

Towards Distributed Verilog Simulation

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

Keywords: Distributed Verilog Simulation, TimeWarp, Partitioning

1 Introduction

The complexity and size of digital systems described by Verilog continues to grow. The (US) National Technology Roadmap for Semiconductors predicts that by 2005 leading edge designs will exceed 200M transistors operating at across the chip frequencies of 1 GHz. The introduction of the system-on-chip(SoC) used in embedded systems, which contains CPUs, memory and analog circuitry on a single chip has only served to exacerbate this problem.

The use of simulations, both digital and continuous time, will play a key role in their design. Simulators are used for both functional verification and timing assessment at all stages of a design. However, with the increasing size and complexity of SoCs, the present simulation tools have reached their limits. Fortunately, the emergence of parallel and distributed discrete event simulation (PDES) over the last decade may well prove to be a cost-effective solution to this problem, as SoC simulations will be able to execute on a network of workstations, instead of being restricted to a single processor. Significant speed-ups can be obtained relative to single-processor simulations.

Verilog and VHDL are both important VLSI design languages. However, research efforts to date have focused on distributed VHDL simulators[10, 18, 19]. 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 redesigned Clustered Time Warp[4] and used it as our back-end simulation environment.

In the next section we 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. A distributed simulation is correct if every LP processes events in non-

decreasing timestamp order. 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[7] are characterized by blocking behavior. An LP blocks until it has a safe event to process. An event is safe to process if no events with a lower timestamp than this event will be received. Blocking behavior and deadlocks are drawbacks to conservative methods.

An example of an optimistic algorithm is Time Warp[15]. 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 Distributed VHDL simulators

A number of recent efforts have centered around developing distributed VHDL simulators, and more recently on mixed-mode(analog/discrete) simulators because of the emergence of system on a chip designs. [24] describe a mixed-mode simulator based on a conservative paradigm. In [18] the authors present an object-oriented TimeWarp VHDL simulator based on the actor paradigm. Using string partitioning on relatively small multiplier circuits, the authors obtained scalable execution times on a SPARC server 1000 and on an Intel Paragon. In [10] the authors describe a simulation environment for both discrete-event models in VHDL and mixed(discrete-event/analog) models in VHDL-AMS. Experiments for several partitioning algorithms, including a multi-level algorithm developed by the authors [11] were reported, with the multi-level algorithm resulting in the fastest simulation times. Overheads for the digital portion of the circuit were not reported. [8] describes a mixed-mode simulator as well in which the digital portion of the circuit is described in VHDL, while the analog portion is described in SPICE. The simulation made use of optimistic synchronization for digital components and conservative synchronization for analog components. A speed-up of 6.4 was obtained for a circuit described in VHDL and SPICE. In [9], a combination of a conservative(for synchronous components) and optimistic(for asynchronous components) synchronization mechanisms was used for VHDL simulation. Linear speed-ups were observed for a range of small to medium size circuits.

2.3 Verilog

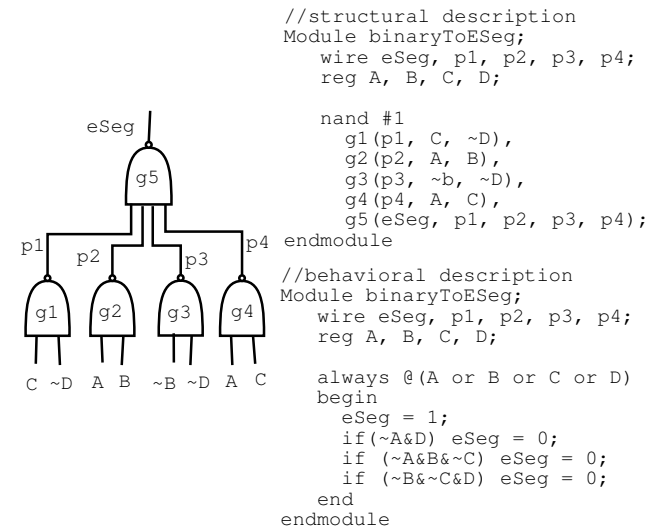


Figure 1: Structural and behavioral description of Verilog

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[23]. 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[6]. A comprehensive description of Verilog can be found in [23].

2.4 Overview of Icarus Verilog

Icarus Verilog [25] is an open-source EDA (Electronic Design Automation) Verilog simulator being developed by Stephen Williams. 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.

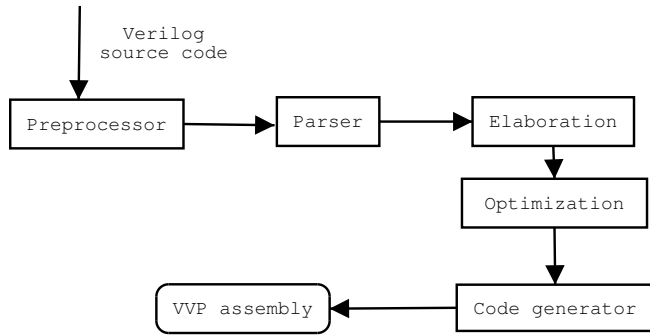


Figure 2: Architecture for Icarus Verilog compiler

2.4.1 IVerilog Compiler

As figure 2 shown, the Icarus Verilog compiler is composed of five components as the following:

- Preprocessor

The Preprocessor deals with file inclusion and macro substitution such as ‘include and ‘define. It generates the single output file which is equivalent to the original source files but without directives.

- Parser

The parser performs the syntax checking, semantic checking and transformation of the source code. The output of parsing is a list of module objects, which is just a direct reflection of the compilation. It’s similar to an abstract syntax tree and may contain dangling references.

- Elaboration

The elaboration component takes in the output generated by the parser and generates the netlist. It begins with the root module which is requested by the designers. If no root module is indicated, it’ll pick a root module arbitrarily, usually the first module without ports. It resolves references and expands the instantiations to form the design netlist which includes all behavioral descriptions as well as gates and wires.

- Optimization

The optimization component aims to eliminate null effect circuitry, execute combinational reduction and constant propagation. It’s essential for the performance of the simulator.

- Code generator

The code generator takes in the netlist and generates the target code. VVP assembly is the target code for Verilog simulation.

VVP assembly uses *.functor* operation code to define the basic structural units of the digital system such as gates. Every functor includes a type in the form of truth table and up to 4 inputs. A *.udp* operation code is used to define user defined primitives in Verilog.

A *.thread* statement is used to represent the initial and always block in Verilog source code. Each thread contains code space which is equivalent to a behavioral description of Verilog source code. It is important to note that thread runs in virtual machine of the VVP simulator instead of running directly in the operating system.

.event statements are used for interactions between functors and threads. *.event* statement provides a means of watching for a value change of a functor or some functors. Once the event is triggered, it will wake up all the sleeping threads which are waiting for the event to change.

2.4.2 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)[4]. 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.

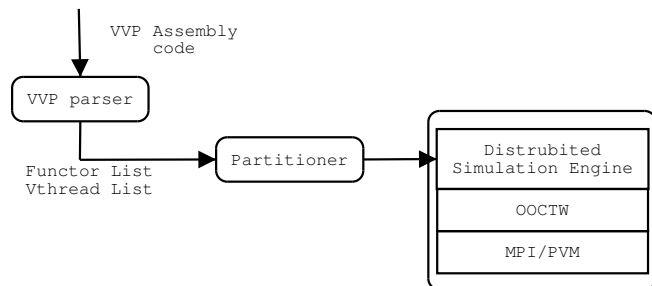


Figure 3: Architecture of DVS

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.

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[5]. 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 [2]. Empirical studies [5, 4, 17, 22] 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 trade-off among these three factors. In order to research and 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

Partitioning is the starting point for a distributed simulation. Should we employ a sophisticated but time-consuming partitioning algorithm, which results in the best tradeoff between cutsizes and concurrency? Or will a simple and fast partitioning algorithm suffice with the dynamic load balancing mechanism? The comparison of these two strategies is one of the major interests in our ongoing research.

The design goal of our Partitioner is to provide a flexible and easily extensible test bed for two purposes: testing different partitioning algorithms and applying the algorithms to different implementations of a circuit. As

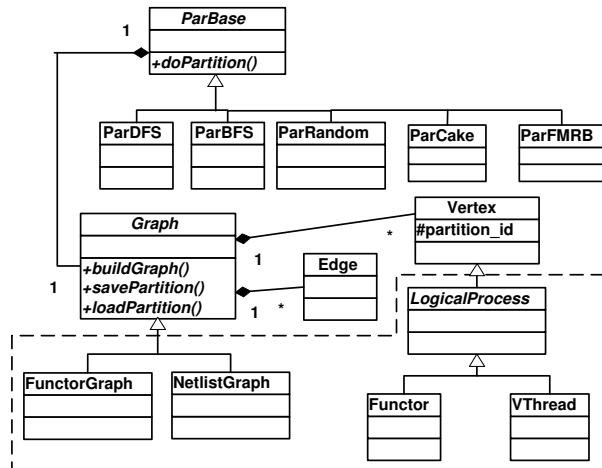


Figure 4: UML description of the Partitioner

shown in figure 4, the Partitioner consists of two separated parts: the circuit graph being partitioned and the partitioning algorithm.

3.3.2 Circuit graph

The circuit graph is the target for all partitioning algorithms. Before partitioning is performed, the concrete gates and wires in the circuits need to be mapped to vertices and edges and interconnected into an abstract graph. A circuit graph can be expressed as either a simple graph or a hypergraph. We choose the hypergraph format because it also contains all of the information for a simple graph. Another choice concerns the graph itself: whether to use an adjacency matrix or an adjacency list. Since our target circuit graphs are huge (we intend to simulate million-gate circuit) and the average degree of vertices is small, the adjacency list implementation is apparently more appropriate and efficient.

The circuit graph is represented by Vertex and Edge objects in the abstract Graph class. In order to use all the partitioning algorithms, users of our Partitioner only need to subclass the Graph class and provide implementation for the buildGraph method to fill in vertices and edges using application-specific information. The Graph class provides interfaces to partitioning algorithms for retrieving information for vertices and edges in the graph.

One of the major aims of DVS is to simulate million-gate circuits. From experiments we discovered that there is not much performance improvement potential of PDES for simulating small to medium size VLSI circuits. The speedup gained from PDES can hardly compensate even the overhead of PDES itself. However, the major difficulty we are facing is the lack of huge circuit

benchmarks. The classical ISCAS85 and ISCAS89 circuit benchmarks are small. The ISPD98[3] circuit benchmark suite is much more up-to-date and larger, but they cannot be used for simulation since they contain no gate information. They are good candidates for testing the partitioning algorithms in the Partitioner. We implement NetlistGraph to construct graph from the netlist files in the ISPD98 circuit benchmark suite.

In DVS, we subclass the Graph by FunctorGraph to build the graph using the information in functor list. From the class hierarchy displayed in figure 4, we can see that functors are themselves vertices, so we only need to instantiate edges and connect them.

3.3.3 Partitioning algorithm

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. Currently we have implemented five algorithms: DFS, BFS, Random, Cake and FMRB.

DFS, BFS, Random and Cake are all simple and fast algorithms. The Cake algorithm, which makes use of the proximity attributes of adjacent vertices, was inspired by the structure of the functor list. We simply cut the functor list into parts of required number as in slicing a birthday cake. The running time of Cake is the shortest and its result is as good as BFS. For small size circuit graphs such as the 16bit multiplier in the ISCAS85 benchmark, DFS, BFS and Cake have comparable resulting cutsizes to the sophisticated FMRB. The FMRB is the Recursive Bisection version of the classical FM[12] partitioning algorithm. For medium to large size circuit graphs, FMRB has the best resulting cutsizes together with a corresponding lengthy running time.

3.3.4 Design of FM

To perform a multiway partitioning, we have to make many decisions before we begin. Actually there are two dimensions in the classification of partitioning strategies in different perspectives. The first dimension is to use either flat multiway partitioning or recursive bisection multiway partitioning. The second dimension is to use either multilevel algorithm such as Metis/hMetis[16, 13], or non-multilevel algorithm such as CLIP/CDIP[21]. The Cartesian product of these two dimensions provides us with four different strategies. In the Partitioner we choose to use recursive bisection with non-multilevel FM algorithm.

Many popular partitioning algorithms are offspring of the classical FM, which is described extensively in the original paper[12]. Many details in different phases of the classical FM are unspecified, which arouse many interesting research[1]. The design of our FMRB aims to make it a good test bed for different variations of the classical FM. Some phases of the FMRB algorithm are especially designed as independent components which can be substituted for easily:

- Initial partitioning algorithm. Different initial partitioning algorithms will for sure affect the resulting quality of the partitioning. However, how big an effect it has and in what manner it biases the final result remains unclear to us. In the Partitioner we can easily plug in different algorithms as the initial partitioning algorithm. Some preliminary results (Figure 6 and Figure 7) are presented in the experiments section.
- Base cell selection. In our Partitioner, the vertices and edges are the cells and nets in FM. The base cell is the vertex being selected for the current movement. There are many different combinations of criteria for selecting the base cell.
- Gain updating. After each cell movement, the gains of the neighboring cells of the base cell need to be updated accordingly. CLIP/CDIP[21] modifies the rule of gain updating in order to extract large clusters out from the cutest.
- Slice and splice graph. In the recursive bisection, after the FM is applied on the graph of the current iteration, the graph needs to be sliced and spliced into two subgraphs for the next iteration. There are many design considerations about the slicing and splicing process. When a part of a hyperedge is cut in the current iteration, we can either remove the whole hyperedge in the next iteration, or just delete the part that is cut and the remaining hyperedge will still exist in the next iteration. When we construct the two subgraphs, we can either maintain the original order of the vertices or reorder the vertices.

3.3.5 Partitioning functors and vthreads

Since circuit information is available in both the IVerilog compiler and the VVP simulator, we can perform partitioning in either place. 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 the functor list, it can be readily used for message routing when the destination functor resides on a 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 the 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.

3.4 OOCTW(Object-oriented CTW)

3.4.1 Motivation

Clustered Time Warp(CTW)[4] 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[20].

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.

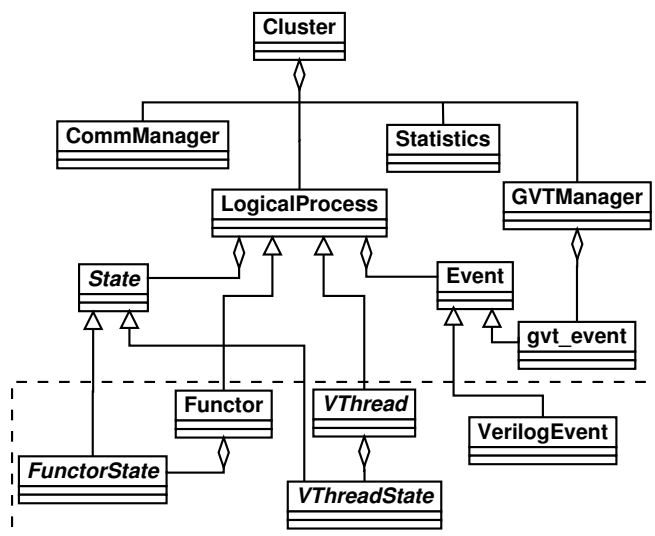


Figure 5: UML description of OOCTW

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 pro-

cessed 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[20].

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.

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. *Func* 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. *FuncState* 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 feed 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

<i>Type</i>	<i>Usage</i>
THREAD	Schedule a virtual thread
EVAL	Evaluate the functor
PROP	Propagate the value change after 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

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

- 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.

- Dynamic checkpointing

An dynamic checkpointing algorithm[14] is implemented in DVS. The regulating algorithm is initiated every 1000 events processed. The cost of saving state and coasting forward is computed through multiplying the average state saving and event processing cost in table 2 by the number of state savings and reprocessed events.

- Lazy cancellation

Some of LPs which are independent of the straggler become victims of the clustered rollback. Therefore, the same message that was produced during the original execution is again created when the event is reprocessed because of the unnecessary rollbacks. Hence, we implemented the lazy cancellation in DVS to try to avoid sending out unnecessary antimesages caused by clustered rollback. DVS only sends an antimesage if the original message was not again created.

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 1G byte per second. All machines run the FreeBSD operating system. Message passing between different processors uses MPICH-GM, which is developed by Myricom as a port of MPICH on top of GM(`ch_gm`) in order to exploit the lower latency and higher data rates of Myrinet networks.

We have two research groups which work in parallel. One is partitioning group, which concentrates on the research of the advanced partitioning algorithm. The other is the simulation group which focus on the distributed simulator. However, due to the shortage of circuits described in Verilog, the partitioning group uses different benchmark circuit with the simulation group.

All of the 18 circuits in the ISPD98[3] suite are used by the partitioning group in order to find out common patterns of FMRB with different initial partitioning algorithms.

The Verilog source file used by the simulation group describes a 16bit multiplier. It includes 2416 gates and one virtual thread which feeds 23 random vectors to 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 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 sequential VVP simulator without partitioning.

4.1 Selection of initial partitioning algorithm

We have run tests on all of the 18 circuits in the ISPD98 suite in order to find out common patterns of FMRB with different initial partitioning algorithms. The results for 2-way partitioning are presented in Figure 6

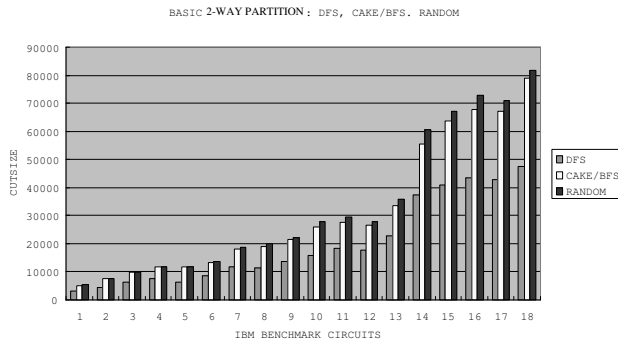


Figure 6: Basic 2-way partitioning algorithms.

and Figure 7 (We group CAKE and BFS together since they have the same results). As shown in Figure 6 all of the 18 circuits have the same pattern for the simple algorithms: DFS performs the best and Random is the worst. However, from Figure 7 we discover that this consistent pattern does not maintain anymore for FMRB. For most of the circuits, FMRB with initial Random partitioning algorithm is not the worst one. Also for the 4-way, 8-way, 16-way and 32-way FMRB partitioning, their results have similar attributes to the 2-way FMRB partitioning. We can come to the following conclusions with respect to the ISPD98 suite:

- FMRB is not very sensitive to the initial partitioning algorithm. The small cutsize of the initial partition does not guarantee the best final result. Although the Random algorithm generates the worst initial partition with a large cutsize, it may be more appropriate for the FM algorithm.
- FMRB is highly sensitive to the structure of the circuit graph. A larger circuit does not necessarily result in a larger cutsize. The structure of the circuit is more important. For instance, the circuit `ibm13` is larger than circuits `ibm01` through `ibm12`, but its resulting cutsize is smaller than most of them. For all the 2-way to 32-way FMRB partitioning, the relative cutsize of all the 18 circuits is the same.

4.2 Performance of DVS

The simulation time vs. the number of machines is shown in figure 8. 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 9, we know that the partitioning algorithm only reduces the total number of events processed on machine 1 by a small amount when

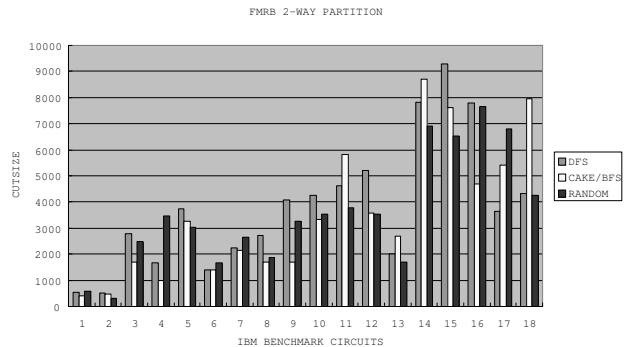


Figure 7: FMRB 2-way partitioning using different initial partitioning algorithms.

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 9 with average sending/receiving cost, which is listed in table 2. The reduction in workload is not large enough to compensate for 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, the large communication cost, the 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

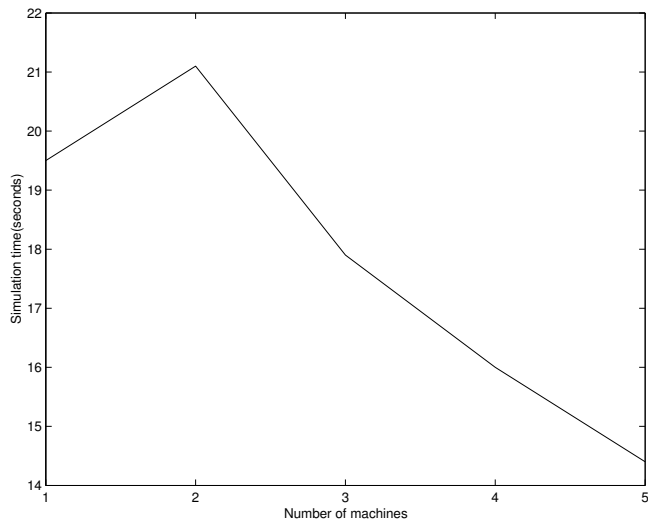


Figure 8: Simulation time in seconds vs. number of machines

<i>Operation</i>	<i>Time</i>
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

OOCTW as the synchronization backend and takes advantage of the open source code of the 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[10, 18, 19].

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.

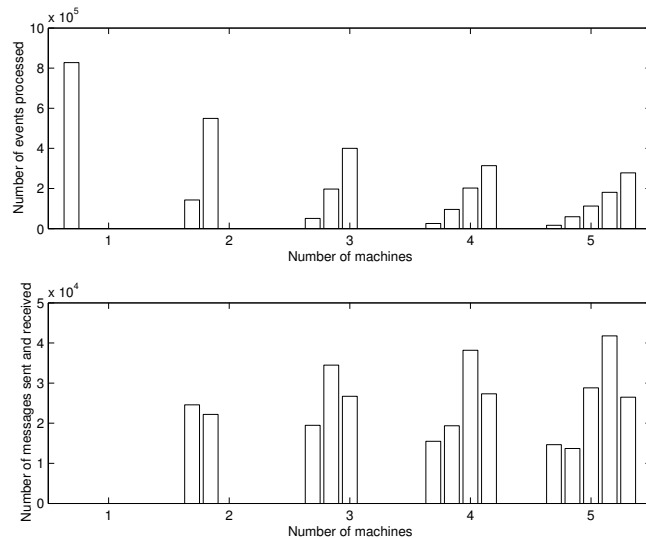


Figure 9: Number of events processed by every machine(Upper part) and number of messages sent and received(Lower part) by every machine vs. number of machines. Note: The two figures use different scale.

6 Acknowledgement

Hao Zhang and Tiantian Jiang, Master students in School of Computer Science, McGill University, contributed to the implementation of dynamic checkpointing and lazy cancellation as well as the experiments.

References

- [1] A. B. Kahng A. E. Caldwell and I. L. Markov. Design and implementation of the fiduccia-mattheyses heuristic for vlsi netlist partitioning. In *Proc. Workshop on Algorithm Engineering and Experimentation (ALENEX), Baltimore*, pages 177–193, Jan. 1999.
- [2] C. J. Alpert and A. B. Kahng. Recent directions in netlist partitioning: A survey. *Integr. VLSI Journal*, 19(1-2):1–81, Aug 1995.
- [3] Charles. J. Alpert. The ispd98 circuit benchmark suite. In *Proc. ACM/IEEE International Symposium on Physical Design*, pages 80–85, April 1998.
- [4] Herve Avril and Carl Tropper. Scalable clustered time warp and logic simulation. *VLSI design*, 00:1–23, 1998.
- [5] M. L. Briner Bailey and Chamberlain. Parallel logic simulation of vlsi systems. In *ACM Computing Surveys*, volume 26, pages 255–294, Sept 1994.

- [6] Prithviraj Banerjee. *Parallel Algorithms for VLSI Computer Aided Design*. Prentice Hall, Inc., 1994.
- [7] K.M. Chandy and J. Misra. Asynchronous distributed simulation via a sequence of parallel computations. *Communications of the ACM*, 24(11):198–206, November 1981.
- [8] C.J.R. Shi D. Lungeanu. Distributed event-driven simulation of vhdl-spice mixed signal circuits.
- [9] C.J.R. Shi D. Lungeanu. Distributed simulation of vlsi circuits via lookahead-free self-adaptive optimistic and conservative synchronization. In *Proc. ICCAD*, pages 500–504, Nov 1999.
- [10] Radharamanan Radhakrishnan Dale E. Martin and Philip Wilsey. Analysis and simulation of mixed-technology vlsi systems. *Journal of Parallel and Distributed Computing*, 62:468–493, 2002.
- [11] S. Subramanian et al. Study of a multilevel approach to partitioning for parallel logic simulation. In *IPDS00*, May 2000.
- [12] C. Fiduccia and R. Matheyses. A linear-time heuristic for improving network partitions. *ACM/IEEE Design Automation Conference*, pages 175–181, 1982.
- [13] 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.
- [14] 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.
- [15] D. Jefferson. Virtual time. *ACM Transactions on Programming Languages and Systems*, 7(3):405–425, 1985.
- [16] 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.
- [17] 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.
- [18] 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.
- [19] D. Lungeanu and C.-J.R. Shi. Parallel and distributed vhdl simulation. In *Proc. DATE*, pages 658–662, March 2000.
- [20] F. Mattern. Efficient algorithms for distributed snapshots and global virtual time approximation. *Journal of Parallel and Distributed Computing*, 18(4):423–434, 1993.
- [21] 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.
- [22] 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.
- [23] Donald E. Thomas and Philip R. Moorby. *The Verilog Hardware Description Language Fourth Edition*. KLUWER Academic Publisher, 1992.
- [24] A. Todesco and T. Ming. Symphony: A simulation backplane for parallel mixed-mode co-simulation of vlsi systems. In *Proc. 33rd DAC*, volume 6, pages 149–154, June 1996.
- [25] Stephen Williams. *Icarus Verilog*. <http://icarus.com/eda/verilog>.