Event Reconstruction in Time Warp

Lijun Li and Carl Tropper
School of Computer Science
McGill University
Montreal, Canada
lli22, carl@cs.mcgill.ca

Abstract

In optimistic simulations, checkpointing techniques are often used to reduce the overhead caused by state saving. In this paper, we propose event reconstruction as a technique with which to reduce the overhead caused by event saving, and compare its memory consumption and execution time to the results obtained by dynamic checkpointing. As the name implies, event reconstruction reconstructs input events and anti-events from the differences between adjacent states, and does not save input events in the event queue.

For simulations with fine event granularity and small state size, such as the logic simulation of VLSI circuitry, event reconstruction can yield an improvement in execution time as well as a significant reduction in memory utilization when compared to dynamic checkpointing. Moreover, this technique facilitates load migration because only the state queue needs to be moved from one processor to another.

1 Introduction

Modern VLSI systems are becoming increasingly complicated, posing a never-ending challenge to sequential simulation. To accommodate the growing need for increased memory as well as the need for decreased simulation time, it is becoming increasingly necessary to make use of distributed simulation[12].

Time Warp[8] is an appealing technique for the distributed logic simulation of VLSI circuitry because it can potentially uncover a high degrees of parallelism in the VLSI system being simulated. However, this benefit is at the expense of the processing time and memory required for state saving, state restoration and event saving. Several techniques[10, 3, 4, 5] have been proposed to reduce this overhead, all of which focus on reducing the overhead for state saving and state restoration.

Event saving is a significant cost in Time Warp. However, to the best of our knowledge, no research has been directed at reducing its associated cost. In optimistic simulation, every event has to be saved in case of a rollback. However, in optimistic logic simulation of VLSI circuits, the event size is larger than the state size and, as a consequence, event saving costs more in memory and processing time than does state saving. It follows that if we try to reduce the overhead of event saving, the resulting performance may well be better than that obtained by reducing the overhead associated with state saving alone. It is unclear if this improvement can be achieved for coarse-grained models.

The rest of this paper is organized as follows. Section 2 is devoted to related work. In section 3, we introduce logic simulation. Our distributed simulation environment DVS[9] is briefly described in section 4. In section 5, we present the details of our event reconstruction technique. A comparison of the memory consumption and the execution time making use of event reconstruction and dynamic checkpointing is presented in section 6. The last section contains our conclusions and thoughts about future work.

2 Related work

A number of algorithms have been proposed to reduce the memory overhead caused by state saving, including incremental saving[3], checkpointing[10, 4], reverse computation[5] and rollback relaxation[13].

[7] presents a comparative analysis of four approaches to dynamically adjusting the checkpoint interval and proposes an algorithm for dynamic checkpointing. The algorithm tries to balance the time spent saving state versus the time spent coasting forward. The goal of the algorithm is to minimize the time for state saving and coasting forward and to adjust the checkpoint interval accordingly. In our experimental section6, we compare the performance of event reconstruction and this heuristic algorithm.

Checkpointing results in a lower memory consumption

Proceedings of the 18th Workshop on Parallel and Distributed Simulation (PADS'04)
1087-4097/04 $20.00 © 2004 **IEEE**

value	*Purpose*	*Value Encoding*
0	Forcing zero	00
1	Forcing one	01
X	Forcing unknown	10
Z	High impedance	11

Table 1. Logic values and their purposes

and an improved execution time. However, it is difficult to achieve the optimal frequency of checkpointing. Dynamic checkpointing can be used to alleviate this problem. However, it faces the problems of choosing tuning parameters, including the initial checkpointing frequency, the average cost of event processing and the average cost of coasting forward.

Reverse computation[5] computes state variables by reversing the operation sequence applied on the variables. It uses compiler-based techniques to generate the reverse computation code automatically. As a result, its implementation is more complex, although it is able to provide a significant performance improvement over checkpointing.

In rollback relaxation[13] all LPs are classified into two categories, memoryless and LPs and LPs with memory. A memoryless LPs' output is determined by the values of its inputs. Therefore, no state is saved for memoryless LPs. Instead, the LP reconstructs any required input state from the events of the input queue. The rollback relaxation mechanism is able to reduce the state saving overhead by a considerable amount in logic simulations because most LPs(AND, OR, XOR gates) in such simulation are memoryless.

3 Logic simulation and its characteristics

3.1 The Simulation Model

In a logic simulation, the LPs represent logic gates (AND,NAND,NOR,OR). The incoming channels to an LP correspond to the fanin list of wires of a logic gate while the outgoing channels correspond to its fanout list.

The circuit model uses a finite set of values to represent the type of signal propagating throughout the circuit. The 4 values which a signal may have are portrayed in table 1.

A signal change is modeled as an update event which contains a timestamp, source and destination gate identifications and a value which corresponds to the new value of the wire. When an LP receives an update event, it sets its local clock to the timestamp of the event, and then evaluates its output and schedules the resulting output change as update events to its fanout list.

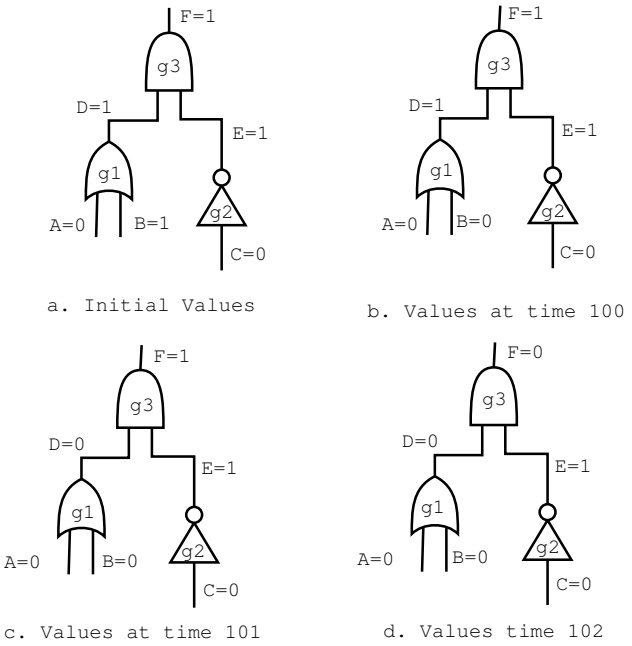

Figure 1. Logic simulation of a digital circuit

3.2 Discrete Event Logic Simulation

Figure 1 represents a simple logic circuit consisting of three gates. The circuit has three inputs(A, B and C), one output(F) and two internal wires(D and E). Assume that each gate has unit (processing) delay. At the initial point of our example, the logic gates have the values shown in Figure 1.a. An event occurs on wire B at time 100, changing it from 1 to 0 as shown in Figure 1.b. At time 100, gate g1 is evaluated to see if there's a change on its output D. Since D will change from 1 to 0, this event is scheduled for a unit delay in the future.

At time 101, gate g1s' output D will be set to 0 as indicated in Figure 1.c and this new value will be propagated to the gates on g1s' fanout, g3 in this circuit. Then g3 is evaluated to see if there will be an output change on F. As can be seen in Figure 1.d, F will change from 1 to 0.

3.3 Characteristics of logic simulation

In this section we discuss the characteristics of logic simulation which inspired our work on event reconstruction.

- Relatively small state size

  In the implementation of a logic simulator, such as DVS [9], the 4 signal values are encoded with two bits as shown in table 1. Every gate has up to four inputs and one output. Therefore, the state of a gate in Figure 2 includes ival and oval, each of which are one byte in

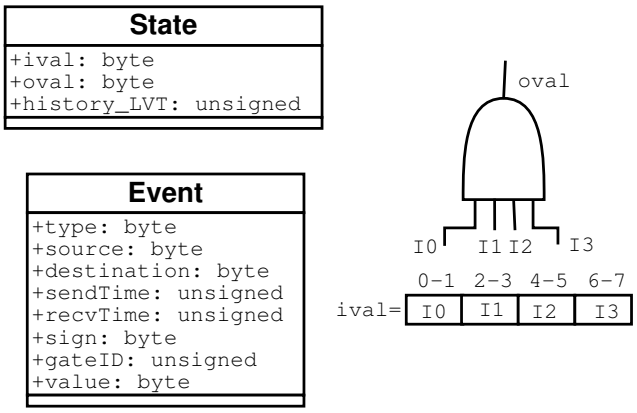

Figure 2. The size of the state and the event

length. The bits 0-1 are used to store the value of I0, bits 2 and 3 for I1, 4 and 5 for I2 and bits 6 and 7 for I3, as shown in Figure 2. For example, if ival is equal to 00001001, we know that I0 is equal to logical value 0, I1 logical value 0, I2 logical value 'x' and I3 logical value 1. This compact storage helps to save memory. The size of the state is only 16 bytes in DVS[9]. However, checkpointing has its greatest value when the size of the state is large.

- Large event size

  Figure 2 shows the structure of an event. The size of an event is 56 bytes, almost four times of the size of a state. Therefore, event saving causes at least 3.5(56/16) times more memory to be used than state saving if state saving is done for every event processed. In order to underscore this point, the amount of memory consumed in the simulation of a 16 bit multiplier is presented in our experimental section.

- Large event population

  The event population is large because of the large number of gates, each of which is mapped to an LP. For example, the event population is 8,129,815 for s38584(about 20K gates) when the clock is 500kHz and the number of random input vectors is 100. If every event has to be saved, the associated memory consumption will be very large.

- Fine event granularity

  Logic simulation is known for its fine event granularity. Thus, the performance of distributed logic simulation is especially sensitive to the overhead caused by state saving and event saving. Reducing this overhead would certainly be useful for performance improvement.

With these characteristics of logic simulation in mind, we decided to reduce the memory occupied by events instead of reducing the memory consumed by states. The following section describes our approach to event reconstruction.

4 DVS[9]: A framework for distributed Verilog[11] simulation

Before we present the implementation of event reconstruction, we give a brief introduction to DVS, a framework for distributed Verilog simulation. Event reconstruction and dynamic checkpointing are implemented in this framework.

Figure 3 portrays the architecture of DVS. The 3 layers of DVS are shown on the right side of figure 3. The bottom layer is the communication layer, which provides a common message passing interface to the upper layer. Inside this layer, the software communication platform can be PVM or MPI. Users can chose one of them without affecting the code of upper layers.

The middle layer is a parallel discrete event simulation kernel, OOCTW, which is an object-oriented version of Clustered Time Warp (CTW)[1]. It provides services such as rollback, state saving and restoration, GVT computation and fossil collection to the top layer.

The top layer is the distributed simulation engine, which includes an event process handler and an interpreter which executes instructions in the code space of a virtual thread.

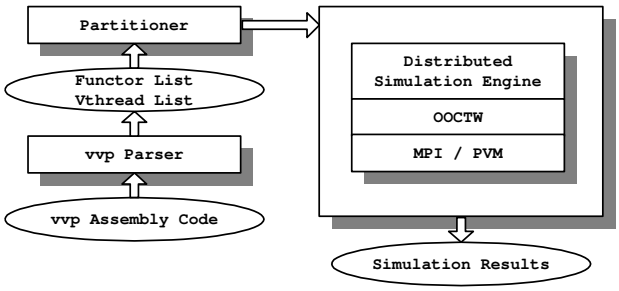

Figure 3. Architecture of DVS

A partitioner implements several partitioning algorithms including RANDOM[2], BFS(Breath-First-Search)[2], DFS(Depth-First-Search)[2] and CAKE[6]. More advanced partitioning algorithms are still under investigation as part of our research. CAKE results in a minimal inter-processor communication time and the best speed-up of the above algorithms because it takes advantage of the hierarchical design information in the Verilog source file.

5 Implementation of Event Reconstruction

In this section, we explain the implementation of event reconstruction in detail. The data structures and algorithms which comprise this approach are described below.

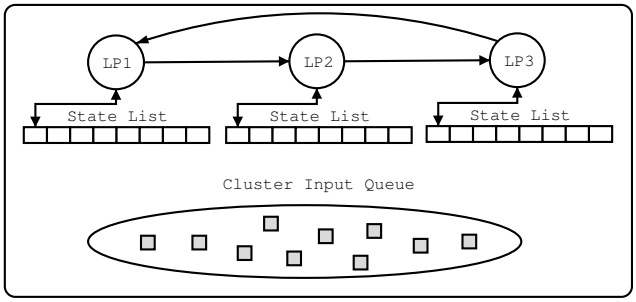

Figure 4. Cluster structure

5.1 Data structure

The data structure for event reconstruction is shown in figure 4. In DVS, the Cluster is the container and scheduler of all of the LPs. An LP maintains a state list. In event reconstruction *i*t is necessary to store all of the states at an LP. Since we build the input events and anti-events from the state list in our approach, *w*e don't need the input event list and output event list for every LP. The unprocessed events for all of the LPs in the same cluster are stored in a single priority queue data structure. The LP scheduling strategy is smallest timestamp first. We note in passing that the GVT computation also benefits from the single queue data structure.

5.2 Event annihilation

Time Warp uses a tuple (LPID, timestamp, eventID) to match positive events and their corresponding anti-events. The LPID is globally unique, indicating which LP will receive the event. The eventID is unique in the cluster, and is increased by one automatically whenever a new event is generated. The EventID is used, along with the timestamp to distinguish between simultaneous events. Unfortunately, the eventID is lost because we don't save input events in our approach. Instead, we use the signal value on the wire to compensate for the lost eventID information. The new tuple for event annihilation is (LPID, timestamp, signalValue). If both the timestamp and the signalValue are the same for two events, they are considered to be identical events. If there exists more than one identical event in the event queue, the anti-event will pick the first one in the queue to annihilate. This approach introduces some indeterminism. However, Verilog[11] is a concurrent language, in which there are sources of non-deterministic behavior such as arbitrary execution order in zero time and arbitrary interleaving of behavior statements. Therefore, the simulation results are not guaranteed to be deterministic. In fact, simultaneous events are executed in arbitrary order in the Verilog simulator.

5.3 Port flag

We use a different rollback strategy for LPs inside the cluster and for LPs outside of the cluster. Inside the cluster, we roll back those LPs which are descendants of the LP which receives the straggler or anti-message. Anti-messages are sent to LPs outside of the cluster.

In order to implement the two rollback algorithms, a port flag is used for each LP port in order to indicate whether it is an internal port or an external port. The port flag is set at run time. Initially, every port flag is set to be an internal flag. When the LP receives an external message, it sets the corresponding flag to external. The implementation of the rollback algorithms making use of these flags will be explained in the following section.

5.4 Event builder

Only those events which change the input signals at a gate need to be reconstructed, as it is only these events which cause a change in the state of a gate.

Let s' be the state before the execution of event e and let s be the state after the event is executed. If s is equal to s', event e is considered null and need not be reconstructed. However, if s is different from s', event e can be rebuilt according to Formula 1. The signal value is the value on the wire, as shown in Table 1.

$$\begin{aligned} e.timestamp &= s.timestamp \\ e.signalValue &= s.signalValue \end{aligned} \quad (1)$$

For example, state s' is shown in Figure 5.a and state s in Figure 5.b. It is worthwhile noting that signal values on all ports are packed into a one byte state. By comparing the value on port I1 of states s and s', the event which happened at time 200 is reconstructed with the value on port I1 of state s, which changed from '00' in state s' to '01' in state s.

5.4.1 Input event builder

In Time Warp an LP saves events after processing them because if the LP rolls back, previously processed events will have to be reprocessed. Through event reconstruction, these previously processed events will no longer have to be saved. Instead, they are reconstructed by the input event builder, depicted in Figure 6.

The algorithm loops through the state queue until the LVT of the state is less than the receive time of the event

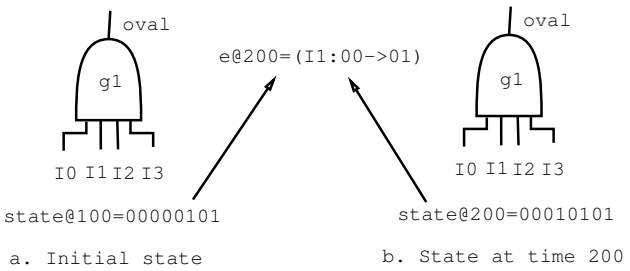

Figure 5. Event reconstruction

which causes the rollback. It picks a state s1 and its predecessor s2 from the state queue. If the input values of state s1 and s2 are different, an event e is reconstructed according to Formula 1. The input values are bound into one byte. Therefore, the comparison is executed four times, once for each input port of the LP, as shown in the Figure 6.

Moreover, due to the different rollback strategies for the internal events and external messages, we set a port flag to indicate the source of the events. For the external port, we reconstruct every event. However, we only reconstruct the events which have LVT equal to the LVT of the straggler event for the internal port. The reason for this is that the internal events which have a larger LVT than the straggler will be regenerated because of the cluster rollback strategy[1] used in DVS[9], which will rollback all LPs in the cluster. Therefore, the internal events will not be reconstructed because they will be regenerated by their source LP in the same cluster. The port flag is used to avoid unnecessary reconstruction of internal events. In fact, the algorithm of event reconstruction does not depend on the cluster rollback strategy. We are continuing to improve the rollback strategy and the event reconstruction algorithm. Further effort will focus on tree rollback instead of the cluster rollback. The tree rollback strategy only rolls back those LPs which reside in a tree whose root is the LP which receives the straggler event.

5.4.2 Anti-event builder

The anti-event builder works in the same way as the input event builder, as shown in figure 7. The anti-event is reconstructed by comparing the output values of two adjacent states, s1 and s2. After reconstruction, the anti-event is sent to those LPs which are in the fanout list of the current LP but not in the same cluster. For a cluster rollback, we don't have to use anti-events to cause a rollback in the same cluster.

5.5 Event processing loop

The basic algorithm for an optimistic LP is sketched in Figure 8. The LP removes the head event from the event

```
input_event_builder(event* rb_event)
{
  reverse_iterator iter=state_list.rbegin();

  //main loop for the event reconstruction
  while((*iter)->LVT >= rb_event->recv_time)
  {
    state* s1 = (*iter);
    state* s2 = (*iter++);

    //only reconstruct the external event
    //ignore the reconstruction of internal events

    //Event reconstruction on port 0
    if (s1->ival&3 != s2->ival&3)
      if (s1->LVT == rb_event->recv_time ||
          external_port_flag[0])

      {
        //reconstruct the event
        e->recv_time = s1->LVT;
        e->ival = s1->ival&3;

        if e->is_anti_event(rb_event)
          annihilate(e, rb_event);
        else
          schedule(e);
      }

    //Event reconstruction on port 1
    if ((s1->ival>>2)&3 != (s2->ival>>2)&3)
      if (s1->LVT == rb_event->recv_time ||
          external_port_flag[1])
      {
        //reconstruct the event
        e->recv_time = s1->LVT;
        e->ival = (s1->ival>>2)&3;

        if e->is_anti_event(rb_event)
          annihilate(e, rb_event);
        else
          schedule(e);
      }

    //Event construction on port 2 & port 3
    //compare((s1->ival>>4)&3, (s2->ival>>4)&3)
    //compare((s1->ival>>6)&3, (s2->ival>>6)&3)
  }
}
```

Figure 6. Input event reconstruction algorithm

queue and checks whether it is a normal event or a straggler or an anti- event. If it is a normal event, the LP first logs its state and processes the event. State is saved after every event. However, processed events will not be saved.

When the LP receives an anti-event or a straggler, it rolls back as in "normal" Time Warp. However, the LP reconstructs the input events and output events from the state queue. This introduces a processing overhead which is similar to the cost for coasting forward in dynamic checkpointing.

6 Experiments

All of our experiments were conducted on a network of 8 computers, each of which has dual PentiumIII processors and 256M RAM. They are interconnected by a Myrinet,

```
anti_event_builder(event* rb_event)
{
  reverse_iterator iter=state_list.rbegin();

  while((*iter)->LVT >= rb_event->recv_time)
  {
    state* s1 = (*iter);
    state* s2 = (*iter++);

    if (s1->oval&3 != s2->oval&3)
      {
        //reconstruct the anti event
        e->recv_time = s1->LVT;
        e->ival = s1->oval&3;
        e->flag = ANTI;

        for each external LP in fanout list
          send e to LP
      }
   }
}
```

Figure 7. Anti event reconstruction algorithm

```
while(GVT < FINISH_TIME)
{
  receive external events;
  pop an event from event queue;
  update LVT;
  if (event is straggler or antimessage)
    {
      input_event_builder();
      anti_event_builder();
      send_anti_events();
    }
  else
    {
      log_state();
      event_processing();
    }
}
```

Figure 8. Optimistic LP simulation algorithm

a high speed network with link capacity of 1Gbit per second. All machines run the FreeBSD operating system while MPICH-GM is used for message passing between different processors.

The Verilog source file used in the simulation describes an ISCAS'89 benchmark circuit, S38584. It includes 19253 gates, 1426 D-type flip-flops and one virtual thread which feed 20 random vectors to the circuit. The clock frequency of S38584 is 1MHz. The other Verilog source file describes a 16bit multiplier. It includes 2416 gates and one virtual thread which feeds 200 random vectors into the circuit.

We assume a unit gate delay and zero transmission delay on the wire. Each data point collected in the experiments is an average of five simulation runs. The number of machines in the figure doesn't include machine 0, which only contains vthreads[9]. The vthreads generate the events for the simulation. The simulation time for 1 machine is the running time of the DVS without partitioning.

circuit	*2*	*3*	*4*	*5*	*6*
16 bits multiplier	4.49	4.50	4.78	4.68	4.73
S38584	3.60	3.68	3.54	3.53	3.55

Table 2. The memory usage ratio

In the experiments, we compare the performance of DVS with dynamic checkpointing and with event reconstruction to that of "pure" Time Warp. The partitioning algorithm which we use is CAKE[6]. The dynamic checkpointing algorithm is initiated every 1000 events. Our event reconstruction algorithm requires that the state is saved after each event is processed.

6.1 Memory Usage

6.1.1 Memory consumption breakdown

The memory consumed by Time Warp is composed of the memory consumed by state saving and by event saving. Figure 9 presents the memory breakdown for the machine which has the maximum memory consumption. The data is collected for a 16 bit multiplier and for S38584 using Time Warp. The top of Figure 9 is the memory breakdown for the 16 bit multiplier with 200 random vectors while the bottom is the memory breakdown for S38584 with 30 random vectors.

We see from both of these that event saving consumes more memory than state saving. We define the *m*emory usage ratio to be the ratio of the memory consumed by event saving to the memory consumed by state saving and list these ratios in table 2 for the 16 bit multiplier and for S38584. We see that event saving consumes 4.73 times the memory used by state saving when 6 machines are used. On the average, *e*vent saving consumes almost four times the memory consumed by state saving.

6.1.2 Peak memory consumption

We define the *p*eak memory usage to be the maximum of all of the machines' maximal memory usages. Figure 10 shows the peak memory vs. the number of machines for the 16 bit multiplier. The memory used by one machine is only 0.45M because memory overhead is unnecessary. When two machines are used, event reconstruction uses 1.79 times less memory than dynamic checkpointing and 2.34 times less than pure Time Warp. The ratio between event reconstruction and dynamic checkpointing decreases when more machines are used. The reason for this decrease is that the average number of events processed decreases when more machines are used, and consequently the memory occupied by event saving decreases. When 6 machines are used, event

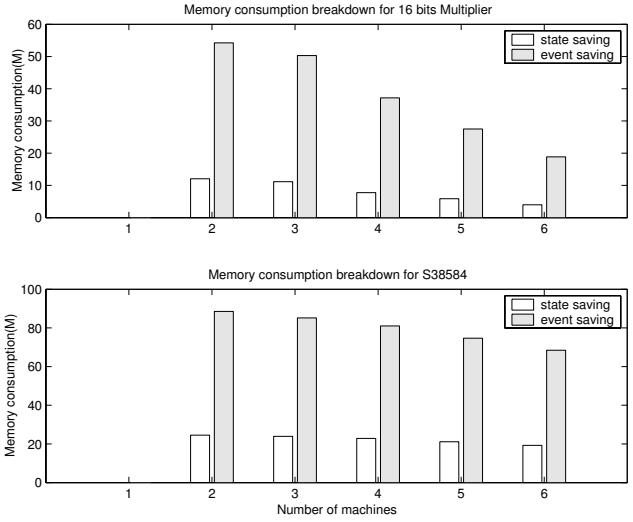

Figure 9. Memory consumption breakdown

reconstruction uses 1.29 times less memory then is used by dynamic checkpointing.

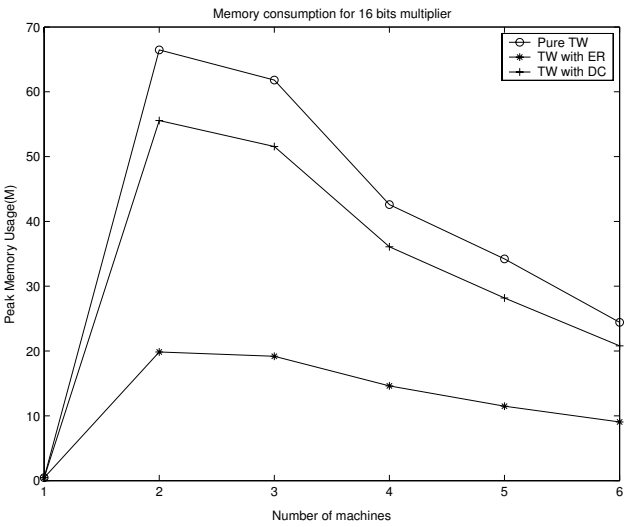

Figure 10. Memory consumption for 16 bits multiplier

Figure 11 presents the peak memory vs. the number of machines for S38584. Time Warp uses 119.06M when 2 machines are used. This leads to memory swapping and bad performance, as shown in Figure 13. Event reconstruction uses 54.21M when two machines are used, versus 99.18M by dynamic checkpointing.

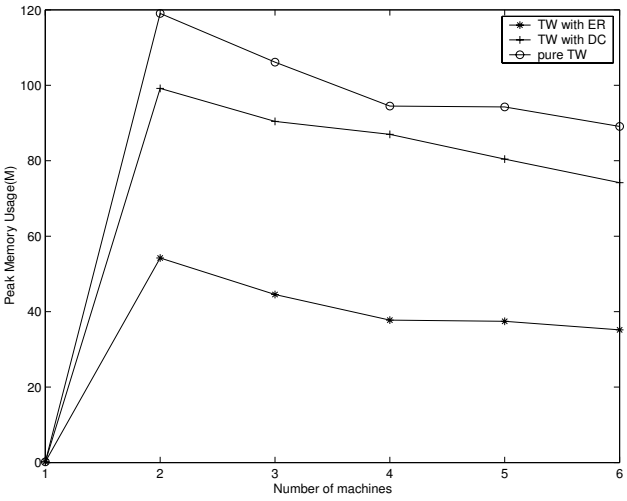

Figure 11. Memory consumption for S38584

6.2 Simulation Time

The simulation time vs. the number of machines for the 16 bit multiplier is presented in figure 12.

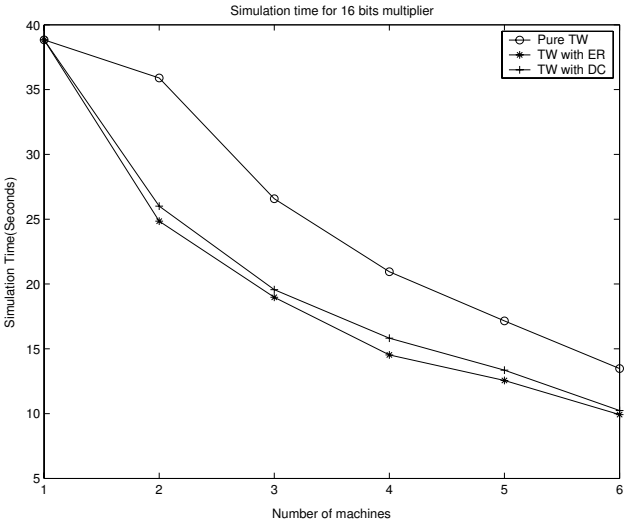

Figure 12. Simulation Time for 16 bits multiplier

We observe from figure 12 that event reconstruction results in a 10% execution time improvement over dynamic checkpointing and a 40% improvement over Time Warp when 2 machines are used. The speedup decreases when more machines are used for the same reason that the improvement in memory consumption diminishes when more machines are used. The speedup obtained using event reconstruction is 3% better then dynamic checkpointing and

35% better then Time Warp when 6 machines are used.

Figure 13 presents the simulation time vs. the number of machines for S38584. The simulation time of pure Time Warp is 25.55 because of memory swapping. Both dynamic checkpointing and event reconstruction eliminate memory swapping. However, event reconstruction is 11% faster than dynamic checkpointing.

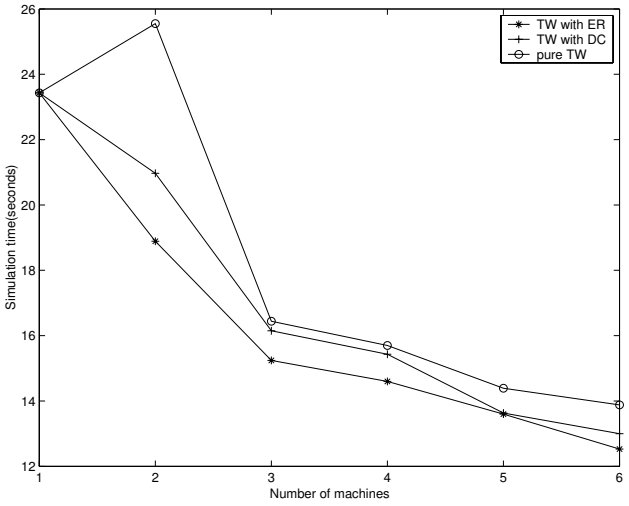

Figure 13. Simulation Time for S38584

7 Conclusions and work in progress

Time Warp is known to use a great deal of memory. In this paper, we observe that event saving is a significant factor in memory consumption for optimistic VLSI simulation. Our proposed technique of event reconstruction eliminates event saving by reconstructing events from the state queue. For simulations with fine event granularity and small state size such as the VLSI logic simulation, event reconstruction yields a significant reduction in memory utilization compared to checkpointing. This reduction leads to a faster execution time. Finally, event reconstruction facilitates load balancing because only the state queue of an LP needs to be moved between processors. We plan to implement dynamic load balancing in DVS[9].

References

[1] Herve Avril and Carl Tropper. Scalable clustered time warp and logic simulation. *VLSI design*, 00:1–23, 1998.

[2] M. Bailey, J. Briner, and R. Chamberlain. Parallel logic simulation of vlsi systems. *ACM Computing Surveys*, 26(03):255–295, Sept. 1994.

[3] H. Bauer and Sporrer C. Reducing rollback overhead in time-warp based distributed simulation with optimized incremental state saving. In *Proc. of the 26th Annual Simulation Symposium*, pages 12–20. Socity for Computer Simulation, April 1993.

[4] S. Bellenot. State skipping performance with the time warp operating system. In *6th Workshop on Parallel and Distributed Simulation*, pages 53–61. Socity for Computer Simulation, January 1992.

[5] Christopher D. Carothers, Kalyan S. Perumalla, and Richard Fujimoto. Efficient optimistic parallel simulations using reverse computation. In *Workshop on Parallel and Distributed Simulation*, pages 126–135, 1999.

[6] Hai Huang. A partitioning framework for distributed verilog simulation. Master's thesis, School of Computer Science, McGill University, 2003.

[7] P.A. Wilsey J. Fleischmann. Comparative analysis of periodic state saving techniques in time warp simulators. In *Ninth Workshop on Parallel and Distributed Simulation (PADS'95)*, pages 50–58, June 1995.

[8] D. Jefferson. Virtual time. *ACM Transactions on Programming Lauguages and Systems*, 7(3):405–425, 1985.

[9] Lijun Li, Hai Huang, and Carl Tropper. Dvs: an object-oriented framework for distributed verilog simulation. In *Parallel and Distributed Simulation, 2003. (PADS 2003)*, pages 173–180, June 2003.

[10] Yi-Bing Lin, Bruno R. Preiss, Wayne M. Loucks, and Edward D. Lazowska. Selecting the checkpoint interval in time warp parallel simulation. In *Proc. 1993 Workshop on Parallel and Distributed Simulation*, pages 3–10. Institute of Electrical and Electronics Engineers, May 1993.

[11] Donald E. Thomas and Philip R. Moorby. *The Verilog Hardware Description Language Fourth Edition*. KLUWER Academic Publisher, 1992.

[12] Carl Tropper. Parallel Discrete-Event Simulation Applications. *Journal of Parallel and Distributed Computing*, 62:327–335, 2002.

[13] P. Wilsey and A. Palaniswamy. Rollback relaxation: A technique for reducing rollback costs in an optimistically synchronized simulation. In *International Conference on Simulation and Hardware Description Languages, Society for Computer Simulation*, pages 143–148, January 1994.