Actor Based Parallel VHDL Simulation Using Time Warp

Venkatram Krishnaswamy **Prithviraj Banerjee**
Center for Reliable and High-Performance Computing
University of Illinois
Urbana, Illinois 61801
{venkat,banerjee}@crhc.uiuc.edu

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.

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 [1]. 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.

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.

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.

In this paper, we describe the design and implementation of a Time Warp based [2] parallel simulator for VHDL descriptions, which we call `ProperVHDL`. The simulator has been implemented on top of the ProperCAD II libraries [3] for providing an actor based [4] model for concurrent object oriented programming. 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. We have simulated actual VHDL models, and the results indicate that this approach is sound.

The remainder of the paper is organized as follows. Section 2 describes some related work. In Section 3, we provide a brief overview of VHDL. Section 4 is an overview of the ProperCAD II environment. Section 5 contains a description of `ProperVHDL` and its implementation within ProperCAD II. Preliminary results from our first set of experiments are reported in Section 5.

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 [5] 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 McBrayer use the QUEST VHDL simulator based on Time Warp to investigate com-

```
h_adder : process ( a_0, b_0 )
begin
  s_out <= a_0 xor b_0 after tpd ns;
  c_out <= a_0 and b_0 after tpd ns;
end process        ;
```

Figure 1: A VHDL process describing a half adder

bination of processes to increase the computation grainsize [6]. Willis and Siewiorek mention the Auriga system in [7], 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.* [8] have built a conservatively synchonized VHDL simulator based on the Chandy-Misra [9] algorithm. Vellandi and Lightner [10] describe a SIMD algorithm for parallel VHDL simulation and use compilation techniques for extracting parallelism from the source VHDL description. Wen and Yelick [11] use a library based runtime system to construct a parallel circuit simulator. Bagrodia *et. al.* have written the Maisie language for describing parallel simulations [12] and this has been used to implement a gate level logic simulator described by Cong *et. al.* in [13]. There has been a great deal of related work in parallel logic simulation, which has been surveyed by Bailey, Briner and Chamberlain in [14]. Much of the preliminary work in parallel logic simulation and its parallelization using asynchronous algorithms was first reported by Soule [15].

3 VHDL

The VHSIC Hardware Description Language (VHDL) was introduced as a standard by the IEEE [1] in 1987. It is a language intended for documentation and simulation of digital systems. It provides support for both behavioral and structural modeling of digital hardware. Hardware may be modeled in a hierarchical manner. When the hierarchy is flattened out, a VHDL description can be thought of as set of concurrent *processes* exchanging information through timestamped messages sent over a netlist of *signals*. While the processes execute in parallel, execution within them is serial. An example of a process statement is shown in Figure 1. The semantics of process execution is identical to that of discrete event systems, with processes looking up event queues, executing the events if any, and scheduling further events for the future.

4 ProperCAD II System Overview

Much of the work in parallel CAD reported to date suffers from a major limitation in that these proposed parallel algorithms are designed with a specific underlying architecture in mind. As a result, these applications perform poorly on architectures other than the one for which they were designed. Just as importantly, incompatibilities in programming environments make it difficult to port these programs across different parallel architectures. This limitation has serious consequences, since a parallel algorithm needs to be developed afresh for every target MIMD architecture.

One of the primary concerns of the ProperCAD project [16] is to address this portability problem by designing algorithms to run on a range of parallel machines including shared memory multiprocessors, distributed memory multicomputers, and networks of workstations. ProperCAD II is a C++ library which provides an object–oriented parallel interface based on the actor model of concurrent object–oriented computing [3].

The fundamental object in the actor model is the actor, an object which communicates with other actors via messages. All actions an actor takes are in response to messages. The actor model lacks explicit sequencing primitives. Synchronization is implicit and derives from the single threaded nature of individual actors. Since an actor cannot suspend execution implicitly in the middle of a computation, continuation passing style (CPS) [17] is used to express control and data dependences.

The Actor/CPS model extends the RPC model to support object-oriented programming and parallelism. Functions are invoked with respect to an actor. The call of an actor method is non-blocking and consequently actor methods do not return values. In Figure 2, the `f` member of actor `a` calls the `g` member of `b` but does not wait. Instead, a *continuation* is passed as an argument to `g`. The continuation that `f` passes, `a.f'`, specifies that the object `a` and the method `f'` will process the value returned by `g`. In this case, `f'` will be that code in the RPC `f` which is dependent on the return value of `g`. `g` treats the continuation `c` as a function pointer; once the return value `x` is computed, `g` calls `c` with that value. Continuation variables are the parallel, object-oriented extrapolation of serial, procedural function pointers.

By mapping multiple actors onto a physical processor, the runtime system on each processor is able to schedule the ready-to-use actors on the processor and obtain high utilization by overlapping computation with communication.

As part of ProperCAD, a suite of parallel applications have been developed, as shown in Figure 3, that address the most significant tasks in VLSI design automation including circuit extraction [18], test generation [19] , fault simulation [20], cell placement [21], and logic synthesis [22].

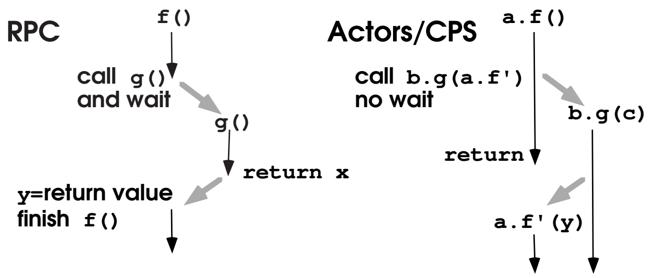

Figure 2: RPC and Actors/CPS.

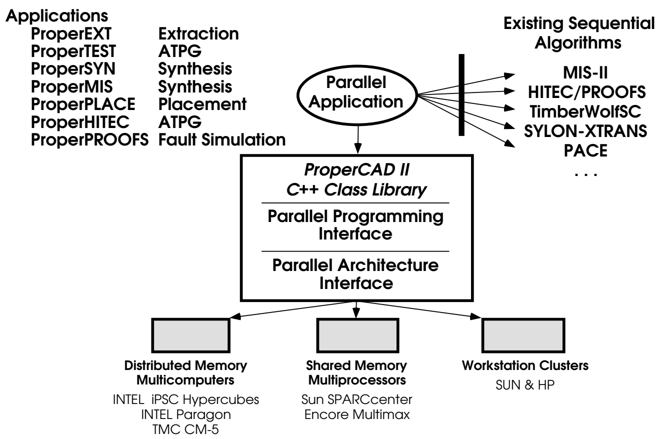

Figure 3: An overview of the ProperCAD project.

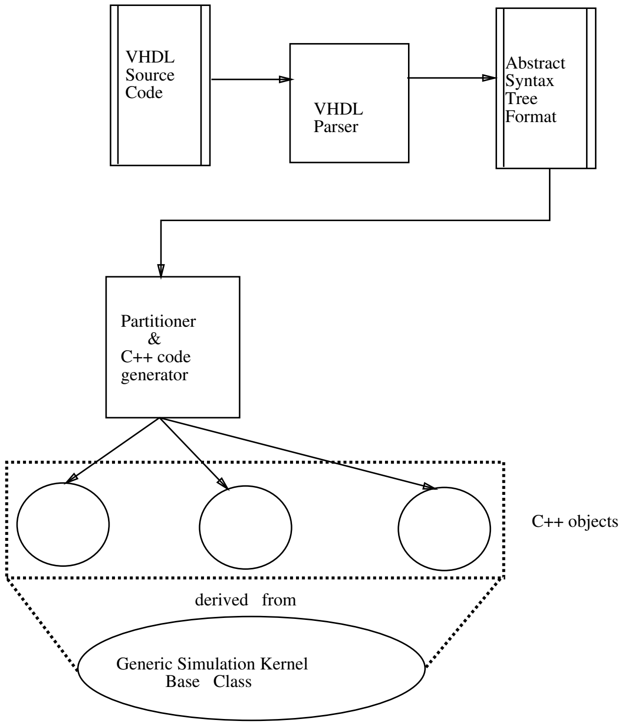

Figure 4: Overview of ProperVHDL.

5 ProperVHDL overview

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. 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 the forthcoming section. See Figure 4 for a pictorial depiction of the simulator.

5.1 Simulation Kernel

The simulation kernel provides the means for the execution of the simulation cycle as defined by the VHDL Language Reference Manual (LRM)[1]. 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.

5.1.1 Mapping Logical Processes onto Actors

Each `process` in the source VHDL is translated into a C++ class. Each such class is derived from the `VHDLActor` class. The `VHDLActor` actor class is, in turn derived from the `Actor` class. The `Actor` class is a base class for all actors, provided by the ProperCAD II library. In other words, each `process` in the source VHDL is represented by an actor. The kernel methods and data structures referred to above are provided within the `VHDLActor` class.

The source VHDL is first flattened out into a set of process statements. The next step is translating each of the processes into a C++ class. To achieve this, we have written a code generator, which is described in the next section. For each process, a header file containing the declarations for data structures required by the process is generated. There is also a file containing methods for the class, namely the constructor, the destructor and the methods for executing the behavior embodied in the `process` statement.

The `VHDLActor` class has a virtual method called `executeProcess`. 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

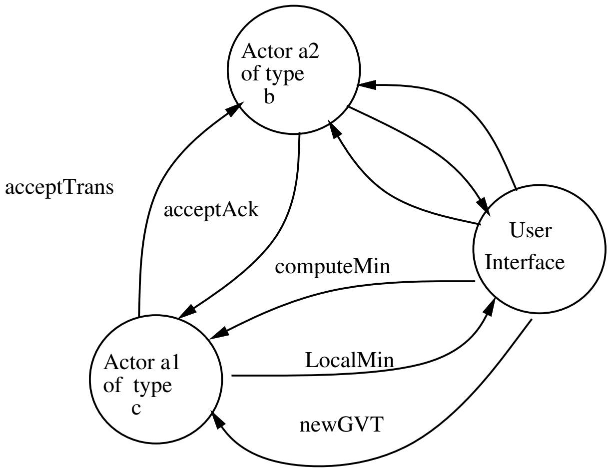

Figure 5: LP `a1` invokes the `acceptTransaction` continuation on LP a2, which responds by invoking the `acceptAckMsg` continuation on `a1`. The UserInterface is shown sending and receiving GVT request continuations.

change has occurred in the *sensitivity list* of the `process`.

5.1.2 Communication between Logical Processes

Logical Processes (LP) 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.

The sending of an event from one LP to another is accomplished by invoking the `ActorMethod` on the receiver with appropriate parameters. In `ProperVHDL` the `ActorMethod` responsible for receiving events is called `acceptTransaction`. Similarly, `ActorMethods` exist for handling receipt of anti messages, requests for participation in GVT computation, and broadcasts of newly computed GVT. Sometimes, it may be necessary for the receiver to respond to a message by invoking an `ActorMethod` on the sender. In this case the continuation is included in the original message. The receiver has to merely execute it. An example of this situation occurs when using Samadi's algorithm for GVT [23] computation, in which the receipt of every event must be acknowledged by the receiver. Figure 5 shows some of the interactions which occur between LPs in terms of the continuations invoked upon one another.

5.1.3 Continuation Table

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 table has been implemented as an `Aggregate` . This is a distributed data structure. There is a copy of the table on each processor, and each actor looks up the copy on its local node. There is therefore no extra overhead incurred in looking up the continuation table.

5.1.4 GVT Computation

For any realistic application of the Time Warp algorithm, it is necessary to pay attention to computing the *Global Virtual Time* or GVT, in order to reclaim state space which is no longer required. GVT is the least simulation time to which a process may roll back. Any state saved at a time prior to this time may be reclaimed[1]. GVT is also essential for commitment of output. There have been several algorithms proposed for computing GVT including [23], [24], [25].

As of now, a centralized GVT management algorithm, Samadi's algorithm is being used. We have chosen this algorithm because of the simplicity of its implementation. 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. It should be noted that the algorithm is not particularly efficient in terms of the message traffic it requires. This is because each message sent by a process must be acknowledged by the sender. On the other hand, it is very easy to compute, given the minima from each LP. Other distributed algorithms which are more efficient in terms of messages have been proposed [24], [25]. However, they are computationally more expensive than Samadi's algorithm. With the declining cost of communication (particularly on shared memory machines), this is a tradeoff which deserves some attention.

The algorithm is shown in Figure 6. The computation of GVT involves finding the minimum simulation time to

[1] In actual fact, one state before GVT is maintained to enable correct handling of rollbacks to GVT.

```
Procedure LocalMinimum:

mode = find;

min_1 = min. timestamp of unacked msgs;

min_2 = min. timestamp of tagged ack msg;

min_3 = local_time;

min_p = MIN ( min_1, min_2, min_3 );

report min_p to manager;
```

Figure 6: Samadi's Algorithm for GVT Computation used in ProperVHDL

which a process may rollback. Lin [24] has noted that the two primary difficulties in this computation are tracking transient messages and the simultaneous reporting problem. Samadi's algorithm solves the former by having message acknowledgement. The second problem is tackled by defining two states of execution of an LP – a normal state, and a state which is reached when a request arrives to compute GVT. The normal state is reached from the latter when a value of GVT is reported by the GVT Manager. Any acknowledgement sent out while a process is in the second state is tagged to denote this fact. Then, the local value of GVT reported is the minimum timestamp of an unacknowledged message, the minimum timestamp on a tagged acknowledgement and the local clock. Computation of the first value is accomplished by searching the output queue. The second quantity decreases monotonically, so it is sufficient to maintain a running minimum. The central manager computes GVT to be the minimum of all the local minima.

5.1.5 Other Time Warp Parameters

The simulator does not use bounded time windows to limit the optimistic execution. We utilize an aggressive cancellation policy to recover from rollbacks. State is saved after every event.

5.2 Code Generation

This section describes some aspects of C++ code generation from VHDL. The front end VHDL analyzer has been implemented using the PCCTS [26] tool for generation of LL(k) parsers. The output of the front end is an abstract syntax tree and a set of symbol tables. A code generator accepts these as input and traverses the tree to generate C++ code.

As mentioned briefly before, each VHDL `process` is translated into a C++ class. This involves generation of a

Original VHDL process

```
a: process
  begin
  ....
  x <= val after 5 ns;
  ....
  end process;
```

Translation

```
Actor_a.H
class Actor_a{
  Definitions of
  Data Structures
};

Actor_aeP.C
 Actor_a::Actor_a()
 {
 ....
 }

 Actor_a::~Actor_a()
 {
 ....
 }

 Actor_a::executeProcess()
 {
 ........
  assignSignal(...);
 ........
 }
```

Figure 7: Code generation for a VHDL process.

header file containing the data structures for each class. In addition a `.C` file containing the constructor, destructor and methods carrying out the behavior embodied in the original VHDL `process` is generated. This is shown in Figure 7.

VHDL constructs such as signal assignments, variable assignments and wait statements are made by calls to methods implementing their functionality in the `VHDLActor` base class. Conditional and looping constructs are directly translated into C++ . In order to correctly handle VHDL code with wait statements arbitrarily interspersed in the source VHDL it was necessary to perform certain transformations which are described below.

Wait statements entail suspension of a process statement until a condition is satisfied, timeout occurs, or event [2] 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 [4].

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 8. Resumption is then implemented simply by calling the enabled method. A simi-

[2] 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*.

Source VHDL

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

Generated Code

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

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

Figure 8: Translation of a VHDL process with a wait statement in the middle of the code

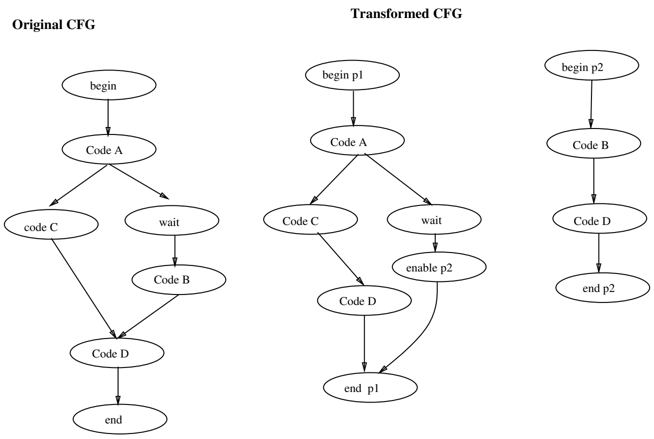

Figure 9: Generating code for a wait statement in a conditional construct.

lar notion was first reported in [27] 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 9 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 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 10. 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

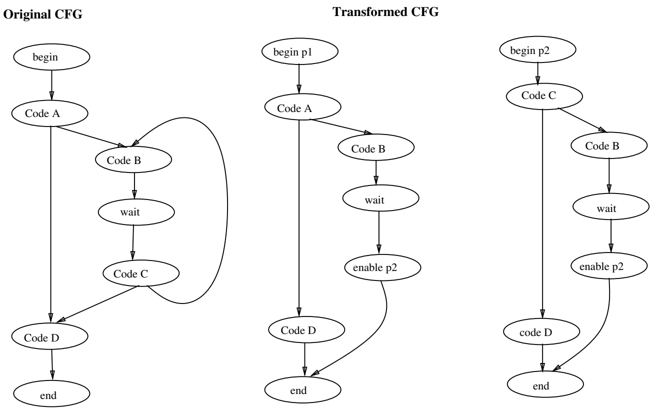

Figure 10: Generating code for a wait statement in a looping construct.

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

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.

6 Experimental Results

We now describe some experiments and preliminary results. Initial experiments were performed using two VHDL descriptions. These are 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.

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 processors upto 4 processors and then a deterioration occurs. On the Paragon, however, the runtimes scale poorly upon increasing the number of pro-

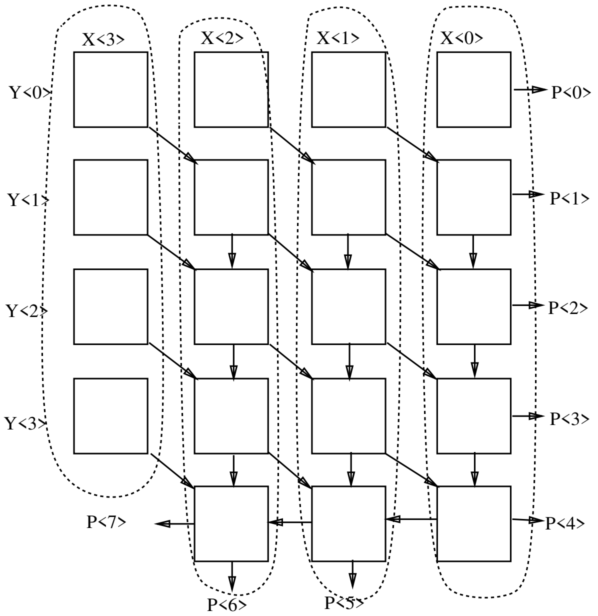

Figure 11: 4x4 bit array multiplier partitioned to run on 4 processors
.

cessors.

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 [28] to the multipliers. This yielded the partition shown in Figure 11. 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 based on a circuit partitioning strategy improves the scalability of the simulation i.e. the overhead of communication does not overcome the benefits of utilizing additional processors.

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. Using a larger number of processors relieves the thrashing problem.

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

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

We have also run some experiments with the ISCAS [29, 30] set of benchmarks, which are shown in Tables 5 and 6. The simulations on the Sparcserver show good scaling. However, serious memory difficulties prevented us from gathering more conclusive data on the Paragon.

7 Conclusions and Future Work

We have described the implementation of a VHDL simulator using the Time Warp algorithm in the context of Actor based parallelism. Furthermore, our preliminary results show that placement of processes based on a partition of the circuit topology leads to improvements in runtime and scalability. In the future, we intend to run simulations on a wider set of benchmark VHDL descriptions to see whether these partitioning strategies will continue to yield favorable results for less regular circuit structures.

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	-	-	-	-	-

References

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

[2] D. Jefferson, "Virtual time," *ACM Transactions on Programming Languages and Systems*, vol. 7, no. 3, pp. 404–425, 1985.

[3] S. Parkes, J. A. Chandy, and P. Banerjee, "A library based approach to portable, parallel, object–oriented programming: Interface, implementation and application," in *Supercomputing '94*, November 1994.

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

[5] Jade Simulations Int. Co., "Implementation issues of Jade's VHDL simulator," tech. rep., Calgary, Alberta, Canada, 1989.

[6] T. J. McBrayer and P. A. Wilsey, "Process combination to increase event granularity in parallel logic simulation," in *Proceedings of Int. Parallel Processing Symposium*, April 1995.

[7] J. C. Willis and D. P. Siewiorek, "Optimizing VHDL compilation for parallel simulation," *IEEE Design and Test of Computers*, pp. 42 – 53, September 1992.

[8] K. L. Kapp, T. C. Hartrum, and T. 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*, pp. 78 – 85, June 1995.

[9] K. M. Chandy and J. Misra, "Asynchronous distributed simulation via a sequence of parallel computations," *Communications of the ACM*, vol. 24, pp. 198 – 206, April 1981.

[10] 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.

[11] C.-P. Wen and K. Yelick, "Portable runtime support for asynchrnous simulation," in *Proc. of the 1995 Int. Conf. on Parallel Processing*, vol. 2, pp. II–196 – II–204, The Pennsylvania State University, 1995.

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

[13] J. Cong, Z. Li, and R. Bagrodia, "Acyclic multiway partitioning of boolean networks," in *31st ACM/IEEE Design Automation Conference*, pp. 670 – 675, June 1994.

[14] M. L. Bailey, J. V. Briner, and R. D. Chamberlain, "Parallel logic simulation of VLSI systems," *ACM Computing Surveys*, vol. 26, pp. 255–294, Sept. 1994.

[15] L. Soule and T. Blank, "Parallel logic simulation on general purpose machines," in *Proc. 26th Descign Automation Conf.*, pp. 81–86, June 1989.

[16] B. Ramkumar and P. Banerjee, "ProperCAD: A portable object-oriented parallel environment for VLSI CAD," *IEEE Trans. Computer-Aided Design*, vol. 13, pp. 829–842, July 1994.

[17] A. W. Appel, *Compiling with Continuations*. Cambridge: Cambridge University Press, 1992.

[18] B. Ramkumar and P. Banerjee, "ProperEXT: A portable parallel algorithm for VLSI circuit extraction," in *International Parallel Processing Symposium*, pp. 434 – 438, 1993.

[19] S. Parkes, P. Banerjee, and J. H. Patel, "ProperHITEC: A portable, parallel, object–oriented approach to sequential test generation," in *Proceedings of the Design Automation Conference*, pp. 717–721, june 1994.

[20] S. Parkes, P. Banerjee, and J. Patel, "A parallel algorithm for fault simulation based on proofs," in *International Conf. on Computer Design*, (Austin, TX), Oct 1995.

[21] S. Kim, J. A. Chandy, S. Parkes, B. Ramkumar, and P. Banerjee, "ProperPLACE: A portable parallel algorithm for cell placement," in *International Parallel Processing Symposium*, (Cancun, Mexico), pp. 932–941, April 1994.

[22] K. De, J. A. Chandy, S. Roy, S. Parkes, and P. Banerjee, "Portable parallel algorithms for logic synthesis using the MIS approach," in *International Parallel Processing Symposium*, (Santa Barbara, CA), pp. 579–585, April 1995.

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

[24] Y. B. Lin and E. D. Lazowska, "Determining the global virtual time in a distributed simulation.," in *Proceedings of International Conf. on Parallel Processing*, pp. 201 – 209, 1990.

[25] L. M. D'Souza, X. Fan, and P. A. Wilsey, "pGVT: An algorithm for accurate gvt estimation," in *Proc. of 8th Workshop on Parallel and Dist. Simulation*, pp. 102–109, June 1994.

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

[27] 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*, (Knoxville, TN), pp. 518–525, May 1994.

[28] Y. Levendel, P. Menon, and S. Patel, "Special purpose computer for logic simulation using distributed processing," *Bell System Tech. J.*, vol. 61, pp. 2873–2910, Dec. 1982.

[29] 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*, vol. 3, June 1985.

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