DVS: An Object-Oriented Framework for Distributed Verilog Simulation

Lijun Li, Hai Huang and Carl Tropper
School of Computer Science
McGill University
Montreal, Canada
lli22, hhuang17, carl@cs.mcgill.ca

Abstract

There is a wide-spread usage of hardware design languages(HDL) to speed up the time-to-market for the design of modern digital systems. Verification engineers can simulate hardware in order to verify its performance and correctness with help of an HDL. However, simulation can't keep pace with the growth in size and complexity of circuits and has become a bottleneck of the design process. Distributed HDL simulation on a cluster of workstations has the potential to provide a cost-effective solution to this problem.

In this paper, we describe the design and implementation of DVS, an object-oriented framework for distributed Verilog simulation. Verilog is an HDL which sees wide industrial use. DVS is an outgrowth of Clustered Time Warp, originally developed for logic simulation. The design of the framework emphasizes simplicity and extensibility and aims to accommodate experiments involving partitioning and dynamic load balancing. Preliminary results obtained by simulating a 16bit multiplier are presented.

1 Introduction

Moore's Law states that computational power will roughly double every 18 months. To the semiconductor designer, this means a never-ending challenge in bringing increasingly larger and more complex IC(Integrated Circuit) to market.

The complexity and size of digital systems described by Verilog continues to grow. The introduction of the system-on-chip(SoC), which contains CPUs, memory and analog circuitry on a single chip has only served to exacerbate this problem. The SoC is intended for use in embedded systems. Sequential Verilog simulators, or even specialized hardware accelerators, cannot keep up with this pace, and has become a bottleneck of the design process. To accommodate the growing need for increased memory demands as well as the need for decreased simulation time, it is necessary to make use of distributed and parallel computer systems[18]. Networks of workstations provide a cost-effective environment for distributed simulation.

Verilog and VHDL are both important VLSI design languages. However, research efforts to date have focused on distributed VHDL simulators[7, 12, 13]. This paper presents a description of our research to date on a distributed Verilog simulation framework. To the best of our knowledge, ours is the first distributed Verilog simulator.

Writing a Verilog compiler represents a substantial commitment. Consequently, we made use of Icarus Verilog, an open source Verilog compiler and simulator. We also re-designed Clustered Time Warp[2] and used it as our back-end simulation environment.

The rest of this paper is organized as follows. In section 2 we (briefly) introduce PDES, the Icarus Verilog compiler and VVP(Verilog Virtual Processor) simulator. In section 3 we detail our effort to design and implement DVS(Distributed Verilog Simulator), a distributed Verilog simulation framework. Preliminary results obtained by simulating a 16bit multiplier are presented in section 4, while the last section contains our conclusions.

2 Background and Related work

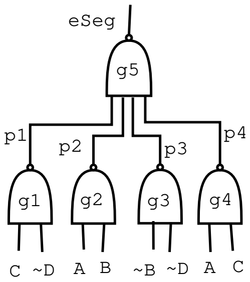
2.1 PDES

A distributed simulation system is composed of processes which communicate with each other via message passing. Each process simulates a portion of the physical system and is referred to as a logical process(LP). During the simulation, LPs create events, send events to other LPs and receive events from others LPs. Two families of synchronization algorithms are widely used in order to maintain causality in a distributed simulation, known as the conservative and optimistic algorithms.

Conservative algorithms[6] are characterized by blocking behavior. An LP blocks until it has a safe event to process.

An example of an optimistic algorithm is Time Warp[9]. In Time Warp, LPs maintain an input queue which contains events received from other LPs, an output queue which stores copies of events sent to other LPs and a state list which stores the LPs state at checkpoint. Time Warp allows a local causality violation but uses rollback and anti-messages to correct possibly erroneous computation. Time Warp can suffer from cascading rollbacks and excessive memory consumption.

2.2 Verilog




```
//structural description
Module binaryToESeg;
   wire eSeg, p1, p2, p3, p4;
   reg A, B, C, D;

   nand #1
     g1(p1, C, ~D),
     g2(p2, A, B),
     g3(p3, ~b, ~D),
     g4(p4, A, C),
     g5(eSeg, p1, p2, p3, p4);
endmodule

//behavioral description
Module binaryToESeg;
   wire eSeg, p1, p2, p3, p4;
   reg A, B, C, D;

   always @(A or B or C or D)
   begin
     eSeg = 1;
     if(~A&D) eSeg = 0;
     if (~A&B&~C) eSeg = 0;
     if (~B&~C&D) eSeg = 0;
   end
endmodule
```

Figure 1. Structural and behavioral description of Verilog

The Verilog Hardware Description Language is standardized in IEEE standard #1364-1995. It supports both a behavioral description and a structural description of a digital system. Figure 1 shows an example of how Verilog describes an IC design[17]. It is part of a binary to seven segment display driver.

The Verilog language describes a digital system as a set of modules. Each module has an interface to other modules and represents a logical unit in a structural description (such as an interconnection of gates) or a behavioral description which is similar to a programming language.

Verilog is designed to allow concurrent execution. A digital system can be conceived of as a set of concurrent processes found in initial blocks, always blocks and continuous assignments. Wait and event control statements can be used to synchronize two concurrent processes. The existence of concurrent processes in Verilog indicates that it is suitable for distributed simulation[4]. A comprehensive description of Verilog can be found in [17].

2.3 Overview of Icarus Verilog

Icarus Verilog [19] is an open-source EDA (Electronic Design Automation) Verilog simulator being developed by Stephen Williams. As shown in figure 2, Icarus Verilog includes two independent parts: an IVerilog compiler and a VVP(Verilog Virtual Processor) simulator. The bridge connecting these two parts is VVP assembly code, an intermediate representation of the original circuit. The IVerilog compiler is a translator that translates the input Verilog source code into VVP assembly code. The VVP simulator is an event-driven simulation engine, which interprets VVP assembly code and process the events. We give a brief introduction to Icarus Verilog in the following sections.

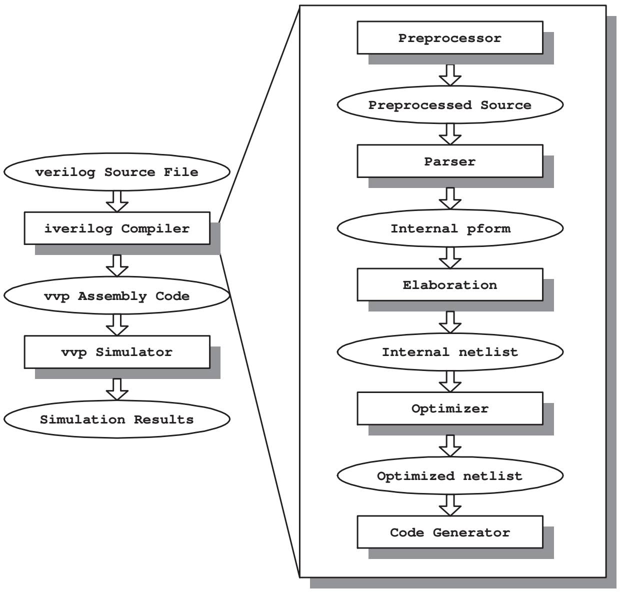


Figure 2. Architecture for Icarus Verilog

2.3.1 IVerilog Compiler

Although the Verilog language enhances modularity and encapsulation by the use of modules in the source file of a circuit, the hierarchical structure of modules is not appropriate for the purpose of simulation. The IVerilog compiler flattens modules in the original source file in the following five consecutive phases:

- Preprocessor. It mainly performs file inclusion for 'include directive and macro substitution for 'define directive.

- Parser. The preprocessed source file is parsed and its internal representation is generated with syntax and semantic checking performed.

- Elaboration. The root module is located, unresolved references are resolved, and all instantiations of modules are expanded. After scope elaboration and netlist elaboration, an internal flattened netlist is generated from the hierarchically structured modules.

- Optimizer. Some useful transformations can be performed on the internal netlist in order to simplify netlist and improve simulation efficiency.

- Code generator. All circuit information is now stored in the flattened and optimized internal netlist. There are five target formats that can be generated from the netlist, of which VVP assembly code is the default one used for simulation.

2.3.2 VVP Simulator

The VVP simulator is an interpreter for VVP assembly code. It parses VVP assembly code to generate netlist of structural items and exert input vectors to drive the simulation.

The separation of the IVerilog compiler and the VVP simulator is similar to the separation of compiler and interpreter in Java. The VVP assembly code is the counterpart of bytecode in Java. Since large VLSI circuit files normally take a long time for compilation, this strategy saves a lot of time. Once the VVP assembly code file is generated by the IVerilog compiler, we can use it in our partitioner and distributed simulator.

3 Design and implementation of the simulation framework

In this section, we explain our effort to design and implement DVS, an object-oriented framework for distributed Verilog simulation.

3.1 Architecture

Figure 3 illustrates the architecture of DVS. It takes VVP assembly code as input, which is generated by the IVerilog compiler for simulation efficiency. The VVP parser constructs the functor list and virtual thread list, which will be used by the distributed simulation engine after partitioning.

The 3 layers of DVS are shown in the right side of figure 3. The bottom layer is the communication layer which provides a common message parsing interface to the upper layer. Inside this layer, the software communication platform can be PVM or MPI. Users can choose one of them without touching the code of upper layer.

The middle layer is a parallel discrete event simulation kernel, OOCTW, which is an object-oriented version of Avril's CTW(Clustered Time Warp)[2]. It provides services such as rollback, state saving and restoring, 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 virtual thread.

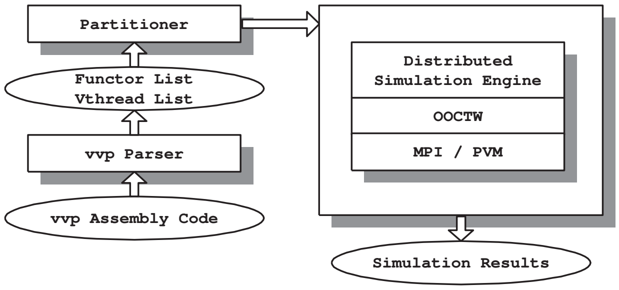


Figure 3. Architecture of DVS

3.2 VVP parser

The Verilog language provides the ability to model a circuit by means of both structural descriptions and behavioral descriptions. Structural descriptions model the circuit as a network of interconnecting gates and wires, while behavioral descriptions model the circuit at a higher level as *always* and *initial* blocks. They are translated to *.functor* statement and *.thread* statement in the VVP assembly code generated by the IVerilog compiler. The VVP parser parses VVP assembly code and instantiates these structural and behavioral statements as functors and vthreads which are described in the following sections.

3.2.1 Structural item: functor

Structural items are represented by functors in the VVP simulator. Each functor has four input ports and one output port. Gates with more than four input ports are divided into smaller gates and cascaded. Functors also have associated delay values. All functors are stored in a functor list which will be used for partitioning and simulation.

During the simulation, when the value in any input port of a functor changes, a new output value is calculated by querying a truth table. If the result is different from the current value in the output port, the value in the output port is updated, and a propagation event is scheduled with the associated delay value. After this delay time expires, the propagation event is processed, and the signal is assigned to corresponding input ports of all fanout functors.

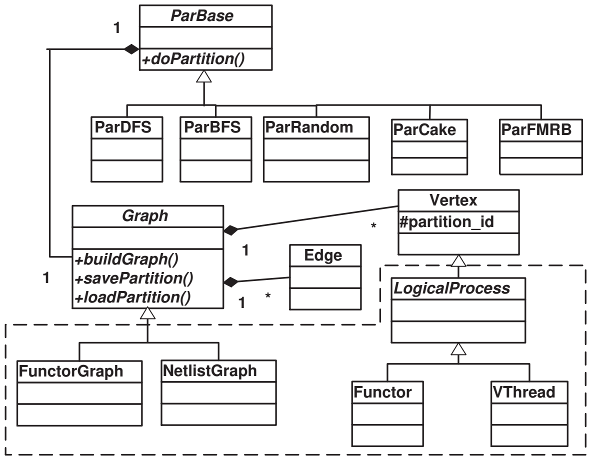


Figure 4. UML description of partitioner

3.2.2 Behavioral item: vthread

Behavioral items are represented by virtual threads (vthread) in the VVP simulator. It should be noted that vthreads run in the virtual machine of the VVP simulator instead of running directly in the operating system. Each vthread contains a mechanism for thread execution, including a program counter, 4 numeric index registers and 64k private bit registers.

All vthreads instantiated by the VVP parser are organized as a vthread list. In gate-level logic simulation, vthreads are normally used to drive functors with input vectors.

3.3 Partitioner

Partitioning plays an important role in affecting the performance of parallel logic simulation[3]. In order to exploit different partitioning algorithms in DVS, we designed a generic partitioner and integrated it into the framework of DVS.

3.3.1 Design of Partitioner

The design goal of our partitioner is to provide a flexible infrastructure for testing different partitioning algorithms applied to different circuit implementations. As shown in figure 4, the partitioner has two major parts: the partitioning algorithm and the circuit graph being partitioned.

The circuit graph is represented by Vertex and Edge objects in the abstract Graph class. The Graph class also provides interfaces to partitioning algorithms for retrieving information for vertices and edges in a graph. Designers of different simulators can subclass it and implement the buildGraph method to fill in vertices and edges using application-specific information. In DVS, we use FunctorGraph to build the graph using the functor list.

The base class for partitioning algorithms, ParBase, is also an abstract class. All partitioning algorithms should be derived from ParBase and provides an algorithm-specific implementation for the doPartition method. In DVS, the partitioner will automatically select the corresponding algorithm at run time based on the partitioning argument in the command line.

3.3.2 Partitioning functors and vthreads

Since circuit information is available in both the IVerilog compiler and the VVP simulator, we can perform partitioning on either side. After investigating the internal data structures on both sides, and also considering that both functors and vthreads are LPs in DVS, we decide to use the functor list and vthread list in our partitioning algorithm.

The structure of the functor list is similar to an adjacency list, which is convenient for partitioning. Furthermore, since every computer in the simulation has the same copy of functor list, it can be readily used for message routing when the destination functor resides on remote computer. If dynamic load balancing is performed during the simulation, the re-partitioning can be done on the functor list, and the re-mapping of functors is as simple as modifying the partition-id of corresponding functors.

The treatment of vthreads is different from functors. We observe that when functors and vthreads are placed in the same partition, more rollbacks tend to occur. OOCTW uses clustered rollback, i.e., a straggler at one LP causes all LPs in the same cluster to rollback. Vthreads tends to advance much faster than functors in LVT because behavioral simulation is more efficient than logic simulation. Thus a fast vthread is likely to cause all of the slow functors in the same cluster to rollback more frequently. Therefore, we put all of the vthreads on one computer. Since the total number of vthreads is small in gate-level logic simulation, the lost concurrency can be compensated for by fewer rollbacks. The large number of functors are partitioned and assigned to the rest of the computers in the simulation.

3.3.3 Partitioning metrics and direction of ongoing research

The graph partitioning problem is NP-complete; therefore most partitioning algorithms are based on heuristics. A comprehensive survey of netlist partitioning can be found in [1]. Empirical studies [3, 2, 11, 16] show that there are three major factors that determine the quality of a partition: load balance, communication and concurrency. The goal of a partitioning algorithm is to maintain load balance, minimize communication and exploit concurrency. The optimal partition is the one that finds the best tradeoff among these three factors.

CLIP/CDIP[15] and Metis/hMetis[10, 8] are state-of-art algorithms for large circuit partitioning, both of which are fast and result in a small cutsize. However, they only consider communication and load balance. Concurrency is not addressed in either CLIP or Metis. We aim to introduce concurrency into CLIP or Metis in our ongoing research. In order to achieve load balancing, pre-simulation[5] can also be used to get an accurate activity level for each gate.

3.4 OOCTW(Object-oriented CTW)

3.4.1 Motivation

Clustered Time Warp(CTW)[2] was developed with logic simulation in mind. LPs (representing gates) are grouped into clusters. Each cluster has an input and an output queue associated with it. Events were executed sequentially within the cluster. Several rollback and checkpoint algorithms were developed for use with CTW.

CTW is a good starting point for the implementation of object-oriented Time Warp. A cluster bundles gates together in order to overcome the fine event granularity of VLSI simulation. Furthermore, a cluster provides a very good basis for load balancing. We can also move an entire cluster between processes instead of just moving gates. However, CTW is not object-oriented. It is not easy to integrate it directly with the sequential simulator. Therefore, we used an object-oriented paradigm to transform CTW into OOCTW, which (we hope) will be an open and flexible synchronization backend.

The main design goal of OOCTW is to integrate it with the original Verilog simulator. The motivation for the design is to limit the changes made to the sequential simulator because we hope to take advantage of its new version. The other design goal is to make the Time Warp library more reusable, readable and understandable so new members in the laboratory can concentrate on the optimization algorithms instead of falling into the black hole of Time Warp. Finally, the Time Warp library must be flexible and open so it can be a test bed for new optimization algorithms.

To date we have only implemented one of the rollback algorithms developed for CTW, clustered rollback, in OOCTW. In clustered rollback, when a straggler or an antimessage arrives at the cluster, all of the LPs with larger LVTs than the straggler or the antimessage are rolled back. Other modifications of CTW are checkpointing when the LVT of an LP advances and the use of Mattern's GVT algorithm[14].

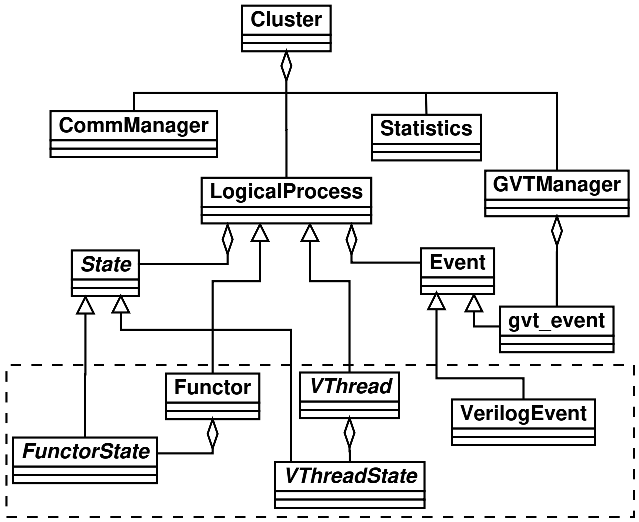


Figure 5. UML description of OOCTW

3.4.2 Class hierarchy of OOCTW

The diagram above the dashed rectangle in figure 5 is a UML description of OOCTW. The Cluster is the container and scheduler of all LPs. The scheduling algorithm we employed is LTSF(Lowest Timestamp First). An LP is scheduled for execution when it has an event with the lowest timestamp in the cluster. The cluster manages a future event list and an output event list. The GVT computation is also processed in the cluster. Each time the cluster receives a new GVT, it invokes fossil collection. Statistics are also collected in the Cluster such as simulation time, rollback number, communication cost, etc.

As shown in figure 5, class LP executes rollback and provides virtual methods for state saving and state restoration. The derived classes override the virtual methods to have application-specific implementations of state saving and restoration. An LP maintains a processed event list but doesn't maintain an output event list. When an LP sends out an event which crosses the cluster boundaries, it inserts a copy of the event into the output event list of the cluster.

The event class provides operators such as $\ll$, $\gg$ and == to compare the timestamp of two events. The procedures to decide whether an event is a negative event are also provided in the class. Class gvt_event inherits from event class. It is used to compute the GVT via Mattern's algorithm[14].

The base class for state is an abstract base class. It provides an interface for the application specific state. In DVS, there are two different kinds of LPs with their own state, which will be explained in detail in the following section.

Type	*Usage*
THREAD	Schedule a virtual thread
EVAL	Evaluate the functor
PROP	Propagate the value change after gate delay
INQUIRY	Inquiry value of a remote functor
RESPOND	Respond inquiry of functor value
FINISH	Finish of the simulation

Table 1. Events in distributed Verilog simulation engine

3.5 Distributed Simulation Engine

The original sequential VVP simulator is turned into a distributed simulation engine via its integration with OOCTW. The classes in the distributed simulation engine are shown in the dashed rectangle of figure 5. *Functor* defines structural items in Verilog while *Vthread* defines behavioral blocks. They both inherit from class LP and override the abstract member methods so they are able to save state, rollback and restore state. *FunctorState* and *VthreadState* implement the interface of *state*, which is used to log the state of the functors and vthreads.

VerilogEvent inherits from class *event*. Several types of events in the distributed simulation engine are shown in table 1.

THREAD event is used to awake the blocked virtual thread which is waiting for an event to happen, such as a value change of a register. *EVAL* and *PROP* are used to propagate value changes among the network of functors.

INQUIRY event is used to detect the value of a functor located in a remote host. For example, the variable 'a' in statement *$display($time,,a)* may be located in a remote processor. Therefore, the virtual thread will send an *INQUIRY* message to get the value of the remote functor. The remote processor will send back the response as soon as it processes the event.

After partitioning, the simulator schedules the *THREAD* event to invoke the virtual threads whose partition ID matches the host id of the local machine. These virtual threads will feeds input vectors to the network of functors. The simulator keeps processing events until it gets FINISH event broadcasted by machine 0.

Each simulator in different machines keeps the topology of all of the functors in order to route messages. However, only those functors with the same ID as the local host are active. The passive functors are only used to route messages. No evaluation happens on passive functors.

The $display and $monitor in Verilog are used to print values of variables or logic gates. However, the state of an LP is not stable until its LVT is smaller than GVT. Therefore, I/O can't be committed immediately after the instruction is issued. Hence, we created a delayed I/O instruction list to save all I/O instructions and the time at which they are issued. Each time a new GVT is generated, the simulator will check the delayed I/O list. If the timestamp of the I/O instruction is smaller than GVT, it will be committed.

3.6 Optimization to distributed Verilog simulation engine

- Direct execution of zero delay event

  When the simulator generates a zero delay event which has the same timestamp as the current LVT, it executes the event directly without inserting it into the event queue then popping it out and executing it. This introduces some indeterminism but doesn't affect the final simulation result. The direct execution reduces memory operations and speeds up the simulation.

  In fact, there are a lot of simultaneous events in the Verilog simulation. A functor will propagate its value change to all of its fanout functors. All propagated evaluation events are simultaneous events which have the same timestamp as the current LVT because we assume zero wire delay. If the fanout functors resides on the same cluster, the Verilog simulator can execute the corresponding evaluation events directly.

- on-the-fly fossil collection

  In order to improve the efficiency of the simulator, the designer of Icarus simulator maintains a free event list in order to minimize the invocation of the system calls such as malloc/free and new/delete. Each time the simulator schedules a new event, it first checks the free list. If it is not empty, the new event can directly use the memory space occupied by the head of the free list. When the simulator finishes processing the event, it puts the event pointer into the free list instead of deleting the memory space.

  The free list is inherited in the distributed simulator. Moreover, we created the free state list for state saving of LPs.

4 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(www.myri.com), a high speed network with link capacity of 1Gbit per second. All machines run the FreeBSD operating system. LAM MPI is used for message passing between different processors.

The Verilog source file used in the simulation describes a 16bit multiplier. It includes 2416 gates and one virtual thread which feed 50 random vectors to the circuit. We assume the unit gate delay and zero transmission delay on the wire. Only the simulation results with BFS partitioning is presented because we have compared the performance of BFS, DFS and random partitioning algorithm and found BFS to have the best performance. BFS partitioning algorithm can reduce communication, which is the most expensive operation in distributed environment. Each data point collected in the experiments is an average of five consecutive simulation runs. The number of machines in the figure doesn't include machine 0 which only contains vthreads. The simulation time for 1 machine is the running time of the DVS without partitioning.

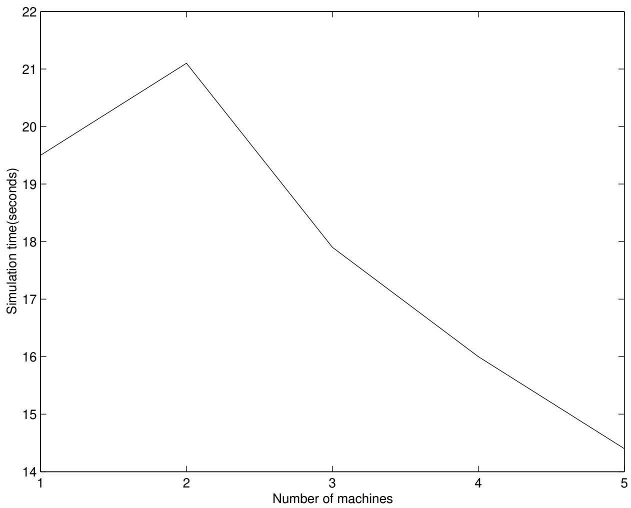


Figure 6. Simulation time in seconds vs. number of machines

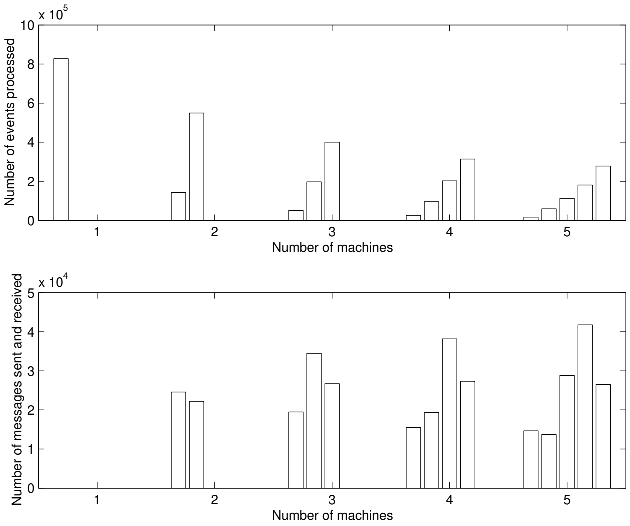


Figure 7. Number of events processed by every machine(Upper part) and number of messages sent and received(Lower part) by every machine vs. number of machines. Note: The two figures use different scale.

Operation	*Time*
Processing an event	1.83us
Saving a state	2.08us
Saving an event	2.56us
Sending a message(Blocking)	31.9us
Receiving a message(Blocking)	32.2us
Message latency	10us

Table 2. Cost of operations in DVS

The simulation time vs. the number of machines is shown in figure 6. It should be noticed that the simulation time is longer when 2 machines are used. This is caused by the load imbalance and communication cost. From the upper part in figure 7, we know that the partitioning algorithm only reduces the total number of event processed on machine 1 by a small amount when 2 machines are used. However, the communication cost increases by a large amount. The total communication cost can be computed by multiplying the number messages shown in the lower part of figure 7 with average sending/receiving cost, which is listed in table 2. The reduction in workload is not large enough to compensate the communication cost. Therefore, the total simulation time for 2 machines is longer than the time for 1 machine.

Using more machines reduces the number of events processed per machine a great deal, thus the time used to process events is reduced by the amount which is large enough to compensate the communication cost involved in the distributed simulation. The simulation times keep decreasing when the number of machines increases from 3 to 5. We get a speedup of 1.4 when 5 machines are used.

Unfortunately, so far DVS still runs slower than the original Icarus Verilog simulator. We attribute this to the fine granularity of VLSI simulation, large communication cost, load imbalance and the small circuit size of our Verilog source file. From table 2, we know that overhead for VLSI simulation is more than 2 times the cost of processing an event.

By increasing the event granularity, reducing communication costs and achieving load balance, we look forward to outperforming the original simulator in further experiments(in which we simulate larger circuits) and demonstrating the scalability of DVS as well.

5 Conclusions and work in progress

We have succeeded in creating DVS, an object-oriented framework for distributed Verilog simulation. It employs OOCTW as the synchronization backend and takes advantage of the open source code of Icarus Verilog simulator. It is designed to be flexible for future extension and optimization.

To our knowledge, DVS is the first distributed Verilog simulator. Previous research in distributed HDL simulation has been focused on VHDL[7, 12, 13].

The performance of DVS in our preliminary experiments is promising. Many optimizations can be applied to make it more efficient. Certainly, larger circuits will lead to better speedup, and will hopefully be able to demonstrate the scalability of DVS.

From our experiments, it is clear that an effective load-balancing and/or partitioning algorithm contains the key to the success of VLSI simulation. We intend to focus our effort in this direction, as previously discussed.

References

[1] C. J. Alpert and A. B. Kahng. Recent directions in netlist partitioning: A survey. *Integr. VLSI Journal*, 19(1-2):1–81, Aug 1995.

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

[3] M. L. Briner Bailey and Chamberlain. Parallel logic simulation of vlsi systems. In *ACM Computing Surveys*, volume 26, pages 255–294, Sept 1994.

[4] Prithviraj Banerjee. *Parallel Algorithms for VLSI Computer Aided Design*. Prentice Hall, Inc., 1994.

[5] R. D. Chamberlain and C. D. Henderson. Evaluation the use of pre-simulation in vlsi circuit partitioning. In *Proceedings of the 8th Workshop on Parallel and Distributed Simulation(PADS'94)*, 1994.

[6] K.M. Chandy and J. Misra. Asynchronous distributed simulation via a sequence of parallel computattions. *Communications of the ACM*, 24(11):198–206, November 1981.

[7] Radharamanan Radhakrihnan Dale E. Martin and Philip Wilsey. Analysis and simulation of mixed-technology vlsi systems. *Journal of Parallel and Distributed Computing*, 62:468–493, 2002.

[8] Vipin Kumar George Karypis, Rajat Aggarwal and Shashi Shekhar. Multilevel hypergraph partitioning: Applications in vlsi domain. In *ACM/IEEE Design Automation Conference*, pages 526–529, 1997.

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

[10] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. Technical Report TR 95-035, Department of Computer Science, University of Minnesota, Minneapolis, MN, 1995.

[11] H. K. Kim and J. Jean. Concurrency preserving partitioning(cpp) for parallel logic simulation. In *10th Workshop on parallel and distributed simulation(PADS'95)*, pages 98–105, May 1996.

[12] V. Krishnaswamy and P. Banerjee. Design and implementation of an actor based parallel vhdl simulator. In *9th Workshop on parallel and distributed simulation(PADS'95)*, pages 135–143, 1995.

[13] D. Lungeanu and C.-J.R. Shi. Parallel and distributed vhdl simulation. In *Proc. DATE*, pages 658–662, March 2000.

[14] F. Mattern. Efficient algorithms for distributed snapshots and global virtual time approximation. *Journal of Parallel and Distributed Computing*, 18(4):423–434, 1993.

[15] Wenyong Deng Shantanu Dutt. Cluster-aware iterative improvement techniques for partitioning large vlsi circuits. *ACM Transactions on Design Automation of Electronic Systems(TODAES)*, 7(1):91–121, Jan 2002.

[16] Swaminathan Subramanian, Dhananjai M. Rao, and Philip A. Wilsey. Applying multilevel partitioning to parallel logic simulation. In *Parallel and Distributed Computing Practices*, volume 4, pages 37–59, March 2001.

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

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

[19] Stephen Williams. *Icarus Verilog*. http://icarus.com/eda/verilog.