Design and Implementation of an Actor Based Parallel VHDL Simulator [1]

V. Krishnaswamy

462 CSRL 1308 W. Main

Urbana IL 61801

phone: 217 333 4767 FAX: 217 333 1910

email: venkat@crhc.uiuc.edu

P. Banerjee

469 CSRL 1308 W. Main

Urbana IL 61801

phone: 217 333 6564 FAX: 217 333 1910

email: banerjee@crhc.uiuc.edu

[1]This research was supported in part by the National Science Foundation under grant MIP-9320854, the Semiconductor Research Corporation under grant SRC 95-DP-109, and the Advanced Research Projects Agency under contract DAA-H04-94-G-0273 administered by the Army Research Office. We would also like to thank Intel Corporation for the donation of an Intel Paragon to the University of Illinois.

Abstract

One of the methods used to reduce the time spent simulating VHDL designs is by parallelizing the simulation. In this paper, we describe the implementation of an object-oriented Time Warp simulator for VHDL on an actor based environment. The actor model of computation allows the exploitation of fine grained parallelism in a truly asynchronous manner and allows for the overlap of computation with communication. Some preliminary results obtained by simulating a set of multipliers and some ISCAS benchmark circuits are provided. In addition, the importance of placing processes based on circuit partitioning techniques for improving runtimes and scalability is demonstrated. Results are reported on a Sun SPARCServer 1000 and an Intel Paragon.

Keywords: VHDL, Parallel Discrete Event Simulation, Time Warp systems.

1 Introduction

The design of a digital VLSI system commonly begins with a description of the system being written in a Hardware Description Language, an example of which is VHDL [12]. Subsequent to verifying the functionality of the description, it is given to a system to perform synthesis at varying levels of abstraction, beginning with architectural synthesis and ending with layout synthesis.

Verification of the functionality of the description can either be done by formal techniques, or by simulation. The latter method is more widely in use. Large amounts of time are spent simulating modern HDL descriptions and parallel processing is an attractive approach to reduce the runtimes. Parallel simulation of digital systems is appropriate due to the increased parallelism available in modern pipelined designs.

VHDL has been designed for documentation and simulation of digital systems. Digital systems may either be described behaviorally or structurally in terms of components and their connectivity. Hierarchical descriptions may be flattened out to a set of equivalent `processes` which may execute in parallel. Execution within a process is serial.

This report describes the design and implementation of `properVHDL`, a parallel discrete event simulation system for VHDL, The simulator has been implemented on top of the ProperCAD II libraries [18] for providing an actor based [1] model for concurrent object oriented programming. The level of granularity of parallelism in properVHDL is the VHDL `process` statement. Equivalently, there is a separate actor corresponding to each process statement in the user's VHDL source code. The synchronization mechanism chosen for the parallel discrete event simulator is Jefferson's [14] Time Warp system for optimistic synchronization. Each actor is a Logical Process or LP as defined by Jefferson in his paper.

The actor model of computation allows the exploitation of fine grained parallelism in a truly asynchronous manner and allows for the overlap of computation with communication. It is appropriate to implement a VHDL simulator in the context of an environment which provides support for fine grained parallelism because the amount of computation typically carried out in a VHDL process is typically small.

The entire properVHDL system comprises a VHDL front end analyzer which parses the user's source

VHDL into an abstract syntax tree form. This is used as an input to a code generator which produces a translation of the VHDL into C++. Each process is translated into a separate class, for which both `.H` and `.C` files are generated. The constructor of each such process contains data structures defining the signals and variables visible to the `process` statement. The translation of the process statement part is found in the `executeProcess` method of each of the generated classes. See Figure 1 for a pictorial depiction of the translation process. Wait statements, signal assignments and variable assignments are performed by making calls to methods in the VHDLActor class which is the base class for each of the generated classes. The derivation structure will be defined in greater detail in forthcoming sections.

As has been alluded to in the preceding paragraph, there exists a simulation kernel, which provides methods for performing actions such as signal assignments, wait statements, and variable assignments. This also provides the base class for each of the generated classes. In addition, all actions required for performing the parallel discrete event simulation are carried out by the kernel. These include extraction of events from the event list, state saving, handling of rollback, and performance of GVT calculations upon arrival of such a request.

This report is organized as follows. Section 2 briefly mentions some related work. A description of the simulation kernel appears in Section 3. The `UserInterface` class is described in Section 4, followed by the `VHDLActor` class in Section 5. The implementation of the GVT algorithm used in the simulator is discussed in Section 6. Some issues regarding File I/O appear in Section 7. Section 8 deals with compilation and code generation. Finally, we report experimental results and provide our observations on these in Section 9.

2 Related Work

In this section, we mention some work in the area of parallel VHDL simulation, and environments for parallel simulation. Jade Simulations International Corporation [13] have described a Time Warp based VHDL simulator. It is implemented on top of the *Sim++* simulation system that is a C++ runtime environment for distributed simulation using Time Warp. However, no experimental results have been reported. Wilsey and

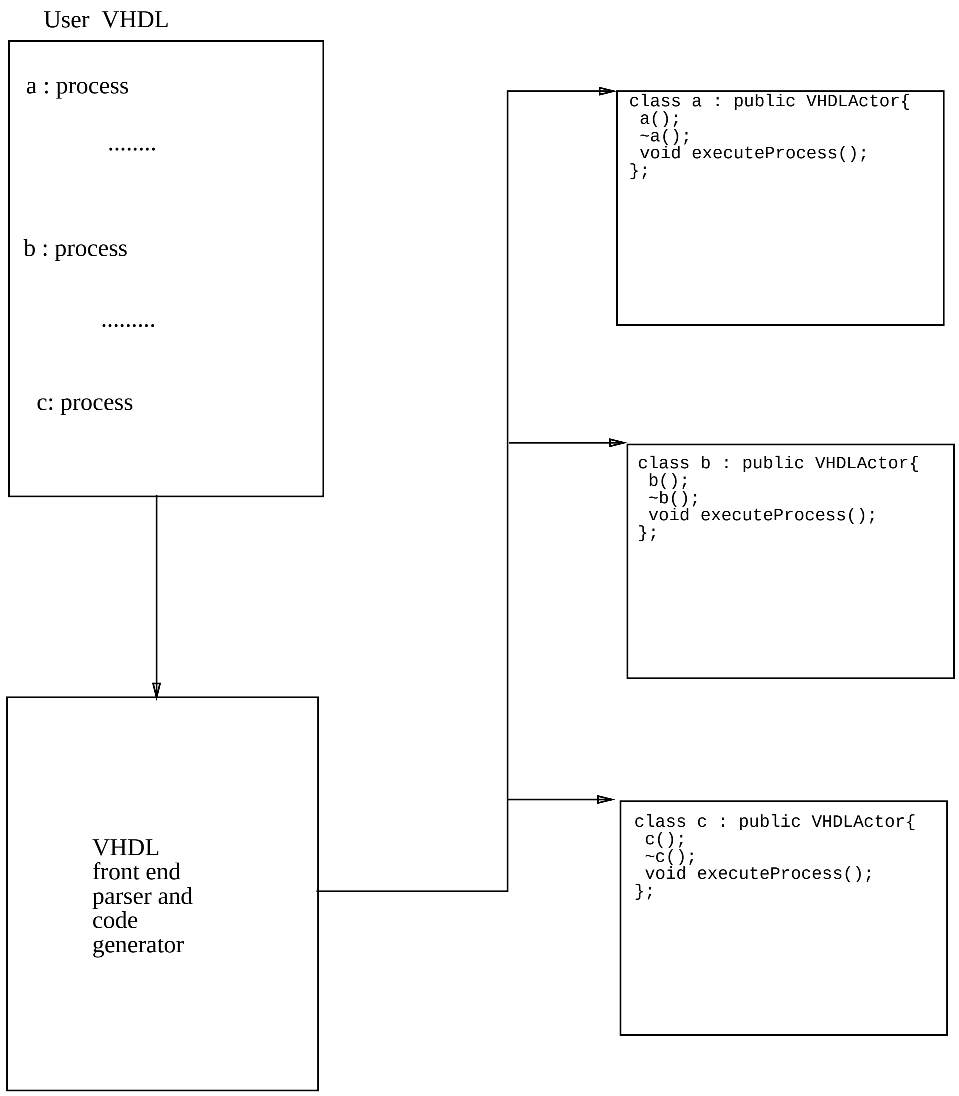

Figure 1: Processes in User VHDL are translated into classes derived from the kernel VHDLActor class.

McBrayer use the QUEST VHDL simulator based on Time Warp to investigate combination of processes to increase the computation grainsize [17]. Willis and Siewiorek mention the Auriga system in [25], which is mostly concerned with techniques for optimizing VHDL compilation for parallel simulation. While they report numbers reflecting their compilation techniques, actual simulation runtimes are not provided. Kapp *et. al.* [15] have built a conservatively synchronized VHDL simulator based on the Chandy-Misra [7] algorithm. Vellandi and Lightner [22] describe a SIMD algorithm for parallel VHDL simulation and use compilation techniques for extracting parallelism from the source VHDL description. Wen and Yelick [23] use a library based runtime system to construct a parallel circuit simulator. Bagrodia *et. al.* have written the Maisie language for describing parallel simulations [3] and this has been used to implement a gate level logic simulator described by Cong *et. al.* in [9]. There has been a great deal of related work in parallel logic simulation, which has been surveyed by Bailey, Briner and Chamberlain in [4]. Much of the preliminary work in parallel logic simulation and its parallelization using asynchronous algorithms was first reported by Soule [21].

3 Simulation Kernel

The simulation kernel provides the means for the execution of the simulation cycle as defined by the VHDL LRM [12]. Hence, the methods for extraction of events from the event queue, advancement of simulation time, insertion of events in the queue, and maintenance of the sensitivity lists are members of the kernel. In addition, data structures and methods for performing Time Warp activities, such as state saving, rollback and GVT computation are also members of the kernel.

3.1 VHDL Time Warp simulation in the context of Actor Model

In this section, the mapping of the VHDL Time Warp simulation onto the Actor Model of concurrent computation [1] is described. The kernel methods and data structures referred to above are provided within the `VHDLActor` class. The `VHDLActor` actor class is itself derived from the `pcActor` class. There is, therefore, one `VHDLActor` class for each `process` in the source VHDL. In other words, each `process` in the source VHDL is represented by an actor.

The `VHDLActor` class has a virtual method called `executeProcess`. Each `process` in the source VHDL is translated into a class derived from `VHDLActor`. The `executeProcess` method in each of the derived classes is a translation of the actions comprising the corresponding `process`. The invocation of `executeProcess` is dependent upon whether or not a change has occurred in the *sensitivity list* of the `process`.

Logical Processes in Time Warp communicate by means of exchanging messages. The pure actor model does not support the simple transfer of messages between communicating LP's. It is therefore necessary to communicate by calling *continuations* on the destination LP's. A continuation may be looked upon as a function pointer. Communication between the actors, which are the LP's, is performed by calling continuations upon one another. Continuations are called with arguments, and messages are exchanged between actors in this way.

Since continuations are similar to function pointers in a global namespace, it is necessary to know the names of the actors in order to ensure that the continuation is called on the correct actor. The code which involves sending of messages is in the methods of `VHDLActor` which is the base class for all the actors. The actual actors which are started up are the classes generated by the VHDL front end. It is therefore, impossible for the `VHDLActor` class to know the names of all the actors to which continuations are sent. This problem is tackled by generating a file which is essentially a table of all continuations, which can be indexed by a unique integer identifier given to each actor upon start up. Furthermore, this table is visible to each actor, and appropriate continuations are called by looking up the table. The actual implementation is described in a later section.

As of now, a centralized GVT management algorithm, Samadi's algorithm [20] is being used. This algorithm presumes the existence of a central GVT manager which is responsible for broadcasting GVT computation requests, receiving responses, computing and broadcasting GVT periodically. This functionality has been encoded in the `UserInterface` class. Figure 2 shows some of the interactions which occur between LP's in terms of the continuations invoked upon one another.

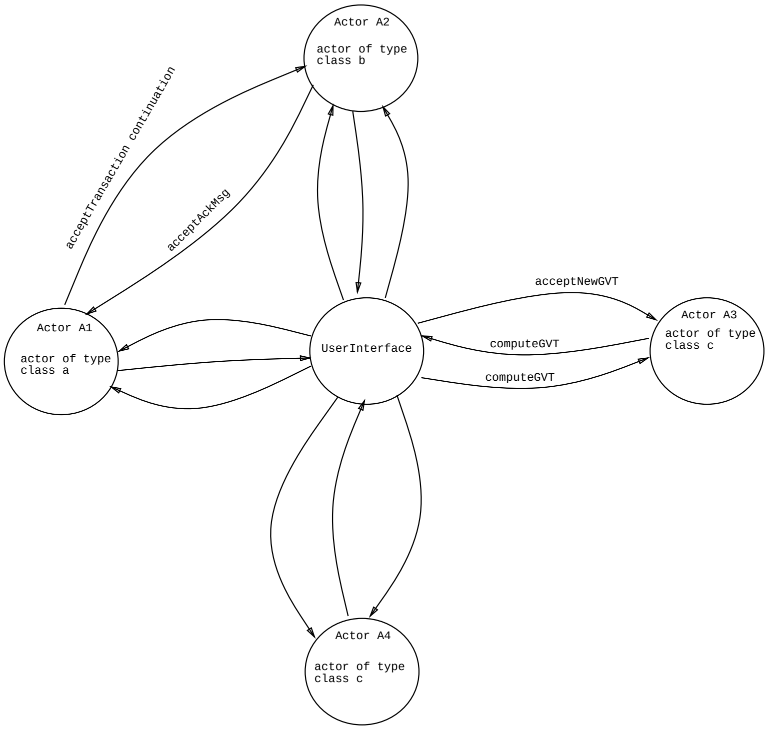

Figure 2: LP A invokes the `acceptTransaction` continuation on LP B, which responds by invoking the `acceptAckMsg` continuation on A. The UserInterface is shown sending and receiving GVT request continuations.

4 The UserInterface

The main program creates the UserInterface actor. The first task incumbent upon the UserInterface actor is to start up the simulation by creating actors of the types of each of the derived classes. It must then build the continuation table (see above) and make this visible to all actors. After the simulation is started up, the UserInterface maintains the GVT, in the case of centralized GVT algorithms.

4.1 Starting up the Simulation

The first action undertaken by the UserInterface is the invocation of the ForkSimActors method. Since the names and types of the actors are dependent upon the source VHDL, this method is generated automatically by the VHDL front end in a separate file. Figure 3 shows an example of this method.

The execution of the initial lines results in the creation of each of the actors in the simulation. Upon creation, each actor invokes a method in the UserInterface to signify its successful creation. The following lines are the building of the continuation table.

4.2 The Continuation Table

The exact nature of the continuation table, in terms of the size of the array is only known at compile time. Hence, this information is also generated automatically by the front end, in the form of the CTableData class. This forms the base class for the CTable class which contains methods for accessing the continuations. An example of the CTableData class is show in Figure 4.

An instantiation of the CTable class is a member of the UserInterface actor. This is the object which is shown being initialized in Figure 3. As soon as this initialization is completed, an *aggregate* [8, 18] is created, with the continuation table as a data member. As soon as the UserInterface knows that all actors have been successfully created, it creates the Continuation Table aggregate, with a representative on each processor. Once this has been created, the initSim actor on each of the VHDLActors is invoked with the name of the Continuation Table aggregate. This makes the continuation table visible to each of the actors

```
void
UserInterface::forkSimActors()
{
  ackActors::Customer report( *this );

  pcActorName<LOAD> A1 = pcActorName<LOAD>::newName();
  LOAD::New Actor1( report );
  Actor1( A1 );
  pcActorName<STORE> A2 = pcActorName<STORE>::newName();
  STORE::New Actor2( report );
  Actor2( A2 );

  numberProcesses = 2;

  ContinuationTable.numberOfRows = numberProcesses;

  ContinuationTable.initSimCustomer[0] =
    LOAD::initActor::Customer( ( pcActorName<VHDLActor>& )A1 );
  ContinuationTable.acceptTransCustomer[0] =
    LOAD::acceptTransaction::Customer( ( pcActorName<VHDLActor>& )A1 );
  ContinuationTable.SimCycleCustomer[0] =
    LOAD::SimCycle::Customer ( ( pcActorName<VHDLActor>& )A1 );
  ContinuationTable.acceptNegCustomer[0] =
    LOAD::acceptNegMsg::Customer ( ( pcActorName<VHDLActor>& )A1 );
  ContinuationTable.computeGVTCustomer[0] =
    LOAD::computeGVT::Customer ( ( pcActorName<VHDLActor>& )A1 );
  ContinuationTable.acceptNGVTCustomer[0] =
    LOAD::acceptNewGVT::Customer( ( pcActorName<VHDLActor>& )A1 );
  ContinuationTable.initSimCustomer[1] =
    STORE::initActor::Customer ( ( pcActorName<VHDLActor>& )A2 );
  ContinuationTable.acceptTransCustomer[1] =
    STORE::acceptTransaction::Customer ( ( pcActorName<VHDLActor>& )A2 );
  ContinuationTable.SimCycleCustomer[1] =
    STORE::SimCycle::Customer ( ( pcActorName<VHDLActor>& )A2 );
  ContinuationTable.acceptNegCustomer[1] =
    STORE::acceptNegMsg::Customer ( ( pcActorName<VHDLActor>& )A2 );
  ContinuationTable.computeGVTCustomer[1] =
    STORE::computeGVT::Customer ( ( pcActorName<VHDLActor>& )A2 );
  ContinuationTable.acceptNGVTCustomer[1] =
    STORE::acceptNewGVT::Customer( ( pcActorName<VHDLActor>& )A2 );
}
```

Figure 3: The `ForkSimActors` method.

```
class CTableData{
public:
  pcCustomer<pcAggregateName<CTableAggregate>> initSimCustomer[2];
  pcCustomer<TransMsg> acceptTransCustomer[2];
  pcCustomerVoid SimCycleCustomer[2];
  pcCustomer<TransMsg> acceptNegCustomer[2];
  pcCustomer<StartGVT> computeGVTCustomer[2];
  pcCustomer<lIntegral<int>> acceptNGVTCustomer[2];
  CTableData(){}
  ~CTableData(){}
};
```

Figure 4: The `CTableData class` - the array limits are determined only at compile time

in the simulation.

4.3 GVT computation methods

The `UserInterface` class is provided with methods for computation of GVT. Currently Samadi's algorithm for GVT computation is used. The `initiateGVT` method invokes the `computeGVT` method on each of the `VHDLActors`. These respond by invoking the `computeGVT` actor method in the UserInterface, which actually computes and broadcasts the new GVT.

5 VHDLActor and its Methods

`VHDLActor` is derived from `pcActor` and is the base class for all the generated classes. It contains methods and data structures for executing the VHDL simulation cycle, as well as carrying out the actions necessary for Time Warp simulation such as state saving and rollback recovery.

Figure 5 shows the Time Warp related data structures required by a Logical Process. Each VHDLActor has a `QMgr` class, whose responsibility it is to handle the input and output queues. There is also a `currentState` data structure, a `stateQueue`, the current LVT, `Now`, and the current `GVT`. The data type of the last two are `Time` which is a dual comprising the time in femtoseconds and an integer sequence number, to distinguish events occurring at the same time epoch, but different simulation cycles.

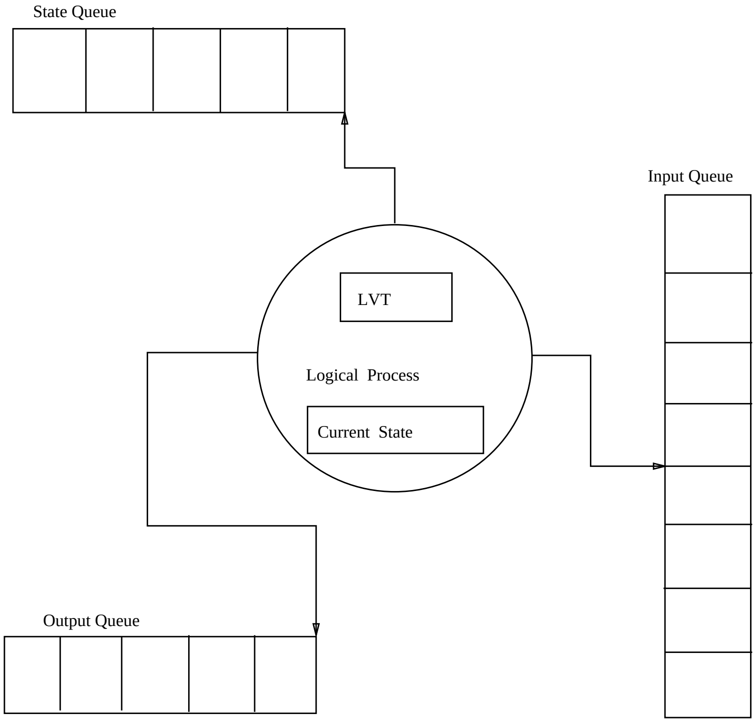

Figure 5: Data Structures internal to a Logical Process.

The initial sections of this section describe in detail, the TimeWarp related data structures and their implementation as members of the `VHDLActor` class. Later sections briefly describe some of the VHDL related activities, and the implementation of the simulation cycle.

5.1 The `QMgr` class

The `QMgr` class deals with the creation and maintenance of the input and output queues. Since both input and output queues contain transactions, or time stamped event messages, the `Transaction` class is the base class for all the queues. The functionality of timestamp order insertion and maintenance of transaction is handled within the `TransactionQ` class.

The input queue contains input transactions to the LP which have yet to be executed. The output queue contains executed transactions from the input queue which resulted in messages being sent to other processes, to enable sending antimessages in case of rollback.

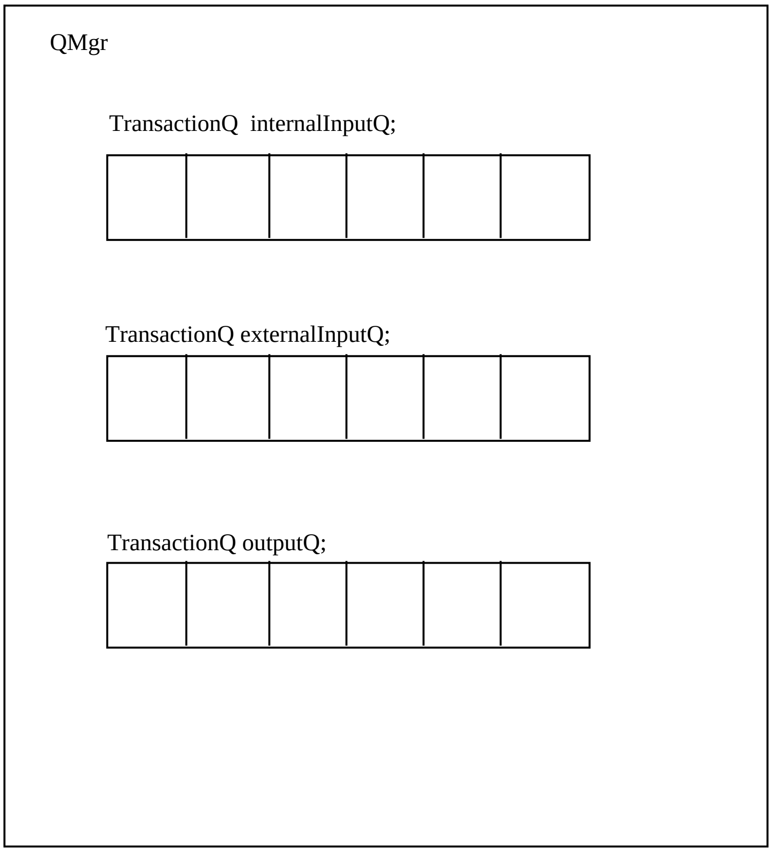

Figure 6: Data Structures in the `QMgr` class.

In our case, there are two sources of input transactions. The first possibility is the arrival of transactions from other LP's in the simulation. The second source is the execution of signal assignment statements in the process itself.[1]

To preserve the causal semantics of discrete event simulation, and to provide a means for `inertial` and `transport` delays on signal assignments, it is necessary to carry out forward and backward *preemption* on events created by the execution of signal assignments [2]. In order to easily implement this, the `QMgr` has two `TransactionQ` objects, `internalInputQ` and `externalInputQ`. As the names indicate, the former deals with internally generated transactions while the latter accepts transactions from other LP's. The third `TransactionQ` is the `outputQ`. Figure 6 shows a pictorial depiction of the `QMgr` class.

[1] Wait statements are also implemented as the insertion of events in the input queue, but these do not result in the sending of messages to other processes as wait statements, unlike signal assignments are local, and do not fanout.

Insertion of transactions into the `internalInputQ` is followed by *marking*, which is the VHDL LRM terminology for forward and backward preemption. Insertion of transactions into the `externalInputQ` must take the *sign*[2] into account. It should be noted that an antimessage can never be inserted into the `internalInputQ`.

The `outputQ` has been implemented such that both a message and its antimessage can simultaneously exist. This is an artifact of the current implementation of Samadi's algorithm. This done, so that it is possible to correctly account for acknowledgements from the message and its antimessage, if it becomes necessary to send out the antimessage. When a decision is taken to rollback, and antimessages to a message are sent, a transaction of opposite sign is created and inserted in the `outputQ`. In this way, it is possible to account for arriving acknowledgements easily. When a GVT algorithm which utilizes sequence numbers is implemented, it will be possible to do away with this creation of extra transactions.

5.2 The `stateQueue`

This is a simple doubly linked list of the state data structure. In the current implementation, state is saved with the occurrence of each event. However, as soon as a `coastForward` method is written, it will be easy to incorporate periodic state saving. The arrival of a new GVT causes fossil collection of the `stateQueue` whereas the arrival of a straggler results in rollback, and the restoration of an older state from the `stateQueue` as the current state.

5.3 ActorMethods dealing with receiving messages

This section describes some methods written for receiving messages from other LP's in the simulation.

5.3.1 `acceptTransaction`

This ActorMethod is called by other LP's when they require to send a transaction. The transaction is a member of the arguments to `acceptTransaction`. The transaction is inserted into the `externalInternalQ`

[2]whether a positive or antimessage

member of the `QMgr` object. In addition, the acknowledgement to the Transaction is sent by invoking the continuation sent as an argument to the method. The `acceptTransaction` method also detects whether the arriving transaction is a straggler, and calls a `rollbackHandler` method, if such should be the case.

5.3.2 `acceptNegMsg`

This is exactly identical to `acceptTransaction`, except that it handles the receipt of antimessages. Again, insertion is into the `externalInputQ` , and rollback detection and recovery management is performed as before.

5.3.3 `computeGVT`

This ActorMethod is called by the central GVT manager, (in this case, the `UserInterface`. Methods are invoked to calculate the minimum message in transit. The GVT is then calculated as explained in a later section, and the continuation in the `UserInterface` is called with the locally determined minimum.

The ActorMethods `acceptNewGVT` and `acceptAckMsg` are used for receiving newly calculated values of GVT and acknowledgement messages respectively. They are simple in their functionality and implementation and are not further described in this document.

5.3.4 Implementation of Wait Statement

A wait statement could come with any combination of none or all of a sensitivity clause, a condition clause and a timeout clause. A `Wait` data structure has been defined, to implement each of the wait statements appearing in a process statement. The fields in this data structure include an integer id to distinguish waits in a given process and an integer type field. The types of waits are enumerated as shown in Figure 7.

A wait statement which looks like `wait;` is recognized to be of type 0 since there is no accompanying sensitivity clause, condition clause or timeout clause. According to the LRM the timeout clause, simply specifies the *latest time* at which the process must resume. In other words if a change occurs in a member of an accompanying sensitivity clause or a specified condition evaluates to TRUE before the timeout expires,

sens_list	cond_clause	timeout
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1

Figure 7: Enumeration of possible types of wait statements.

the process is resumed. However, a wait statement containing a condition clause cannot resume until the condition evaluates to TRUE. Finally, a process statement with a sensitivity list is translated as a process with a wait on the same sensitivity list as the last statement in the process.

A timeout clause is implemented as the insertion of a transaction into the event queue. The wait statement times out when the transaction is executed. To deal with condition clauses the wait data structure contains a pointer to a method (generated by the code generator) which evaluates the condition. Finally sensitivity clauses are checked by looking at whether or not an event has occurred on a signal.

5.4 Implementation of the Simulation Cycle

The VHDL LRM stipulates the existence of an initialization step, during which signal and variable values are initialized. This is carried out by the `initSim` actormethod which is invoked by the `UserInterface` with the name of the Continuation Table aggregate.

The Simulation Cycle is defined by the VHDL LRM to be as follows.

- The current time T_c is set equal to T_n. Simulation is complete when T_n = TIME'HIGH and there are no active drivers or process resumptions at T_n.
- Each active explicit signal in the model is updated. (Events may occur on the signals as a result.)
- Each implicit signal in the model is updated. (Events may occur on the signal as a result.)

- For each process P, if P is currently sensitive to a signal S, and an event has occurred on S in this simulation cycle, then P resumes.
- Each non-postponed process that has resumed in the current simulation cycle is executed until it suspends.
- The time of the next simulation cycle, T_n, is determined by setting it to the earliest of :
  - TIME'HIGH
  - the next time at which a driver becomes active, or
  - the next time at which a process resumes.

  If $T_n = T_c$ then the next simulation cycle (if any) will be a *delta cycle*.

The implementation of the above is found in the `VHDLActor::SimCycle` method. Figure 8 shows a flowchart of the activities occurring within the method.

At the beginning of the simulation cycle, the `QMgr` is queried for outstanding transactions awaiting execution. This is done by searching both `internalInputQ` as well as `externalInputQ`. The transaction scheduled most immediately in the future is chosen. In case of a tie, the `internalInputQ` is chosen.

If a transaction exists, it is "executed". In other words, its value is copied into the signal for which it is intended, and the LVT is made equal to the scheduled time of the transaction. If the transaction has a fanout, the `acceptTransaction` methods on all LP's listed in the fanout list of the signal are invoked with the transaction as an argument. If there is no transaction, LVT is made equal to `p_infinity`. This process is repeated for all further transactions scheduled to occur at the same time.

At this point, it is worth mentioning how changes in the sensitivity list are kept track of. The `State` data structure has a sensitive flag which is initialized to 'off' at the beginning of each simulation cycle. As signals are assigned, events occurring on them are detected. If the signal is on the sensitivity list of the process, the sensitive flag is turned on. The sensitive flag can be overriden by a higher priority flag which is set at compile time, if the process has no sensitivity list (i.e. is always active).

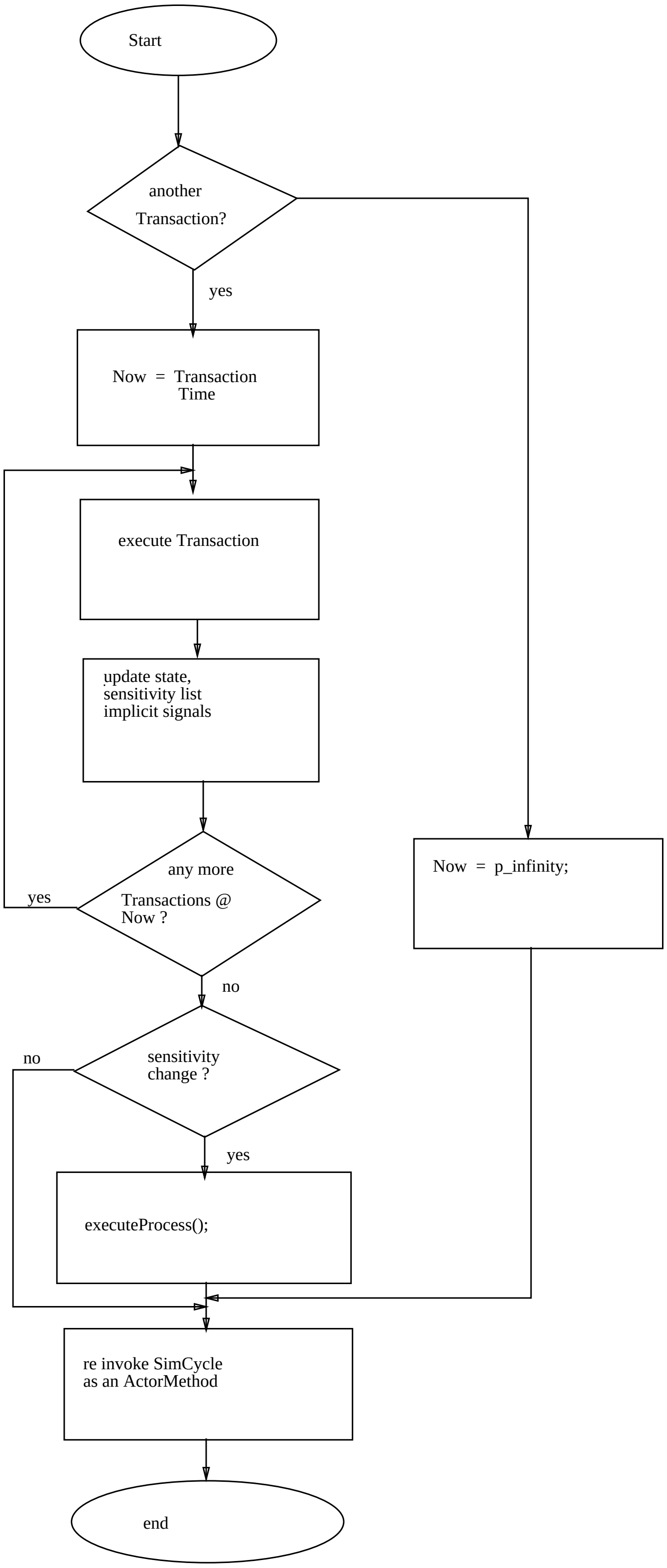

Figure 8: The `VHDLActor::SimCycle method`.

If the process is determined to have resumed by inspection of the sensitive flag, the `executeProcess` [3] method is invoked.

Finally, `SimCycle` is reinvoked as an ActorMethod. This is done in order to allow the properCAD runtime to schedule other actors assigned to the same processors. This prevents any one LP from advancing its virtual time too far, thus minimizing wasted lookahead in case of a rollback.

6 Implementation of GVT using Samadi's algorithm

GVT is defined as the lowest time to which any LP in the simulation may rollback. It is therefore defined as the minimum of all messages in transit, and the minimum LVT of executing LP's. There are two notable difficulties in calculating GVT [10], namely tracking messages in transit, and tackling the *simultaneous reporting* problem.

Samadi's algorithm tackles the first problem by introducing acknowledgements for every message and antimessage in the system. It then becomes trivial to find the minimum unacknowledged message simply by searching the output queue. The manner of implementation (placing sent messages as well as negative messages in the output queue) restricts the search to only the `outputQ`. The second problem is tackled in the following way. Whenever an LP receives a `StartGVT` message, it enters the *find* mode, and it remains in this mode until the GVT is reported to it. Any acknowledgements sent during this time are tagged with the fact that the LP is in find mode. The local minimum reported by each process is the minimum of

- current LVT
- minimum unacknowledged message
- minimum time of all acknowledgements tagged *find*

Notice that the last quantity decreases monotonically during any given GVT cycle for a particular LP. Hence it is sufficient to maintain a running minimum which is updated by the `acceptAckMsg` method. The

[3]recall that `executeProcess()` is virtual in the `VHDLActor` (base) class, but is redefined in each of the derived classes.

running minimum is updated to `p_infinity` when the new GVT is reported.

7 File I/O

File I/O in a Time Warp system presents some unique problems. In the case of output, rollbacks may necessitate invalidation of previously written values. On the other hand, a line of a file may have to be read on several occasions since rollbacks may destroy the work done with the value read on a previous occasion. The same problems present themselves when handling exceptions in the form of the VHDL `assert` statement. In this section, we describe how we handle File I/O in our simulator. We have used the C++ `strstream` facilities in our data structures for input and output.

7.1 Input

Each `VHDLActor` class has data structures for handling file input. The data structures created for this purpose are `InputLine` and `InputLineQ`. When `READ` or `READLINE` commands are detected in the source VHDL, calls are made to the `readLine` and `read` members of `VHDLActor`. The former creates an `inputLine` if that line has not already been read. The `inputLine` data structure is timestamped with the time of reading. It is thus possible to ascertain whether a line has been read from the file into the `InputLineQ` or not. The `read` uses the appropriate line to obtain the desired value. When the `VHDLActor` receives new values of GVT, the `inputLineQ` is garbage collected. The code generator is responsible for generating information on whether an object is a file reader or a file writer. It also generates code to open the appropriate input or output file.

7.2 Output

Data structures for file output are defined in a manner similar to that of file input. In this case they are named `OuputLine` and `OutputLineQ` respectively. `WRITE` and `WRITELINE` statements in the user VHDL are translated into calls to the `write` and `writeline` methods of `VHDLActor`. These calls result in creation and updation of `OutputLine` data structures. These `OutputLines` are enqueued, but are not committed until

GVT rolls past the creation time of the `OutputLine`. When this occurs, the lines are written out into the files, and are then garbage collected.

8 Code Generation

In this section, we describe how we generate C++ code given a user's source VHDL. The entire code generation process is divided into two parts – analysis or parsing and conversion into an intermediate form, and elaboration and code generation or conversion of the intermediate form into a simulation. The first subsection describes the analysis phase, and the next deals with the code generation. Elaboration deals with extraction of a netlist from a structural description.

8.1 vcomp

The VHDL front end analyzer, `vcomp` has been implemented using the PCCTS [19] LL(k) parser generator. The VHDL 93 BNF provided in the VHDL LRM had to be modified to remove all the left recursive rules. PCCTS allows varying the lookahead in terms of lexical characters. This feature proved particularly useful in dealing with the many grammatical ambiguities of VHDL. In addition, there is support for automatic generation of an abstract syntax tree (AST).

8.2 codegen

The C++ code generator is described in this section. The code generator uses the file containing the AST which has been produced by `vcomp`. In addition the symbol tables are also accessed. Since many data structures are common to both `vcomp` and `codegen`, the code defining these is shared between the two applications.

In its simplest form, `codegen` reads in the AST from the input file and reconstructs the tree in an internal data structure. The AST is then traversed in preorder to generate C++ code embodying the behavior specified in the processes of the user's VHDL. Initially only "flattened" descriptions were dealt with. In other words, `codegen` did not deal with hierarchy in a description.

In order to deal with the suspension and resumption of execution as required by the VHDL `wait` construct, in a manner explained in a forthcoming section, it was necessary to extract the control flow graph (CFG) corresponding to each VHDL process.

8.2.1 Extraction of Control Flow

In this section, we describe the method used to extract the control flow graph from the AST. There is no separate data structure for the nodes of the CFG. Pointers reflecting control flow are placed in the AST nodes. The algorithm for control flow is shown below. In practice, owing to the structure of the AST, it is necessary to treat looping constructs and conditionals slightly differently, but these details are not shown here.

```
procedure extractCFG( Node )
begin
  Enqueue( Node );
  while( Queue has members )
    BuildBB( Dequeue() );
  end while;
end;


procedure BuildBB( Node )
begin
  while( Node is not a Leader )
    add Node to the current BB;
    Node <- nextNode( Node );
  end while;
  Enqueue ( Node );
```

Source VHDL	Generated Code

```
a : process
begin
  code A
  wait for timeout;
  code  B
end process;
```

```
void
ActorN::executeProcess()
{
  translate code A;
  enable executeProcess1;
  wait( 0 );
}

void
ActorN::executeProcess1()
{
  code B;
}
```

Figure 9: Translation of a VHDL process with a wait statement in the middle of the code

```
end;
```

8.2.2 Dealing with Suspension and Resumption

Wait statements entail suspension of a process statement until a condition is satisfied, timeout occurs, or event [4] occurs on a signal. When the process is ready for resumption, it must begin executing code from the statement succeeding the wait statement. In our scenario, this means that the `executeProcess()` method can potentially be suspended and resumed from an arbitrary position in the code. However, the actor model precludes suspension and resumption of a thread from an arbitrary point within it [1].

Hence, we had to mimic this suspension and resumption by splitting up the `executeProcess()` method into a set of methods with each such method ending in a wait statement after enabling the method containing the code succeeding the wait statement, as in Figure 9. Resumption is then implemented simply by calling the enabled method. A similar notion was first reported in [11] where issues regarding compilation of message driven programs from conventional imperative programs were discussed.

We now consider handling wait statements appearing in a branch of the *if-then-else* construct. Figure 10 contains the original and transformed control flow graphs (CFG) for this case. All the code upto the wait statement, including the *if* statement is placed in the method p1 (code A). If the condition in the *if* statement evaluates to TRUE, then the wait statement is executed after enabling method p2. When the wait

[4]In VHDL, an event is a change in the value of a signal. The VHDL equivalent of an event or timestamped message as used in PDES terminology is *transaction*.

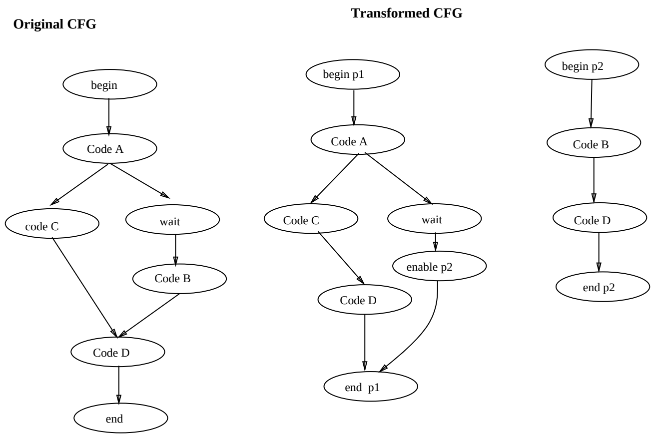

Figure 10: Generating code for a wait statement in a conditional construct.

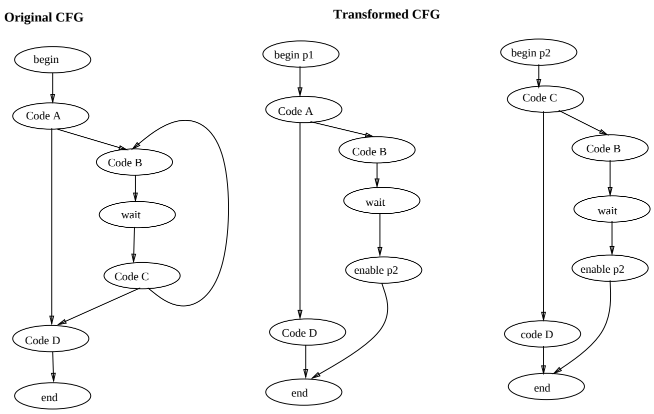

Figure 11: Generating code for a wait statement in a looping construct.

statement is satisfied, the method p2 is enabled. On the other hand, if the condition evaluates to FALSE, then execution of p1 continues in the appropriate manner.

The case of wait statements in a loop body is shown in Figure 11. The looping condition is evaluated in the method p1, and if the loop must be entered, then the wait statement is started after enabling p2. If the looping condition evaluates to FALSE then the code after the loop is executed. When the wait condition is fulfilled, the method p2 is invoked. On entry to p2, the remainder of the loop (Code C) is completed. Then the looping condition (Code B) is reevaluated. If it is necessary to reenter the loop, the wait is executed after enabling p2 again. If Code B evaluates to FALSE, the remainder of the program (Code D) is evaluated.

We now describe how we implemented the procedures described above. The CFG is first constructed as we have described in the previous section. In our implementation we have not dealt with wait statements in conditional statements. However, it is common in VHDL to have wait statements within while loops and straight line code. Furthermore, it is unlikely that these cases occur in isolation in a process. Therefore, it is necessary to determine the order in which to apply the CFG transformations described above. We perform the while loop transform from the innermost while loop containing a wait statement and proceed outward. We then apply the straight line transform to each of the CFG's resulting from the previous step. After transformation, there is a set of CFGs corresponding to each process. Code is generated by traversing the CFGs in a modified BFS order. In order to avoid being stuck in a loop by traversing the back edge of a loop in the CFG, it is necessary to mark nodes which have been traversed. However, in order to conserve space, CFGs which have common nodes merely contain pointers to the same instance of the common node. Hence the first time the node is traversed, the node is marked, making subsequent traversals of the other CFGs which point to this node believe that the node has already been traversed. As result code for this node is not generated. This problem is tackled by replacing the boolean flag with a counter, which is incremented with each subsequent traversal. If this counter exceeds the index value of the CFG currently begin traversed, then we have traversed a back edge and are getting stuck in a loop. This observation is stated without formal proof.

8.2.3 Dealing with Hierarchy

VHDL allows both behavioral and structural descriptions. Structural descriptions rely upon instantiating primitive components and specifying a set of signals which determine the connectivity of the components. In order to simulate such a description, it is necessary to obtain the circuit graph wherein the nodes represent primitive components and the edges represent nets. From the standpoint of code generation, we would like to accomplish the following goals :

- generate the circuit graph.

- generate code (a class) corresponding to each primitive component, and instantiate objects of each type corresponding to the component instantiations of the VHDL.

In order to accomplish the second goal, we must also generate a method (`initSignals`) in the generated class which will accept parameters specifying the circuit connectivity in a general way. Finally, a procedure is generated, calling `initSignal` on every object with the appropriate parameters. Due to the general nature of circuit connectivity, `initSignals` must be able to accept a variable number of parameters. In Figure 12, we include a structural architecture for the ISCAS 85 benchmark, c17. This will serve as an example to illustrate our point.

The figure shows instantiations of nand gates. A set of signals is declared at architecture scope, and these are used to form the connections between the instantiated components. Notice that ultimately this architecture will be instantiated within a test bench. Signals will be declared at that level to make connections from the `c17_i89` entity to the file reader and file writer. The problem then is, given such a hierarchical description, extract the circuit graph. This process is termed as elaboration.

Since a structural description presumes the existence of certain primitive components, we must analyze these prior to analyzing the structural architecture. In `codegen` C++ code is generated for every behavioral architecture analyzed. Hence, by the time the elaboration is performed the C++ classes corresponding to the architectures which will be instantiated have already been generated.

The signals defined at each level of the hierarchy correspond to nets in the circuit graph. A net, as used in this sense is analogous to a net in a physical circuit, wherein all the points it connects are electrically equivalent. Hence, for every signal, `codegen` instantiates a `Net` data structure which contains a pair of integers corresponding to the object Id and pin number of the driver of the signal, and two arrays of integers corresponding to the object Id's of the components which read the signal, and the pin numbers of readers. A circuit node is instantiated when it is detected that a behavioral component has been reached in the hierarchy. The algorithm in Figure elaborate shows this process.

The parameters to `elaborate` comprise a data structure containing a pointer to either a process or component descriptor, and pointers to each of the Nets associated with the ports of the descriptor.

```
ARCHITECTURE structural OF c17_i89 IS
  signal INTERP : std_ulogic_vector(0 to 3):=(others=>'0') ;
BEGIN
  NAND0 : NANDG_N generic map (2,1 ns,1 ns)
    port map (
      inp(0) => INP(0),
      inp(1) => INP(2),
      out1 => INTERP(0));

  NAND1 : NANDG_N generic map (2,1 ns,1 ns)
    port map (
      inp(0) => INP(2),
      inp(1) => INP(3),
      out1 => INTERP(1));

  NAND2 : NANDG_N generic map (2,1 ns,1 ns)
    port map (
      inp(0) => INP(1),
      inp(1) => INTERP(1),
      out1 => INTERP(2));

  NAND3 : NANDG_N generic map (2,1 ns,1 ns)
    port map (
      inp(0) => INTERP(1),
      inp(1) => INP(4),
      out1 => INTERP(3));

  NAND4 : NANDG_N generic map (2,1 ns,1 ns)
    port map (
      inp(0) => INTERP(0),
      inp(1) => INTERP(2),
      out1 => OUTP(0));

  NAND5 : NANDG_N generic map (2,1 ns,1 ns)
    port map (
      inp(0) => INTERP(2),
      inp(1) => INTERP(3),
      out1 => OUTP(1));

END structural ;
```

Figure 12: Structural architecture for c17

```
procedure elaborate( params P )
begin
  foreach signal in P
    create a Net;
  end for;

  if P contains a behavioral architecture then
    create a CktNode;
    update each Net in P with object and pin Ids;
  else
    foreach component instantiated in P
      create params with the associated component and Nets;
      elaborate( params );
    end for;
  endif;
end;
```

Figure 13: The elaborate procedure

9 Experimental Results and Observations

Our initial experience was with a pair of array multipliers – an 8x8 bit array multiplier (with 71 VHDL processes) and a 16x16 bit array multiplier (with 271 processes). Each description is fed an input vector file of 100 vectors. These multipliers are extensions of the 4x4 bit array multiplier reported in [24]. A diagram of such a multiplier is shown in Figure 14.

The ProperCAD II runtime system assigns actors randomly to processors. [5] unless otherwise advised. On a bus based symmetric shared memory machine (such as the Sparcserver 1000), the default random assignment of actors does not make too much of a difference, owing to the low communication cost. On the other hand, on a true distributed memory machine such as the Intel Paragon, placement of actors without attempting to localize communication in some manner can have impact the performance heavily.

The simulations were run on a shared memory multiprocessor (8 processor SUN SPARCServer 1000) and a distributed memory machine (20 node Intel Paragon). Our initial experiments are shown in Tables 1 and 2 . The numbers show that on the SPARCserver, the runtimes scale well with increasing number of

[5] actually the actors are assigned to processes, but we'll assume that they're equivalent. Note that this is not necessarily true on Unix SMP's where the OS decides whether or not each process is assigned to a different processor.

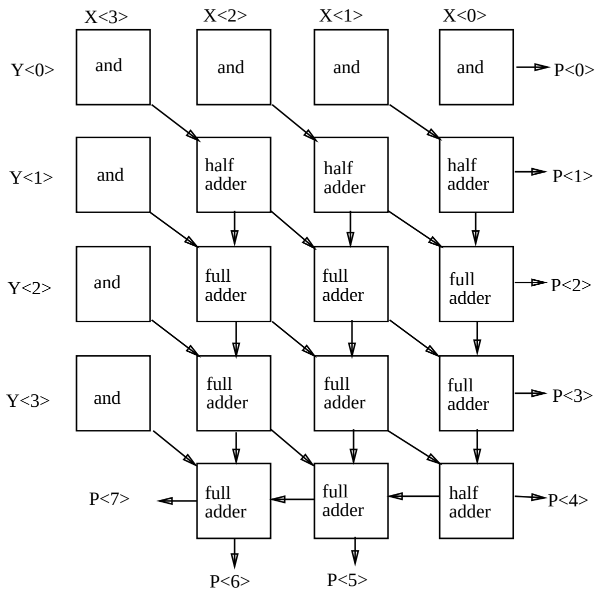

Figure 14: 4x4 bit array multiplier

.

Table 1: Runtimes in seconds with default actor placement on SPARCServer 1000

circuit	1 proc	2 procs	4 procs	8 procs
8x8	11.3	6.1	3.9	5.3
16x16	48.6	25.6	16.2	15.2

Table 2: Runtimes in seconds with default actor placement on Intel Paragon

circuit	1 proc	2 procs	4 procs	8 procs	16 procs
8x8	10.6	10.3	8.6	7.2	-
16x16	411.9	440.1	31.22	20.1	14.1

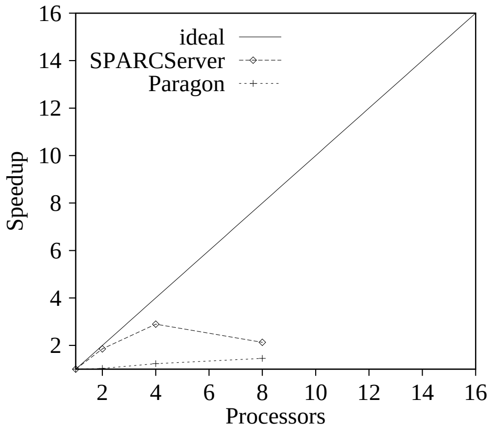

Figure 15: Speedups on the 8x8 multiplier

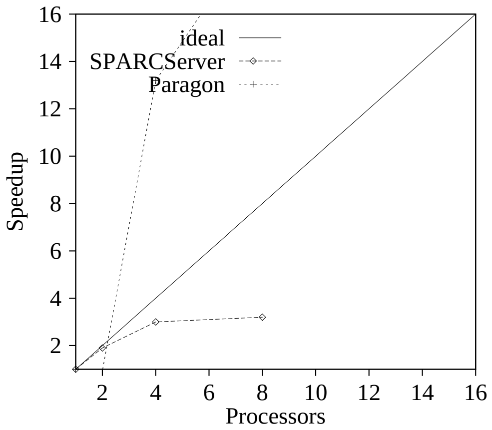

Figure 16: Speedups on the 16x16 multiplier. The seemingly superlinear speedups on the paragon are due to memory effects.

processors upto 4 processors and then a deterioration occurs. On the Paragon, however, the runtimes scale poorly upon increasing the number of processors. Figures 15 and 16 show plots of the speedups obtained from the runtimes shown in Tables 1 and 2.

The ProperCAD II runtime environment assigns actors to processes randomly. It starts up the simulation using the desired number of heavyweight processes, but assignment of actors to these processes is entirely random. To gain speedups especially on distributed memory machines, it is essential to place actors in such a way that communication is kept as local as possible. Random assignment of actors will therefore not optimize the communication. Hence, our next step was to apply strings based partitioning [16] to the

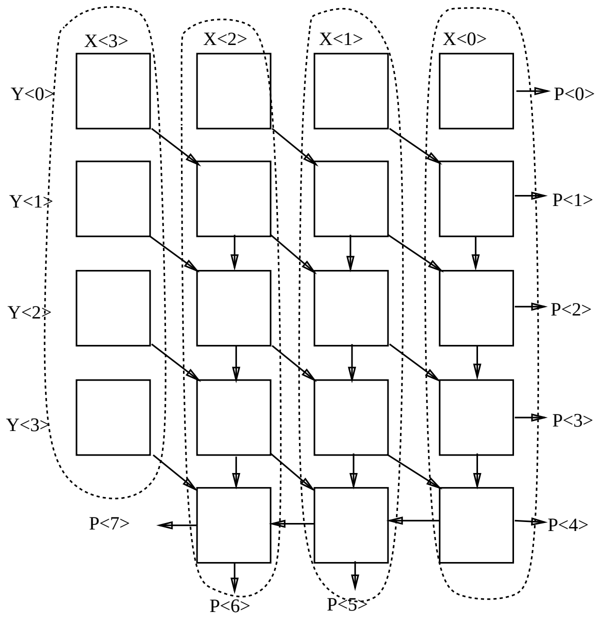

Figure 17: 4x4 bit array multiplier partitioned to run on 4 processors

.

multipliers. This yielded the partition shown in Figure 17. The partitioning strategy also attempts to take care of load balance by assigning approximately equal numbers of actors to each partition. In this case, the actors have approximately equal computational grain size. We therefore did not assign weights (reflecting the computational grain size) to each of the actors prior to partitioning. In descriptions having actors of varying grain size, it will be necessary to do so, in order to obtain a reasonably balanced partition.

The results obtained with partitioning applied are shown in Tables 3 and 4. Clearly, placement of actors

Table 3: Runtimes in seconds with circuit partitioned actor placement on SPARCServer1000

circuit	1 proc	2 procs	4 procs	8 procs
8x8	12.8	6.1	3.9	2.93
16x16	48.7	24.8	14.1	8.8

Table 4: Runtimes in seconds with circuit partitioned actor placement on Intel Paragon

circuit	1 proc	2 procs	4 procs	8 procs	16 procs
8x8	11.1	9.8	7.7	6.4	-
16x16	411.7	289.2	17.4	12.7	10.4

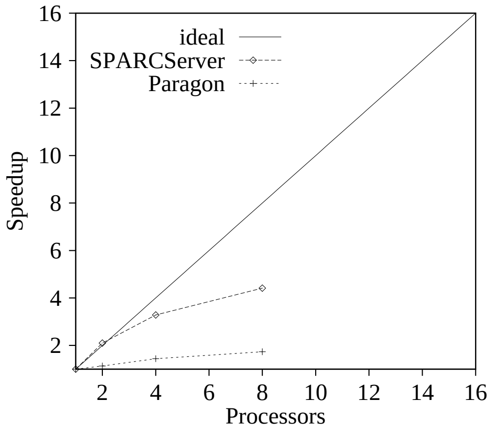

Figure 18: Speedups on the partitioned 8x8 multiplier

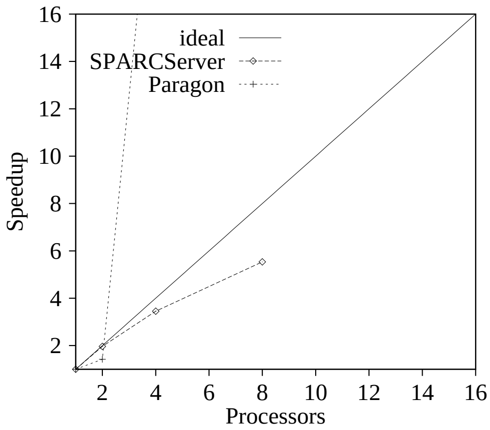

Figure 19: Speedups on the partitioned 16x16 multiplier. The seemingly superlinear speedups on the paragon are due to memory effects.

based on a circuit partitioning strategy improves the scalability of the simulation i.e. the overhead of communication does overcome the benefits of utilizing additional processors. Furthermore, since communication is localized, the number of rollbacks in the simulation decreases, leading to the improvements seen in the results. This corresponds to the result obtained by Cong et.al. [9], wherein a good partitioning leads to less blocking in a conservatively synchronized simulator. Figures 18 and 19 show speedups calculated from the runtimes shown in Tables 3 and 4.

It should be noticed that for the 16x16 multiplier, when 1 and 2 processors are used, the simulation is thrashing badly, but that the partitioning is able to produce a good result for 2 processors in spite of this.

Table 5: Runtimes in seconds for ISCAS benchmarks executing on SPARCserver.

circuit	#actors	1 proc	2 proc	4 proc	8 proc
c17	7	250.8	145.6	72.8	109.3
c432	162	356.6	258.3	121.3	102.4
c499	204	637.63	310.7	188.3	-
s298	135	28.8	12.85	8.11	5.8
s344	177	52.55	40.58	14.84	10.03

Table 6: Runtimes in seconds for ISCAS benchmarks executing on Paragon

circuit	#actors	1 proc	2 proc	4 proc	8 proc	16 proc
c432	162	-	582.747	-	24.92	20.81
c499	204	-	-	-	-	-
s298	135	-	-	-	-	-
s344	177	-	-	-	-	-

Using a larger number of processors relieves the thrashing problem.

In addition to the two circuits above, we have simulated some of the ISCAS [6, 5] set of benchmarks, described in structural VHDL style. The runtimes for these simulations can be seen in Tables 5 and 6. We had serious difficulties with memory management in simulating these benchmarks. These stemmed from two reasons. The first is the fact that Time Warp is inherently space intensive. The second and more uncontrollable reason was the behavior of the message queue management system of ProperCAD II . Under heavy message traffic, the space the runtime system continues doubling the length of the message system. The processes performing file I/O add a significant amount of message traffic. As a result, our results are biased by memory effects. In fact, it was difficult to run even small benchmarks on the Intel Paragon which has only 16M of memory per node.

Table 7: Runtime in seconds for ISCAS benchmarks on Vantage executing on a Sun 4 architecture

circuit	time	#vectors
c432	1.38	102
c499	1.3	100
s298	1.36	100
s344	0.95	100

We can see that there is a fair amount of parallelism to be exploited even in small circuits. Even c17 showed parallelism upto 4. The others all improved with increasing numbers of processors. However, the memory usage is unacceptably high. c499 could not be simulated on 8 processors due to excessive memory demands. While the memory demands due to Time Warp may be improved somewhat by applying more sophisticated state saving and GVT algorithms, and the runtime system can be improved, examination of Table 7, which contains runtimes of the ISCAS circuits using the commercial Vantage serial simulator suggests that there has to be a fundamental shift in the approach to parallel VHDL simulation.

References

[1] G. A. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* The MIT Press, 1986.

[2] Larry M. Augustin, David C. Luckham, Benoit A. Gennart, Youm Huh, and Alec G. Stanculescu. *Hardware Design and Simulation in VAL/VHDL*. Kluwer Academic Publishers, 1991.

[3] R. Bagrodia and W. Liao. A language for design of efficient discrete event simulations. *IEEE Transactions on Software Engineering*, March 1994.

[4] Mary L. Bailey, Jack V. Briner, and Roger D Chamberlain. Parallel logic simulation of VLSI systems. *ACM Computing Surveys*, 26(3):255–294, Sept. 1994.

[5] F. Brglez, D. Bryan, and K. Kozminski. Combinational Profiles of Sequential Benchmark Circuits. *IEEE Intl. Symp. on Circuits and Systems*, pages 1929–1934, May 1989.

[6] F. Brglez and H. Fujiwara. A Neutral Netlist of 10 Combinational Benchmark Circuits and a Target Translator in Fortran. *IEEE Intl. Symp. on Circuits and Systems*, 3(3), June 1985.

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

[8] A. A. Chien. *Concurrent Aggregates: Supporting Modularity in Massively Parallel Programs*. The MIT Press, 1993.

[9] Jason Cong, Zheng Li, and Rajive Bagrodia. Acyclic multiway partitioning of boolean networks. In *31st ACM/IEEE Design Automation Conference*, pages 670 – 675, June 1994.

[10] Richard M Fujimoto. Optimistic approaches to parallel discrete event simulation. *Transactions of SCS*, 7(3):153 – 191, June 1990.

[11] J G Holm, A Lain, and P Banerjee. Compilation of scientific programs into multithreaded and message driven computation. In *Proceedings of the 1994 Scalable High Performance Computing Conference*, pages 518–525, Knoxville, TN, May 1994.

[12] IEEE, New York, NY. *IEEE Standard VHDL Language Reference Manual*, 1988.

[13] Jade Simulations Int. Co. Implementation issues of Jade's VHDL simulator. Technical report, Calgary, Alberta, Canada, 1989.

[14] David Jefferson. Virtual time. *ACM Transactions on Programming Languages and Systems*, 7(3):404–425, 1985.

[15] Kevin L. Kapp, Thomas C. Hartrum, and Tom S. Wailes. An improved cost function for static partitioning of parallel circuit simulations using a conservative synchronization protocol. In *Proc. of 9th Workshop on Parallel and Distributed Simulation*, pages 78 – 85, June 1995.

[16] Y.H. Levendel, P.R. Menon, and S.H. Patel. Special purpose computer for logic simulation using distributed processing. *Bell System Tech. J.*, 61(10):2873–2910, Dec. 1982.

[17] Timothy J. McBrayer and Philip A. Wilsey. Process combination to increase event granularity in parallel logic simulation. In *Proceedings of Int. Parallel Processing Symposium*, April 1995.

[18] Steven Parkes, John A. Chandy, and Prithviraj Banerjee. A library based approach to portable, parallel, object–oriented programming: Interface, implementation and application. In *Supercomputing '94*, November 1994.

[19] T. J. Parr and R. W. Quong. ANTLR: A predicated-LL(k) parser generator. *Software, Practice and Experience*, 25(7):789, July 1995.

[20] B. Samadi. *Distributed Simulation, Algorithms and Performance Analysis.* PhD thesis, UCLA, 1985.

[21] Larry Soule and T. Blank. Parallel logic simulation on general purpose machines. In *Proc. 26th Descign Automation Conf.*, pages 81–86, June 1989.

[22] B. Vellandi and M. Lightner. Parallelism extraction and program restructuring of VHDL for parallel simulation. In *Proc. European Design Automation Conf. (EDAC-93)*, March 1993.

[23] Chih-Po Wen and Katherine Yelick. Portable runtime support for asynchrnous simulation. In *Proc. of the 1995 Int. Conf. on Parallel Processing*, volume 2, pages II–196 – II–204. The Pennsylvania State University, 1995.

[24] Neil Weste and Kamran Eshraghian. *Principles of CMOS VLSI Design.* Addison Wesley Publishing Company, 1992.

[25] John C. Willis and Daniel P. Siewiorek. Optimizing VHDL compilation for parallel simulation. *IEEE Design and Test of Computers*, pages 42 – 53, September 1992.