate of the algorithms. The commit rate is defined as the number of non rolled back messages divided by the total number of messages. This is the metric which we make use of to determine the reward in the Q-learning algorithm and the intermediate results obtained by the SA algorithm. Figures 6-7-a and 6-7-b show the commit rate for the LEON processor and the small RPI circuit. As we can see, the commit rate decreases when we increase the number of processors. The reason for this that when the LPs are distributed over
more processors, the event cancellation time and the number of rollback messages increase. For the small RPI circuit, SA obtains better results than the RL algorithm, while in the LEON processor they have almost the same performance.
Figure 6–2: The average simulation time of the load balancing algorithm for different values of $L$: (a) $L = 50$, (b) $L = 200$
Figure 6–3: The average simulation time of the load balancing algorithm for different values of $C$: a) $C = 5$, b) $C = 10$
Figure 6–4: The average simulation time of the load balancing and Multi-agent Q-learning algorithms a) Large RPI and b) Small RPI
Figure 6–5: The average simulation time of the load balancing and Multi-agent Q-learning algorithms a) OpenSparc T2 processor b) LEON Processor
Figure 6–6: The average simulation time of the dynamic load balancing, multi-agent Q-learning and simulated annealing algorithms a) OpenSparc T2 processor b) Small RPI
Figure 6–7: Commit Rate VS. the Number of Processors a) LEON processor b) Small RPI Circuit
CHAPTER 7
A Genetic Algorithm for Parallel Digital Logic Simulation

A genetic algorithm (GA) is a heuristic search which is widely used to generate useful solutions for optimization and search problems. Genetic algorithms use techniques inspired by natural evaluation such as inheritance, mutation, selection, and crossover. The Genetic algorithm starts with a random population of candidates for the search/optimization problem and tries to find the optimal candidate by combining the exploitation of past results and exploration of the new search spaces.

7.1 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 [26, 29, 41], we develop one in this section to determine values for these parameters. A genetic algorithm (GA) is an evolutionary search algorithm in which generations of candidate solutions are iteratively computed by the algorithm [22]. 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. An outline for a genetic algorithm follows:
• Initialize the population with a random set of candidates.
• Evaluate each candidate in the set to find its fitness.
• Select the parents and put them in a mating pool.
• Recombine parents to produce new children.
• Mutate the children.
• Evaluate the children and find their fitness.
• Select individuals from the set of parents and generate children to form the new generation.
• If the termination condition has not satisfied, all steps from step 3 are executed again.

A classic approach to optimization problems is the hill-climbing approach [53], in which we start at a random point in a solution space and try to find the optimum solution by moving among neighboring solutions. The main drawback of the hill-climbing approach is that it might get trapped at a local minimum. GA solves the problem faced by the hill-climbing approach by choosing more than one starting point, i.e. by starting with candidate solutions.

7.1.1 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.
Event Commit Rate

Since the main goal of the parallel simulation is to decrease the simulation time, the elapsed wall clock time of the simulation plays a central role in defining our function. As previous chapters, we use event commit rate as the objective function:

\[
ECR_i = \frac{NC_i}{(t_i - t_{i-1})}, \tag{7.1}
\]

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 also result in a communication link becoming a bottleneck in the simulation. Hence, both computation and communication loads are considered by our algorithm. The average computational load imbalance, \(\text{AvgComp}\), is defined by

\[
\text{AveComp} = \frac{\sum_{j=1}^{N} \sum_{i=1}^{N} (i \neq j) \text{ABS}(PComp_i - PComp_j)}{2C(2, n)} \tag{7.2}
\]

Similarly, the average communication load imbalance, \(\text{AveComm}\), is defined as:

\[
\text{AveComm} = \frac{\sum_{i=1}^{N} \sum_{j=1}^{N} (i \neq j) \text{PPComm}_i[j]}{2C(2, n)} \tag{7.3}
\]

Finally, we define the Average Total Load-imbalance, \(\text{ATL}\), as the weighted sum of these two parameters.
\[ ATL = \lambda * AveComp + (1 - \lambda) * AveComm \]  

(7.4)

where \( \lambda \) is a user defined parameter between 0 and 1.

**Fitness Function**

In our genetic algorithm, the fitness value of each gene is defined by a both the event commit rate and the load imbalance.

\[ F_i = ECR * 1/ATL \]  

(7.5)

Where \( ECR \) and \( ATL \) both show the current average load-imbalance and event commit rate of the system.

**7.1.2 Tuning Parameters**

Our Time Warp simulator makes use of a bounded window in order to decrease the number 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 both the parameters of the dynamic load balancing algorithm and the window.

**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 nodes which participate in the load balancing algorithm (\( P\% \) in overloaded set and \( P\% \) in under-loaded set). Our experiments indicate that for our experimental platform (32 dual core, 64 bit Intel processors on a fast Ethernet), we can 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 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 platforms (e.g. various number 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.

**Window**

As mentioned, a large difference between the virtual times of the LPs in the simulation can lead to an excessive number of rollbacks. Hence we control the optimism of the simulation by defining a time window. The window is simply 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; any 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.

7.2 Implementation

7.2.1 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 [52], including binary, integer, floating-point phenotypic and tree representations. In this thesis, 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 fifteen 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.

7.2.2 Candidate Initialization and Evaluation

It is important to note that our genetic algorithm is an online algorithm, meaning that it executes concurrently with the simulation.

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 utilized different initialization sets.

The genetic algorithm is run in a central node. There is a master node 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 nodes. The central node also uses these values to calculate the fitness value of each candidate.
In order to evaluate the effect of current value of \( C \), we allow the algorithm to run 5 times. Since that dynamic load balancing algorithm is run every \( C \) cycles, we need \( 5C \) cycles to determine the fitness value of first candidates.

### 7.2.3 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 [16]. We make use of describe the Stochastic Universal Sampling (SUS) [16] algorithm.

As already mentioned, the fitness of each candidate can be evaluated using formula 7-5. With this fitness value, the probability of selecting a candidate as a parent can be calculated by:

\[
P_i = \frac{f_i}{\sum_{j=1}^{U} f_j}
\]  

(7.6)

where \( U \) is the population size. With these values of \( P_i \), an array \( a \) with \( U \) candidates 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 previous elements, i.e. it is the cumulative probability. Table 7-1 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 by \( rand + 1/U \) and \( i + 1 \) respectively. This process is continued until \( U \) candidates are selected. Algorithm 13 contains the details of the SUS.
Table 7–1: Fitness Values, Probabilities and Slot Values of Candidates

<table>
<thead>
<tr>
<th>Candidate</th>
<th>Fitness Value</th>
<th>Probability</th>
<th>Slot Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.3321</td>
<td>0.1670</td>
<td>0.1670</td>
</tr>
<tr>
<td>2</td>
<td>0.2213</td>
<td>0.1113</td>
<td>0.2783</td>
</tr>
<tr>
<td>3</td>
<td>0.1156</td>
<td>0.0581</td>
<td>0.3364</td>
</tr>
<tr>
<td>4</td>
<td>0.4987</td>
<td>0.2508</td>
<td>0.5872</td>
</tr>
<tr>
<td>5</td>
<td>0.1137</td>
<td>0.0571</td>
<td>0.6443</td>
</tr>
<tr>
<td>6</td>
<td>0.2087</td>
<td>0.1049</td>
<td>0.7492</td>
</tr>
<tr>
<td>7</td>
<td>0.067</td>
<td>0.0337</td>
<td>0.7829</td>
</tr>
<tr>
<td>8</td>
<td>0.431</td>
<td>0.2167</td>
<td>1</td>
</tr>
</tbody>
</table>

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 performance parameters within each chromosome, we utilized the so-called three section crossover. 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 7-1 illustrates our crossover approach. The three cross over points are selected by choosing random numbers between zero and the length of each section.

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. From the algorithmic point of view, mutation prevents the algorithm from being trapped in a local minimum.

7.2.4 Survivor Selection

After the mutation step, we need to find the fitness value of all the new candidates. In order to do so, we run the load balancing algorithm with corresponding
Algorithm 13 The Stochastic Universal Sampling

for each candidate \( j \) in the solution set do
\[
a[j] = \sum p_i, 1 \leq j \leq U
\]
end for

Generate a random number \( \text{rand} \) in the range \([0, 1/U]\)

\( n = 0 \)

\( i = 1 \)

while \( n < U \) do
\[
\text{while } \text{rand} \leq a[i] \text{ and } n < U \text{ do}
\quad \text{Add candidate } i \text{ to the mating pool}
\quad \text{rand} = \text{rand} + 1/U
\quad n = n + 1
\end{verbatim}
\end{verbatim}
\( \text{end while} \)
\quad i = i + 1
end while

Figure 7–1: The Crossover Operation
values of each candidate. The size of the time window is also set to the window size of the current candidate. As mentioned, one of the parameters that we try to tune is the running frequency of the load balancing algorithm. In order to consider this parameter, the load balancing algorithm is run 5 times for each 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 some 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 thesis, 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 these candidates. If their age is greater than a pre-determined value, we remove the candidate 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.
7.3 Performance of the Genetic Algorithm

In order to evaluate the performance of the genetic algorithms we make use of VXTW [40]. Our simulation environment has 32 dual core 64 bit Intel processors. Each processor has 8 GBytes of internal memory. We utilized Message Passing Interface (MPI) [43] 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 distributed the gates between the processors using a depth first search (DFS) algorithm with load balancing constraints.

Recall that \( L \) represents the number of LPs which we transfer in each cycle of the load balancing algorithm, \( P \) is the percentage of nodes which participate in the algorithm, \( C \) is the frequency of running it. We set \( \lambda \) to 0.6. As mentioned, our experiments indicated that for this platform, we could not set \( P \) to more than 40\% because of load-transferring overhead, while values less than 30\% reduced the impact of dynamic load balancing algorithm. Hence, we set \( P=40\% \), meaning that 80\% of the processors participate in the load balancing algorithm. 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.

The effect of the window size \( W \) without dynamic load balancing is illustrated in figures 7-2 for the small RPI and LEON processor circuits. The results portrayed
in these two figures indicates that the value of $W$ plays an important role. In figure 7-2-a, the best result is achieved when we set $W$ to 8. Increasing $W$ to 16 and 32 worsens the results. 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. Similar results were achieved for the LEON processor.

Our experiments indicated that for various circuits and platforms, it is necessary to pick optimal values for the tuning parameters in order to get good performance. In order to find these values, we utilized a genetic algorithm. Figures 7-3 and 7-4 show the speed up for (1) the static time warp algorithm (2) the dynamic load balancing algorithm we introduced in chapter 6 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 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.
Figure 7–2: Average simulation times for different values of $W$
Figure 7-3: Speed-up utilizing the genetic algorithm
Figure 7–4: Speed-up utilizing the genetic algorithm
8.1 Conclusion

In this chapter, we summarize the chapters of thesis and also talk about my thoughts for the future work. In the first chapter of this thesis, we briefly review the state of the art efforts for parallel gate level simulation and also studied the details of the well-known optimistic concurrency control algorithm named Time Warp.

In the second chapter, we make use of XTW (as the simulation engine) to develop VXTW, which is capable of parsing Verilog designs. We utilized XTW because it exhibited the best performance among the Time Warp based circuit simulators. We use the Synopsis Design Compiler to generate GTECH modules from Verilog source files and develop a Verilog parser to convert the GTECH modules into a flattened bench file. The performance of the simulator is 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 obtain an event rate of 4M events per second for the Viterbi decoder circuit on 32 processors. We note that the speedup depends not just on the size of the circuit, but on its complexity as well-the flat RPI designs exhibits a better speedup than the more complicated open source designs. This observation may be put to use in the design of partitioning 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. As a result, in chapter three, we develop two new dynamic load balancing algorithm for parallel digital logic simulation. The algorithms utilize a combination of a centralized and a distributed approach for selecting the LPs which are to be transferred and try to balance the computational and communication loads during the simulation. Using computation and communication load balancing algorithms, the simulation time is improved up to 20\% comparing with parallel simulation without load balancing.

The experimental results for the dynamic load balancing algorithms showed that for different circuits and different topologies (different number of processors) we need to utilize different algorithms with different parameter values in order to obtain the best possible performance. As a result, in chapter four, we apply machine learning to improve the performance of load balancing algorithms. We utilize reinforcement learning algorithms which learn to select the type of the dynamic load balancing algorithm (communication or computation) and adjust the parameters of the algorithm. In particular, in chapter four, we use the N-armed Bandit method and Q-learning methods. Experimental results revealed that the simulation time is reduced using these approaches.

While time warp potentially makes better use of the system’s parallelism, its over-optimism may lead to instability. In the worst case, it is possible that the LPs
spend most of their time rolling back in order to maintain causality, making it impossible for the simulation to progress. As a result, in chapter 5 a time window is applied to control the optimism of LPs. In this chapter, we apply a simulated annealing algorithm to find the optimum value of the time window in chapter five. The SA algorithm reduced the simulation time by up to 36% compared to the approach in [61] for one of the circuits in ISCAS-89.

In chapter six, we utilize a distributed dynamic load balancing algorithm which is a combination of computation and communication algorithms to balance the computational and communication loads during the simulation of gate level circuits. In order to tune the parameters of the dynamic load balancing algorithm and also find the optimum value of the time window we use reinforcement learning and simulated annealing. The reinforcement learning algorithm has two agents which try to optimize the load balancing algorithm parameters and the size of time window at the same time. A multi-state reinforcement learning approach is utilized which has a better performance than the single state one. In the simulated annealing algorithm, a vector which represents the load balancing parameters and time window is constructed and optimized. The simulation time is improved up to 45%, 56%, 42%, and 54% for the OpenSparc T2, large RPI, small RPI, and LEON circuits respectively. Compared to sequential simulation (i.e. one processor), the total simulation time is decreased up to 92.1%, 92%, 89.1%, and 88% for the OpenSparc T2, large RPI, small RPI, and LEON circuits respectively.

In chapter seven, we used a genetic algorithm to tune the parameters of load balancing algorithm and time window. My experiments indicate that the values of $L$
and $C$ have a significant effect on the performance of the load balancing algorithm. Each candidate solution in the genetic algorithm consists of $L$, $C$ and $W$. Using this algorithm, we improved the simulation time 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.

8.2 Future Work

In my thesis, we applied a single agent Q-learning technique to tune the parameters of the load balancing algorithm. In this algorithm a learning agent tries to find optimal values for the load balancing parameters by interacting with an environment. For the distributed algorithm, a multi-agent learning approach can be utilized to tune the algorithms’ parameters at each node. In a multi-agent algorithm, different nodes can have different values for the control parameters. In this approach the learning of one agent affects the learning of other agents because the agents have to cooperate with each other in order to find optimal values for the control parameters. In such an environment, there is no single point of failure. Because there will be no bottleneck, the algorithm will be more suited to large platforms.

Because it has become harder to exploit higher CPU clock frequency due to several physical issues (e.g. heat, power consumption), current computer processor architectures tend to move towards using multi-core processors [18]. Multi-core computers offer new levels of energy efficient performance and a good performance-price ratio. As a result they could be a viable and less costly alternative for multi processors. Cores in a multi-core device may couple together tightly or loosely. They
may or may not share caches, and they may use message passing or shared memory communication methods. Most of the current operating systems consider each virtual processor or homogeneous core as a separate, independent CPU. As a result, these systems are handled like classic multiprocessor platforms. Therefore, the OS may fail in fully exploiting the capabilities of the system. As no work has been done on the performance of optimistic parallel algorithms for some large systems (e.g. biological systems) on multi-core processors, I plan to study the performance of these algorithms and current operating systems on multi-core processors. Moreover, I would like to use my experience in developing dynamic load balancing algorithms for multi processors to develop centralized and distributed load balancing schemes for multi-core CPUs.

General Purpose Graphical Processing Units (GPGPUs) are another highly efficient and cost-effective platform which can be applied for parallel processing. Current GPGPUs have hundreds of processing elements which can be utilized for efficient parallel processing. While there are some studies regarding the performance of conservative parallel algorithms on GPGPUS [60], work has really just begun in this area. As the performance of optimistic simulation for many applications (e.g. digital logic simulation) is much better than the conservative approach on parallel processors, I would like to investigate optimistic simulation on GPUs for applications which require huge amount of computation. In addition, the communication between the processing elements (PE) of a GPU is several hundreds of magnitude faster than the inter-processor communication on multiprocessors. As a result, I plan to investigate the performance of load balancing algorithms on GPU platforms.
References


[64] Qing XU and Carl Tropper. Xtw, a parallel and distributed logic simulator. pages 1064–1069, 2005.
