

# VAST: Time Warp Simulation of VHDL on SMP Workstations\*

*Tim McBrayer, Venkatram Krishnaswamy, Sidhartha Mohanty, Lantz Moore,  
Xianghong Liu, James Carter, David Charley, Philip A. Wilsey, Debra A. Hensgen,  
Harold W. Carter, Praveen Chawla, John Collier and Scott Bilik<sup>†</sup>*  
Dept of ECE, PO Box 210030; Cincinnati, OH 45221-0030

## Abstract

The VAST project implements a parallel, optimistically synchronized, VHDL simulator on a network of Symmetric MultiProcessor (SMP) workstations. The VAST simulator targets the parallel execution of behavioral VHDL models across a network of heterogeneous SUN (Sparc) workstations. The workstations are heterogeneous in that they can be single processor, multiprocessor or configured with various sized local memory. Within each SUN workstation, the processes execute as lightweight threads. A parameterized communication subsystem was locally developed for the VAST that transparently implements message passing in shared memory or through SCRAMNet or Ethernet as capacity permits. Performance analysis of the VAST simulator execution speed and space including scaling as a function of available resource (e.g., CPU's) is also being conducted.

## 1 Introduction

The simulation of any sizable VHDL models requires considerable machine resources. It is not difficult to find VHDL models that no available sequential simulator can adequately service. Thus, even for moderately sized designs, digital system designers attempting to integrate VHDL into their design cycles are forced to independently simulate distinct portions of the VHDL; the designer must forego simulation as a vehicle for full system test. While this problem can be attacked with more efficient sequential simulators and large computers with huge memories, the size of VHDL models will likewise grow even faster. Consequently, alternatives to sequential simulation such as

parallel simulation need to be considered.

The VAST project proposes to address the problem of behavioral VHDL simulation by implementing an optimistically synchronized (Time Warp) simulator on a network of heterogeneous workstations. The simulator has been implemented to operate on a network of SUN SparcStations. The workstations are heterogeneous in that they can be single processor, multiprocessor or configured with various sized local memory. Virtually all of VHDL is supported and the simulator also implements file processing (often missing in parallel systems).

Within each SUN workstation, the processes execute as lightweight threads. The Time Warp simulation kernel has been extended with adaptive control techniques to dynamically adjust simulation time parameters for improved performance. In particular, the notion of an "index of performance" is borrowed from control theory to define a process level measure of "useful work." Changes in the useful work value are used to dynamically adjust simulation time parameters such as checkpoint intervals and the size of bounding windows. Furthermore, useful work is also used by a special purpose thread scheduler to decide scheduling decisions.

A communication subsystem was locally developed for the VAST simulator. It is parameterized for tuning purposes and transparently implements message passing in shared memory or through SCRAMNet or Ethernet as capacity permits.

Lastly, to better understand the potential performance of the VAST simulator, performance analysis and scaling studies of the simulator execution speed and space requirements are underway. These studies are being conducted using: (i) inline probes to gather data, (ii) design of experiments statistical methods to analyze the data, and (iii) a variety of plotting and data representation methods to view the data. The scaling studies are using a new method called Discrete Event Flow Diagrams (DEFD) to model the ex-

\*This work is partially supported by the US Air Force Wright Laboratory Avionics Directorate under contract number F33615-93-C-1301.

<sup>†</sup>Wright Laboratory, WPAFB, OH 45433-7319.

ecution profile of the simulator on scalable parallel computer architectures.

The remainder of this paper is organized as follows. Section 2 contains a review of the Time Warp synchronization protocol for parallel simulation. Section 3 describes the implementation of the VAST simulator and communication subsystem. Section 4 discusses our work in performance characterization and scaling. Finally, Section 5 contains some concluding remarks.

## 2 Time Warp

Time Warp is a paradigm for optimistic synchronization of parallel discrete event driven systems [4, 6]. In this system, a simulation is decomposed into a set of logical processes (LP's), each with its own time-ordered event list. All LP's are allowed to execute independently, without explicit synchronization with each other. Events are timestamped messages exchanged between processes. When an event is received by an LP, it is inserted into the event list. The LP then executes events out of its event list in timestamp sequence, saving a complete copy of its internal state after each event is processed. Since the LP's are not explicitly synchronized, it is possible for an LP to receive an event with a timestamp less than the timestamp of the most recently executed message. When an LP detects such an event in its list, it must *rollback* its state to the state that existed at the time stamp of the straggler message causing the rollback. Any events that the rolled back LP has generated are cancelled by sending out antimessages to every destination the LP communicated with during the time interval being rolled back. If an LP receiving an antimessage has already processed the corresponding message, that LP must also roll back to preserve its own causality.

All LP's have certain data structures required to carry out simulation. The most significant of these are a state queue, an input queue, an output queue, and a local clock. The state queue is used to store the state after each event execution to allow rollbacks to occur. The input queue, or event queue, contains the events that the LP processes during the course of the simulation, while the output queue contains a copy of all messages generated by the LP and sent to other LPs. The data in the output queue enables the sending LP to generate the appropriate antimessages in the case of a rollback.

In addition to the set of logical processes, a central process is used to keep track of the Global Virtual Time (GVT), which is defined as the time before

which no process can roll back. This value, in general, can not be computed precisely, but several algorithms exist that can supply a very good lower bound for it [3, 7, 8]. Since, by definition, the states prior to GVT cannot be rolled back to, the memory used to save those states can be reclaimed and reused. In addition, since no LP can roll back before GVT, GVT is also used to signal the commitment of file output.

## 3 The VAST Simulator

A pictorial representation of the VAST simulator is presented in Figure 1. There are three chief components to the simulator, namely: the code generator, the simulation kernel, and the communication subsystem. Each of these is described in more detail below.

### 3.1 The Code Generator

The code generator operates on the compiled output of the Vantage tools [11] and produces compilable C++ code. It traverses the syntax tree, elaborating each block, component, or generate statement as it is located. A C++ class object is created for each fully elaborated VHDL process in the simulation description. These class objects become the LP's in the Time Warp simulation. The object contains methods for the creation, initialization, and execution of the process. It also contains methods corresponding to any VHDL functions or procedures called by the original VHDL process. All other methods, such as those for Time Warp synchronization or process bookkeeping, are supplied by the simulation kernel. Each process class is derived from the simulation kernel class to allow access the kernel's methods. The code generator also creates a makefile detailing the compilation of the entire simulation system.

### 3.2 The Simulation Kernel

The simulation kernel has been implemented in C++. It has been designed to implement the semantics of the simulation cycle as defined by the VHDL LRM [5]. In addition, it has the functionality and data structures to implement the synchronization requirements of the Time Warp paradigm for optimistic parallel discrete event simulation.

The simulation is divided into a set of logical processes (LP) that are created by the code generator. The input queue is divided into two queues, to differentiate between transactions arriving from other LP's via the message layer (the fan-in queue), and those inserted by means of signal assignment statements (the source queue). Wait statements are also

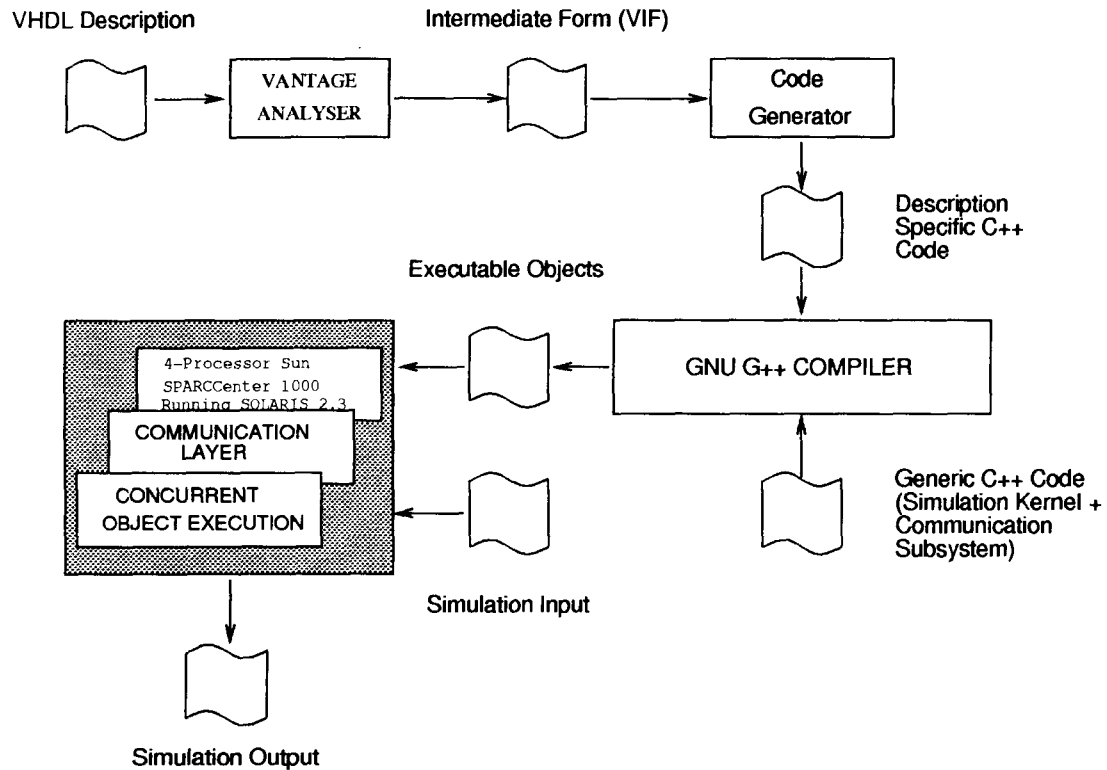


Figure 1: The VAST Simulator

handled by means of inserting a transaction into the source queue.

Marking is carried out on transactions in the source queue, as this constitutes the driver of a signal. In addition, it is sufficient to restrict GVT computation to the source queues. These two reasons are the motivation for separating the input queue into two distinct queues.

Input and output are dealt with by sending messages to dedicated objects for handling files. Input and output objects are derived from classes specially written to handle input and output. These objects have no part in the calculation of GVT although they must be able to access it, in order to commit output.

Several optimizations have been made in the kernel. In the past LP's were forked off by a manager process, as heavyweight Unix processes. A threaded version was implemented in which each LP is a Solaris thread. Initially the threaded simulator performed horribly. The poor performance was due to inefficient dynamic allocation of memory in the threaded environment. A more efficient memory management system was developed and used, successfully improving the overall performance of the simulator [9].

Aside from these optimizations, there are several

Time Warp related optimizations in the kernel as well. One such is Bounded Time Windows. This restricts the optimism of the process to a specified lookahead. The effect of this optimization is to prevent wasted lookahead, thus conserving system resources, and enabling the simulation, as a whole to proceed efficiently.

Other optimizations, namely periodic state saving and lazy cancelation have been implemented, and their effect upon the simulation is under study.

### 3.3 The Communication Subsystem

In this section, we overview the communication subsystem supporting the VAST simulator. We begin by describing our original parameterized communication subsystem which supported heavyweight processes in a networked environment [1, 2]. Next we summarize our work at optimizing this system for a particular environment: a single multiprocessor workstation. Using our experience and learnings from the original communication subsystem, we built a system that is tuned for facilitating communication between lightweight processes in this environment.

### 3.3.1 Parameterized Communication Subsystem

The communication subsystem provides message passing semantics, including `send_message()` and `receive_message()`, to support simulated VHDL processes on processors connected via physical shared memory, Ethernet, and SCRAMNet.<sup>1</sup>

The communication subsystem includes five types of objects (see Figure 2):

- A:** Simulation Objects that use the message passing system.
- B:** Message routers that route all messages passed through a shared area to the appropriate destination.
- C:** Message receivers that receive Ethernet messages and place them in shared memory.
- D:** A single SCRAMNet router that performs the same function as the message routers over multiple workstations.
- E:** Memory to memory interface manipulators that move messages between SCRAMNet and local shared memory.

The message routers manage a shared memory area and maintain routing information for simulation objects. A shared memory area is broken up into three parts:

- A single header containing the number of interfaces in the shared area.
- An array of object interfaces (One interface per object).
- An array of message blocks.

Each interface consists of a finite state machine that allows a process and message router to communicate in a mutually exclusive manner. This finite state machine allows for initialization between a process and the message router, the allocation of free blocks, the sending of messages, and shutdown activities. Each interface also contains a receive queue where messages are delivered to the associated simulation process.

---

<sup>1</sup>SCRAMNet, a product of Systran, INC, consists of a set of memory boards, one per workstation, connected via ring-structured fiber optics. The function of SCRAMNet is to emulate shared memory between physically distinct workstations. Upon writing to a SCRAMNet location from a workstation, a message is passed around the ring and the appropriate location on each memory board is updated to the new value. SCRAMNet does not guarantee consistency.

The message routers enable Ethernet connections by opening writable UNIX sockets so simulation objects that are unreachable through shared memory can be reached via Ethernet. Corresponding sockets are opened for reading on the appropriate remote machines. Each message router forks a message receiver in order to move Ethernet reading tasks to a background process.

The message receivers provide one way Ethernet communication from remote machines. Receivers block while waiting for incoming messages. Upon arrival of an incoming message, a receiver awakens, places the message in shared memory, and returns to the wait state.

If SCRAMNet is available then a memory-to-memory interface manipulator will automatically be created on each machine. This manipulator appears like a simulation object to the message router. However, any messages delivered to the manipulator are placed directly in SCRAMNet memory where they are delivered to another manipulator running on the destination machine. Once the manipulator on the destination machine finds the message, it is moved to local shared memory on the destination machine and eventually delivered to the appropriate process.

The SCRAMNet router performs the same function as the message router, except it does not deal with Ethernet connections. The only processes that will use SCRAMNet directly, in addition to the SCRAMNet router, are memory to memory interface manipulators.

Our original communication subsystem has many parameters that can be set at compile time. These parameters allow application designers to tune the communication subsystem to their own needs. A few of the parameters are described briefly: one parameter determines how often the router will attempt to recycle used blocks, another determines whether sent blocks are immediately replaced with free blocks, and yet another determines how often the router will look for situations where the router and a process need to be resynchronized.

### Tuning the Parameters for a Single Multiprocessor Workstation

In this section we discuss the tuning of some of the parameters of the communication subsystem for the multiprocessor workstation and the effect of changes in the choice of parameters on the runtime of the parallel simulation. All data presented below was taken from executing a set of 3 simulations several times, and performing a one factor analysis to determine statistical significance with a 99% degree of confidence.

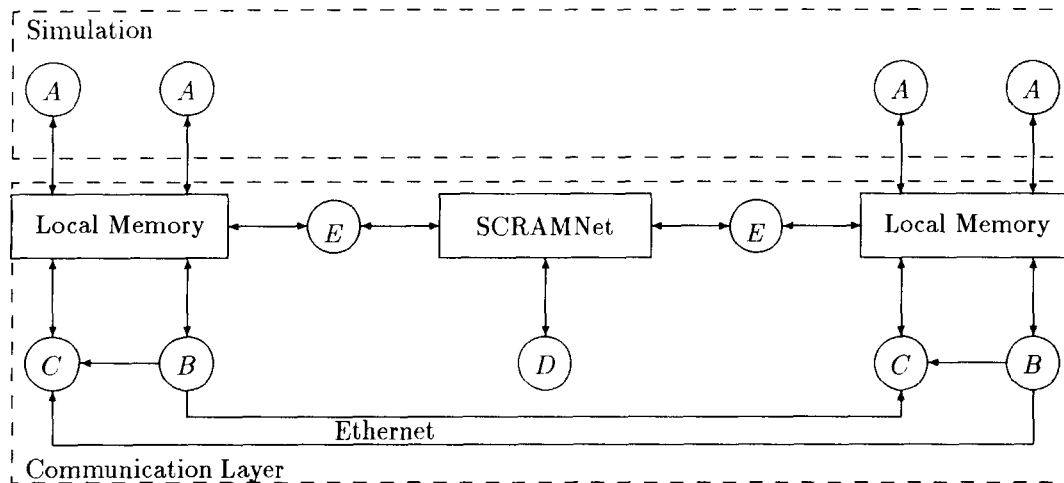


Figure 2: Circular figures represent processes, rectangles represent shared memory areas, and vectors represent communication connections.

- At startup time, each simulation object is given a number of message *blocks*; the default number of *blocks* is specified at compile time. When this initial resource is depleted, the object makes a request to the router for the default number of blocks. This parameter was varied from 5 to 95 blocks, giving peak performance at 65 blocks. A default value ranging from 65 to 95 showed no statistical difference. Using 65 blocks instead of 5 gave a speedup of 1.67.
- When the percentage of free blocks falls below a specified threshold, the router goes into reclamation mode and reclaims all free blocks. We were surprised by the results of the reclamation threshold study. This parameter was varied from 5% to 95% and showed no statistical difference.
- There are three different strategies employed by the router to locate messages to send. First, the router cycles through the list of objects, checking each for messages to send. If the router completes a full cycle, and finds no messages to send, it sleeps for a small period of time. In the second strategy, the router waits on a condition variable. When the condition is signaled by a sending object, the router wakes up and follows a hint directly to the index of the signaling object. The last mechanism simply has the router continuously cycling through the list checking for

messages to send. For our three example simulations, the continuous cycling strategy performed the best—1.2 times faster than using a condition variable, and 1.75 times faster than idling.

### 3.3.2 The Threaded Communication System

The parameterized communication subsystem described above is used for simulations made up of many Unix processes, each one representing a single logical process. However, large simulations consisting of several thousand logical processes and executing each as a Unix process requires substantial time for context switching. The design team decided to reimplement the simulation environment so that logical processes correspond to Solaris threads, rather than Unix processes. Neither the simulation kernel, nor the communication subsystem needed major modification during the ensuing port, however, the communication system was completely transformed to take advantage of the shared address space.

The major differences between the two implementations are the mode of message transit and message buffer allocation. The centralized router has been removed, making the communication between objects point-to-point. Message buffers are dynamically allocated from the heap, rather than a static buffer pool.

With the removal of the router, message traffic has become decentralized. Source objects directly in-

sert messages into destination object receive queues. There are two different receive queue implementations that maintain queue consistency: one uses software locks and the other uses a hardware swap instruction. Either implementation is available to the user via a compile time parameter.

The second major difference in the communication implementation is message buffer allocation. The threaded implementation dynamically allocates message blocks out of the heap, instead of from a static sized shared memory segment. This gives the threaded implementation greater flexibility.

Message buffers are also dynamically reused. When an object receives a message, the associated buffer is reused as the next outgoing buffer. This technique improves cache hits and reduces the number of new message buffer allocations.

Even though relatively young compared to the original communication subsystem, the threaded communication subsystem has enabled the simulation to show a marked improvement of 1.2 to 3.4 speedup for many simulations.

Future work includes integrating the threaded communication subsystem, as a parameterized instantiation, into the original communication subsystem.

## 4 Performance Modeling

### 4.1 General Simulator Evaluation

Automated methods are being employed to run, collect, and analyze sets of performance data. A user selects the parameters for a set of runs. These include: VHDL benchmarks to be run, numbers of processors or machines, a series of random input vector lengths, simulator optimization techniques, and number of repetitions. Families of performance runs are then run in sequential batch mode overnight when machines may be reserved and clear. An automatically generated perl script program is readily produced which follows the experiment plan. It can perform all necessary preparation for an individual run, launch the VAST simulator, and perform housekeeping, storage, analysis and graphing of results. For example, a user may select an experiment running the 16x16 bit multiplier benchmark for 1, 2, 3, and 4 processors, with 500, 1000, 2000, and 5000 random input vectors, for both the lazy cancelation and bounded time windows optimizations vs. the no optimization simulator configuration. Families of curves are then ready to be plotted and analyzed automatically or manually.

The data from each individual run is stored using a naming convention which uniquely identifies it. A

```
Graze Msgs
Event StartSend = Plus;
Event EndSend(Dst) = X:Dst;

Interval Send[StartSend, EndSend] = Line;
End Msgs.
```

Figure 3: An example Graze specification.

database is accumulated which allows for maximum reuse of performance run data.

### 4.2 Profiling in the Communication Subsystem

When analyzing large quantities of data, means and variances present a very limited perspective of the actions of the program generating that data. Particularly in multi-thread and multi-process paradigms, we need information that reveals temporal relations between those threads. Summarizing the data with just a few statistics blurs unusual interactions, making it difficult to reason about and fine tune a system. Presenting the data in a visual form allows a designer to examine as much, or as little, of the data as needed.

Graze is a visualization tool that allows a designer to focus on different aspects of system execution. Its purpose is to provide a simple, but robust analysis environment for performance data. Graze employs user defined events and intervals to carry out its graphical representation and statistical analysis. While graze provides a visual diagnostic for performance bugs, it also retains the functionality of a statistical package. This coupling furnishes very detailed performance analysis of a system.

Graze is designed to be user reconfigurable. The designer specifies *events* and *intervals* using a small language comparable to PSpec [10]; an example specification is shown in Figure 3. Graze is first used as a preprocessor, synthesizing C code from the graze specifications. The designer can insert these probes into his code where he sees fit. When the instrumented code is executed, a set of log files is produced.

When visualizing performance data, graze uses the designer's specifications to identify intervals from the events found in the log files. The profiled data is presented as a collection of vertical time lines, each corresponding to a single thread. Each symbol representing an event or interval, specified by the designer, is displayed in a different color making them readily distinguishable. Further easing the burden of analysis, the user has the ability to focus in on and move

around in the time lines, enable/disable events, intervals, and threads, and to find the significance of an interval type for a given period of time.

We have found graze to be a useful tool in identifying performance bugs in our simulation system. Within hours of first using graze, we were able to identify two performance bugs.

While using graze to analyze the Communication System, a curious pattern in the data was identified. We used graze to filter out all messages concerned with the calculation of GVT to see the percentage of messages directly involved with the simulation itself. We noticed midway through some simulations that the simulation was actually complete, however GVT cycles continued. We looked into the code and found that simulation objects were calculating GVT too conservatively, using undue restrictions. Alleviating this bug yielded up to 1.2 speedup.

The second performance bug involved simulation objects which write to files. When a simulation object writes information to a file, it sends its output to an intervening *file object*. The file object stores the write request until it can safely write the information to the file, that is, when GVT surpasses the virtual time at which the request was made. After the simulation object makes the write request, it must wait for an acknowledgment. We noted in some simulations that the writing objects would spend substantial time sending information to the file object after the rest of the simulation had completed. We also observed a reduction of concurrency due to the acknowledgments. Our current solution to this problem gives only meager performance gains and we are exploring other solutions.

### 4.3 Scaling Studies: Discrete Event Flow Diagrams (DEFD)

A new simulation modeling paradigm called Discrete Event Flow Diagrams (DEFD) has been developed for the system level performance evaluation, and in particular, scaling studies.

In the DEFD “world view,” system behavior consists of a set of interactive processes. A process can be represented by an active entity that cycles through a sequence of activities. By mapping this “world view” to a DEFD representation, a system can be modeled as a DEFD network, in which nodes represent the activities and arcs represent the sequencing of those activities. The active entity is represented by a structure called a *token*. Tokens are created and destroyed by special nodes. During their lifetime, tokens can pass through activity nodes, wait in queues for a resource, and conditionally branch given probabilities

on a user specified token attribute condition.

A DEFD network that models the operation of the VAST simulator has been developed and is being validated. A DEFD network represents the general operation of an object to be modeled (*e.g.*, VAST simulator). It includes operational flow and branch conditions.

While token flow is constrained by the DEFD network, the activity pattern in the DEFD network is largely governed by activity delays and branching probabilities. These factors are the VHDL program’s individual process delays and the overall VHDL model’s signal activity pattern. Thus, different VHDL benchmark models will have different characteristic delay and branching probability behavior which varies with differing VHDL models run on the VAST simulator DEFD model.

#### 4.3.1 DEFD Primitives

The DEFD modeling approach is based on this activity-based modeling concept. We define the following conceptual structures and types of nodes which form DEFD’s:

- **Token:** It is the representation of an ongoing process. A token may also have certain attributes.
- **Resource:** A resource has an attribute to show how many tokens it can process simultaneously. A waiting queue is attached to each resource.
- **Source Node:** Source nodes generate and submit tokens to the network based on a given random interval time distribution or “trigger tokens” from other nodes.
- **Sink Node:** Sink nodes consumes incoming tokens.
- **Activity Node:** Activity nodes model one system activity or a single step of the action that the system requires to finish a task. The resources and the time this activity needs are represented by the tags of the node. Tokens which arrive in this node must request and get the resources to go through the node. A departing token will release the resources.
- **Fork Node:** A token arriving at a fork node will trigger the creation of a number of buddy tokens. Each of them leave the node by a different outgoing path. This models the beginning of some parallel activities in the execution.

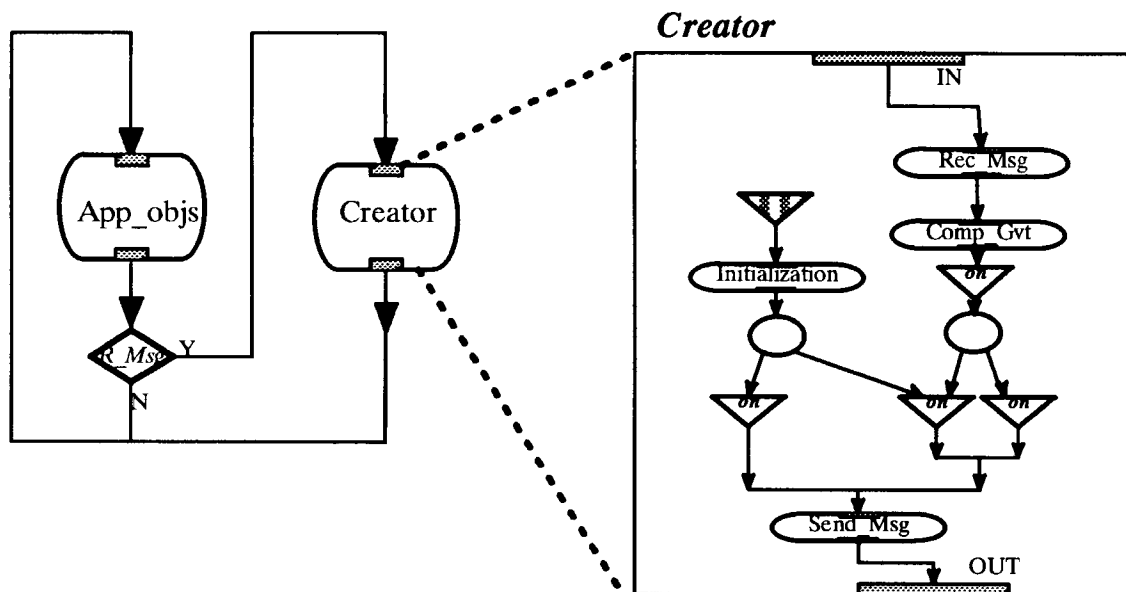


Figure 4: The top level hierarchy of VAST simulator DEFD model

- **Join Node:** Tokens arriving at a join node have to wait to join together with their respective buddy tokens from all possible incoming paths. Then the single joined token moves on to the next node. This models a synchronization point in the system.
- **Branch Node:** Tokens arriving into the node are branched to one of its outgoing paths based on user defined probabilities and/or conditions. It models the corresponding branching point of the system behavior.
- **Block Node:** A block node represents a DEFD subnet. It is included to support hierarchical modeling.

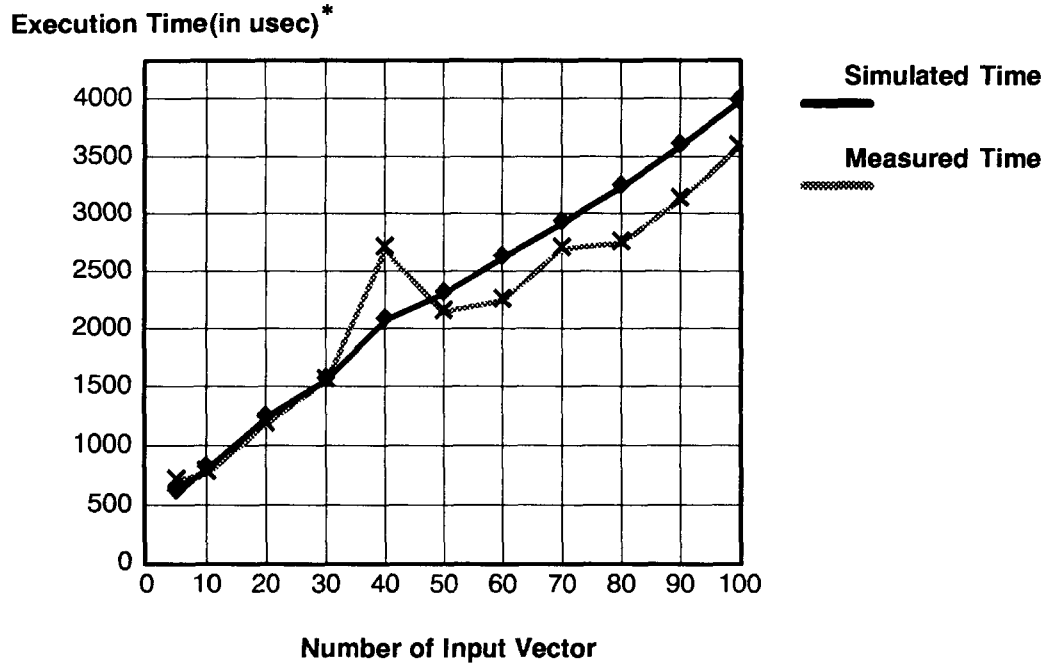
#### 4.3.2 DEFD Modeling

A DEFD simulator (DEFDsim) has been developed which accepts DEFD model descriptions in an easy-to-use DEFD description language, DDL. A DEFD description usually includes the model specification and the simulation control specification. In the control specification, a user can set the simulation to collect trace information about the token movement around a node or a resource, to report summary

statistics about the absorbed tokens at a sink node, or to monitor the token movement between two nodes.

For example, in modeling the VAST simulator, the activation of the application processes by the special process *Creator* at the beginning of the VHDL simulation can be modeled by a source node which generates a number of tokens simultaneously at the beginning of a DEFD simulation. The different operations on the messages received by the simulation processes can be modeled by a branch node which branches the tokens based on their attributes (*e.g.*, type). The probability of process rollback can be modeled by the branch node which branches the tokens based on predefined (and predetermined) probabilities.

For a computer system, a DEFD model can be created given the architecture of the hardware and the flow chart of embodied software. A DEFD modeling process includes the steps of DEFD model creation, parameters setting, and model verification and validation. Like any other modeling processes, a number of iterations of model modification and model verification and validation may be needed before a satisfactory model is obtained.



\* Benchmark program PMULT\_DEMO was used.

Figure 5: Comparison of VAST simulator DEFD simulation results vs. actual measured times

### 4.3.3 VAST simulator DEFD Model

For the VAST system, the DEFD model has been created as a three level hierarchy. Figure 4 shows the very top level model and the decomposition of the VAST Creator process. Here, the block *App-Objs* models the execution of general logic processes (main simulator function housing several dozen primitives) which execute VHDL processes. Block *Creator* models a special process which performs the initialization and the GVT calculation.

The DEFD network was made by translating the VAST simulator code's functionality. The parameters of the DEFD model were determined iteratively by running the simulator for one benchmark (16x16 bit parallel multiplier). Raw performance data collected from the executions were analyzed and the DEFD model parameters were chosen based on the analysis results. The benchmark was executed against the instrumented simulator and fine tuning was performed. The parameter values are likely to be different for different types of VHDL descriptions. The DEFD model was validated by comparing the results from the model simulation and those from the actual sys-

tem.

Figure 5 shows the validation result, in which the simulated execution time and the measured execution time for 5 to 100 input vectors is compared. The difference in tracking the actual performance is less the 15%. The one outlier point is believed to be due to someone inadvertently using the system and disrupting that run.

The DEFD models for systems with larger number of processors were built and simulated. They make predictions on performance for up to 10 processors even though we can only run a maximum of 4 shared memory processors. They also show utilization of shared memory and the processors.

All initial modeling efforts are in the process of being compared and validated wherever possible, to actual execution behavior. The degree of success of scaling forecasting is not yet known.

## 5 Concluding Remarks

The VAST parallel simulation project is attempting to establish a framework to demonstrate the suitabil-

ity of optimistic synchronization for parallel VHDL simulation. The project includes an implementation of a Time Warp simulation kernel together with several new and existing optimizations required to make Time Warp a practical solution. In addition, a suite of performance analysis and modeling tools have been constructed to evaluate the effectiveness of the simulator. Early indications show execution time performance comparable to sequential simulators but, because of its distributed nature, the parallel simulator has an ability to run much larger VHDL simulations than its sequential counterparts.

## References

- [1] CHARLEY, D., HENSGEN, D. A., MCBRAYER, T., WILSEY, P. A., AND ANKOLA, M. High speed communication for simulation of large VHDL models. In *Proc. of the Fall 1992 VHDL Int Users' Forum* (October 1992), pp. 212–216.
- [2] CHARLEY, D., MCBRAYER, T., HENSGEN, D., WILSEY, P. A., AND ANKOLA, M. Distributed simulation on a reconfigurable network using non-uniform message passing. In *Proc. of the 5th ISMM International Conference on Parallel and Distributed Computing and Systems* (October 1992), R. Melhem, Ed., pp. 247–250.
- [3] D'SOUZA, L. M., FAN, X., AND WILSEY, P. A. pgvt: An algorithm for accurate gvt estimation. In *Proc. of the 8th Workshop on Parallel and Distributed Simulation (PADS 94)* (July 1994), Society for Computer Simulation, pp. 102–109.
- [4] FUJIMOTO, R. Parallel discrete event simulation. *Communications of the ACM* 33, 10 (October 1990), 30–53.
- [5] *IEEE Standard VHDL Language Reference Manual*. New York, NY, 1988.
- [6] JEFFERSON, D. Virtual time. *ACM Transactions on Programming Languages and Systems* 7, 3 (July 1985), 405–425.
- [7] LIN, Y.-B., AND LAZOWSKA, E. Determining the global virtual time in a distributed simulation. In *1990 International Conference on Parallel Processing* (1990), pp. III-201–III-209.
- [8] MATTERN, F. Efficient algorithms for distributed snapshots and global virtual time approximation. *Journal of Parallel and Distributed Computing* 18, 4 (August 1993), 423–434.
- [9] MOORE, L., HENSGEN, D. A., KRISHNASWAMY, V., AND WILSEY, P. A. The importance of dynamic memory allocation in threaded applications. Tech. Rep. TR 171-6-94-ECE, Dept of ECE, University of Cincinnati, Cincinnati, OH, June 1994.
- [10] PERL, S. E., AND WEIHL, W. E. Performance assertion checking. *Operating Systems Review* 27, 5 (December, 1993), 134–145.
- [11] VANTAGE ANALYSIS SYSTEMS. *VHDL Intermediate Format: Access Routines, Reference Manual*, July 1989.