ADVISE!
Performance Evaluation of Parallel VHDL Simulation

Wilco van Hoogstraeten Henk Corporaal

Delft University of Technology
Department of Electrical Engineering P.O. Box 5031, Mekelweg 4
2600 GA Delft, The Netherlands
e-mail: chess@duteca.et.tudelft.nl

July 27, 1994

Abstract

VHDL is the one of the most important and widely used hardware description languages at this time. The increase in size and complexity of VHDL models necessitates the use of parallel and distributed algorithms to obtain acceptable simulation performance.

We have investigated the use of optimistic distributed algorithms with VHDL simulation. Using an optimistic algorithm with VHDL introduces some major problems which can only be solved at some cost.

With our simulation environment we obtain speedups of around four for a medium-sized benchmark. We expect that the optimisation of our environment and partitioning, and the use of larger benchmarks will lead to higher speedups, which makes it worthwhile to investigate this approach further.

1 Introduction

The design of todays complex digital systems requires the use of a hardware description and simulation language. VHDL has emerged as one of the most widely used hardware description languages, and has become a world-wide accepted standard.

The rapid increase in model size has led to a slow simulation throughput for VHDL. This has led to the investigation of parallelising VHDL simulation, both to shared and distributed memory architectures.

Our research has focussed on the use of optimistic distributed algorithms for the parallelisation of VHDL simulation. Algorithms with optimistic synchronisation promise most inherent parallelism, but at a cost. This work shows the speedup that can be obtained by parallelising our VHDL simulation framework.

1.1 VHDL

VHDL was developed by the Department of Defense as a standard hardware description and simulation language to be used in government projects. The syntax of VHDL is largely based on Ada, but in contrast with Ada VHDL has become a world-wide standard for description, simulation, and synthesis, with a market share of roughly 40%.

The important parts of VHDL, especially for simulation, are *processes* and *signals*. Processes model hardware parts that execute concurrently, while signals are used to connect them. Figure 1 shows a simple VHDL circuit containing processes and signals. Processes and signals are easiest understood by comparing them with gates and the wires connecting them.

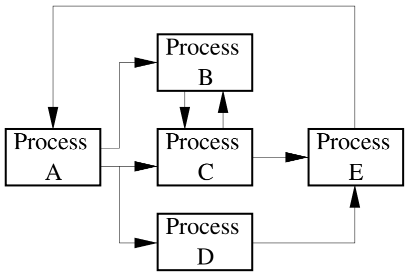

Figure 1: Processes and signals in a VHDL circuit.

However, processes and signals are much more powerful than could be expected from a *hardware* description language. Within a process, the code that represents the function is executed sequentially, and almost the complete functionality of Ada can be used. That means that VHDL has a powerful and strict type system, all types of control flow, functions and procedures, recursion, and dynamically allocated memory. Signals can be made to carry almost any VHDL type, including floating point types, and array and record types. The result is a very powerful language that is on a par with many general purpose programming languages.

The power of VHDL introduces a host of problems for simulators, and especially optimistic distributed simulators. The state of a VHDL process can be very complex due to the availability of recursive functions, and because it can allocate dynamic memory. All this means that an optimistic simulator that can handle the full functionality of VHDL must be able to handle these complex issues.

1.2 Related work

Many of the problems encountered in distributed simulation of VHDL can be illustrated nicely by the studying of previous work. They show that people have set out with different goals and expectations, yielding varied results.

Wilsey A team at the University of Cincinnati, headed by Philip Wilsey, has been researching optimistic VHDL simulation for some time now [2]. However, their research does not seem directed at full VHDL simulation, but at logic simulation, with VHDL as the input language. From their work it seems that only the low abstraction levels of VHDL are used; this means that important optimisations can be made, which lead to high speedup [12].

Willis IBM and Carnegie Mellon University have been researching parallel VHDL simulation, led by John Willis. Their approach has been to compile the VHDL as efficiently as possible, while using a simple parallel, dead-lock free simulation algorithm [11]. This work shows that by optimising the compilation process, a large acceleration can be obtained. This applies of course to both parallel and sequential simulation. Similar work has been done at the University of Colorado [9].

This research shows that for fast VHDL simulation compilation techniques have to be optimised. However, the focus has not been on new simulation algorithms.

Ashenden A team at the University of Adelaide has also been active in parallel VHDL simulation, headed by Peter Ashenden; work on this project has however recently been abandoned, due to financial reasons. Because of this, their research has never gotten past a centralised-time implementation [1].

Faraboschi At the University of Genoa a team including Paolo Faraboschi has investigated special hardware for the parallel simulation of VHDL [3]. Their approach was to make a processor for a subset of VHDL and a state-driven approach, i.e. every simulation cycle the complete circuit is re-evaluated. Although their results for simple benchmarks were excellent, the possibilities of the system were too small to grant further investigation, and research has been abandoned.

1.3 Organisation

First our approach of the problem is described in section 2. Then the implementation of the optimistic distributed simulation algorithm is described in section 4. Then the results that have been obtained with our framework are shown in section 5. Our conclusions are in section 6.

2 ADVISE!

The project has adopted the simple and logic model that has been used many times before. Figure 2 shows the outline of the process: VHDL source is read in by a compiler/partitioner that compiles the VHDL to an intermediate language and performs a static partitioning. Each of these partitions, after compilation and linking with the parallel simulation kernel, are run on a separate processor in the network (only one shown).

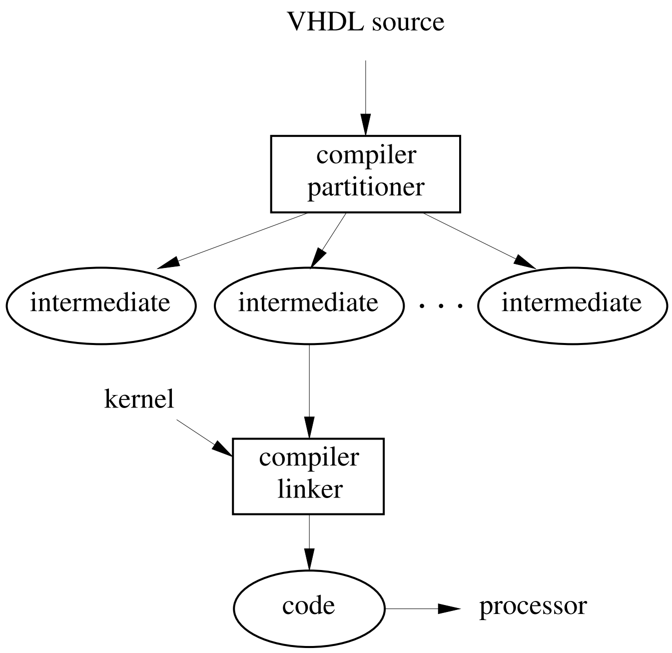

Figure 2: The VHDL to network code trajectory.

This method, especially the static partitioning, is dependent on the topology of the network and the hardware of the processors. This poses a problem, because one of the fundamental intentions of the project is to make tools that are as independent of these issues as possible. This problem was solved in two ways: different tools would be created, more or less analogous with the squares in the figure, and the processors and network would be simulated by a special tool. This last tool, a network simulator, would allow testing of the simulator on different topologies and allow complete control of the network characteristics, such as bandwidth. By making the kernel a separate module as well, it would also be possible to exchange simulation algorithm quickly, without recompiling the VHDL.

Finally the structure shown in Figure 3 emerged; the tools depicted by the square boxes had to be developed. This structure has two advantages; the primary use is to predict performance and

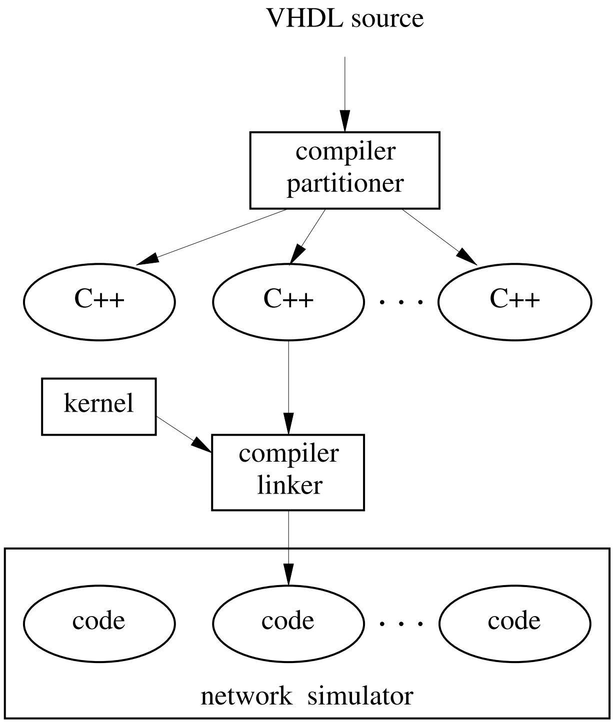

Figure 3: Our approach.

compare the performance of different network-algorithm combinations, but it can easily be ported to a real system because all the necessary parts are there. A disadvantage is that optimisations for special networks and configurations that can boost performance considerably cannot be applied; however, impact of simulation algorithms and optimisations can be studied on a more general level.

Several considerations had impact on the choice of the intermediate language:

semantic gap One of the aims of the project was to have a simple compiler that could be implemented in a minimum amount of time. It was therefore necessary to make the semantic gap between VHDL and the intermediate language as small as possible. This notion seems to be validated by Hueber et al. [5] who show that existing VHDL compilers perform very poorly, probably due to the fact that the semantic gap they try to bridge is very large; this makes massive restructurings necessary, taking both time and space.

Because VHDL is very similar to Ada, this would be seem to be the best choice. However, lack of Ada experience and the absence of a solid Ada compiler made the advantages of using Ada questionable. Another problem is that Ada has the same semantic level as VHDL, which means that performance would be very dependent on the quality of the Ada compiler.

Another option was to use C++ and raise the semantic level of C++ by writing a set of support classes that make C++ behave very much like VHDL. This looked a better option that using Ada.

availability An extra incentive to use C++ was the availability of a network simulator working with C++. Together with the availability of a solid reliable compiler made this point weigh heavy in the favour of using C++.

experience The experience of the primary participants was strongly focussed on C, which would accommodate the step up to C++ very nicely.

The above makes it clear that ultimately C++ was chosen as intermediate language for the compiler.

The semantic gap between VHDL and C++ was decreased to an enormous extent by writing a set of classes that are very similar to VHDL objects and datatypes. This set of classes is called the RunTime Environment (RTE) and has made the compilation process much simpler. The RTE contains classes for VHDL basic types like `INTEGER`, `REAL`, and `TIME`, composite classes for `ARRAY` and `RECORD` types, classes for signal types, and classes for VHDL processes. The added bonus of developing a solid interface like the RTE was the possibility of concurrent development that would arise after the completion of the interface description. With time and resources low, this was a very important issue.

2.1 The Compiler/Partitioner

As was mentioned earlier, one of the aims of the RTE design was to minimalise the complexity of the compiler. Especially for sequential simulation this is the case. Compilation for the optimistic kernel is a little bit more difficult, and code generation is subject to stricter constraints; mostly for reasons that involve visibility of data to the state saving mechanism.

Partitioning is another issue that complicates the compiler. If partitioning is integrated in the compiler, the most efficient code can be generated. However, because of difficulties already encountered which stretched the implementation time of the compiler, it was decided to split the partitioner from the compiler. This would bring the complexity of the compiler down, while not much extra had to be done.

2.2 Structure of the compiler

From the beginning it was not realistic to write the front-end of the VHDL compiler ourselves. Even if all syntax and static semantic checking is abandoned, then there is still much work in providing overloading resolution for functions and operators. Fortunately an industrial strength front-end is commercially available: CLSI[1] has a front-end that does all possible static checking and an object-oriented library to walk the resulting Abstract Syntax Tree (AST).

[1] Cad Language Systems Inc.

The back-end of the compiler consists of two parts: one part generates the C++ code for the VHDL process bodies, while the other part generates a set of description files for the partitioner. These description files contain a description of every VHDL entity, including its inputs and outputs and processes that are called.

Figure 4 shows a graphic representation of the complete compiler. Also shown is the partitioner that reads in the description files and generates the code that starts up the VHDL process code on the right processor.

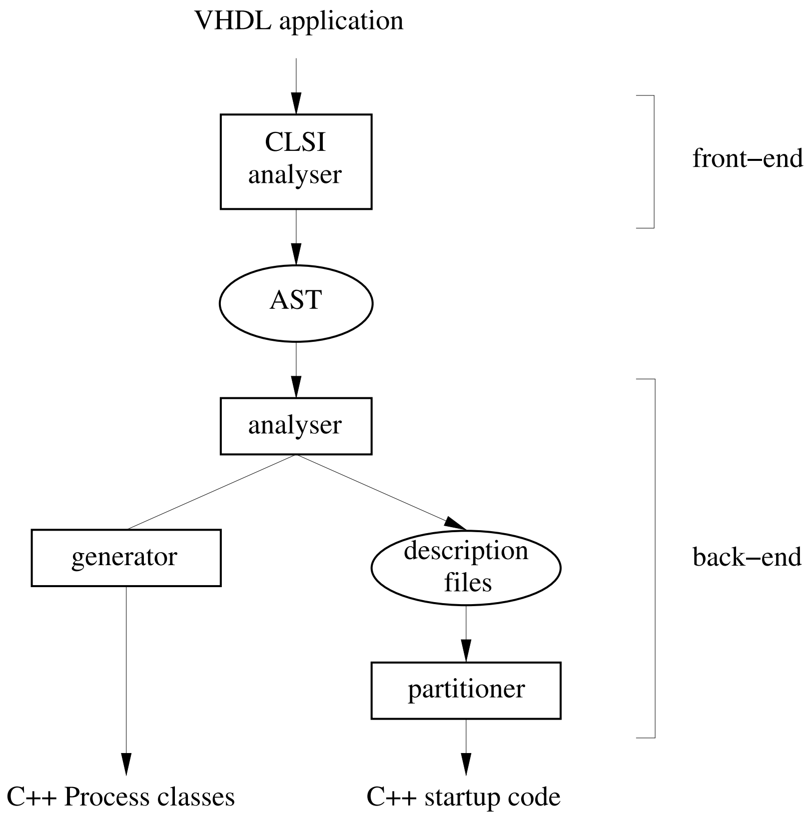

Figure 4: From VHDL to C++.

An additional advantage of the split between the compiler and partitioner is the independence of the code generation from the partitioning. A VHDL description needs only be compiled once, while multiple partitioning runs can be made on the description files. This is especially useful for performance evaluation use: benchmarks only have to be compiled once, while multiple test runs with different partitions can be run without recompiling (compiling for network simulation is time-consuming work).

The partitioner is currently implemented as a simple Kernighan and Lin type partitioner [7]. The flattened VHDL circuit is partitioned in n partitionings, where n is a power of 2.

2.3 The Runtime Environment

The runtime environment is a set of C++ classes and library functions that behave like the basic VHDL construct and imperatives. The RTE has three important characteristics:

- Translation from VHDL to C++ becomes relatively easy,

- Its interface allows concurrent development of compiler and simulator,
- A flexible and elegant environment like the RTE allows easy modification, enhancement, extension, and optimisation.

An example of the simplicity of the code generation is shows by the next example. The following piece of VHDL:

```
architecture simple of Nor_Gate is
begin
   process
   begin
      y <= a nor b after 10 fs;
      wait on a, b;
   end process;
end simple;
```

can be translated into the following C++:

```
int Nor_Gate::main( )
{
   while ( TRUE )
   {
      y.waveform_assign( a.nor(b), 10 );
      wait_on( a, b );
   }
}
```

As can be seen from this example, the RTE allows a very simple compiler that in most places can translate statements and variable declarations on a one-to-one basis.

The four most important classes in the RTE are:

`Value` This class represents all the basic types in VHDL, such as `integer` and `time`.

`Signal` This class represents the VHDL signals, which model physical wires and are the only way to communicate between two entities.

`Driver` This class represents the VHDL notion of a driver, which is not a part of the language visible to users, but is needed to specify the simulation results of VHDL waveform assignments. Drivers are an official part of the VHDL standard [6]. Drivers are necessary for the correct propagation of signal values.

`Process` This class represents the VHDL processes, which are the "working" parts of VHDL. The body of a VHDL process will become the working code inside a `Process` instantiation.

Virtually all the functionality of VHDL is hidden in these classes and their member functions.

2.4 The Network Simulator

The network simulator was originally developed outside the VHDL simulation project, but has been rewritten for it. Besides the simulation of the parallel processors, the network simulator also simulates the network. It is possible to use different network topologies and many configuration options are available; an extensive description of the capabilities and internals of the network simulator can be found in Olk [8].

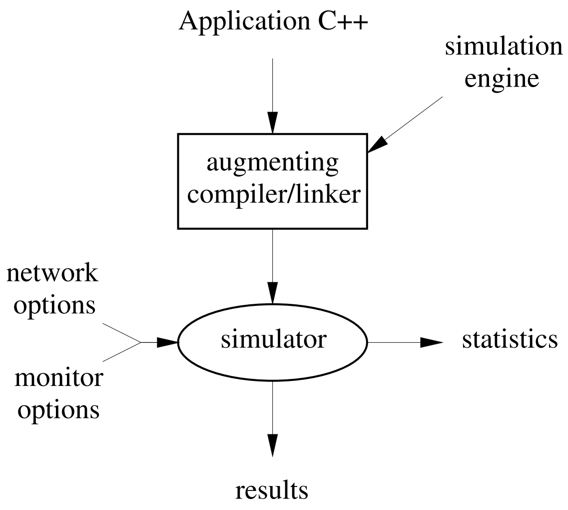

Figure 5: Data flow in the network simulator.

The simulator uses the machine code of the host machine to run the simulation. This is significantly faster than emulating or interpreting non-native code, and is accurate enough when using *code augmentation*: extra code is added to the native code to count the amount of instructions executed [10]. The code augmentation is done by a compiler specially tweaked for this purpose.[2]

Figure 5 shows the general outline of the network simulator. The C++ code of the application is compiled by the augmenting compiler and linked to the simulation engine of the network simulator. This leaves an executable simulator that on invocation reads in a set of options set by the user describing the network characteristics (topology, latency, bandwidth, etc.) and statistics generation options (number of messages, message size, link utilisation, etc.). During the run the simulator outputs the statistics that it has gathered next to the output generated by the application code.

The network simulator is, all in all, nicely versatile and has some features that are unavailable in other such simulators (pressed by the need of the VHDL simulator), one of which is the possibility to run multi-threaded code on each simulated processor. This was very important, because our VHDL simulator is itself multi-threaded.

3 Parallel Simulation Options

This section gives an overview of different methods for VHDL simulation, and motivates our choice to develop a new VHDL accelerator. The simulation methods can be classified as follows:

[2]The compiler in question is the GNU C/C++ compiler.

- Centralized time simulation. In this simulation method, every partition of the circuit is evaluated concurrently by a different processor. The processors operate in a synchronous way, the virtual simulation times of all processors are equal.

- Conservative distributed time simulation. This method is more general than the previous one, in the sense that every processor has its own virtual simulation time. These virtual times do not have to be equal. Generally, when we look at a simulation of a circuit, the processors that evaluate the elements at the input part of the circuit will be further ahead than the processors that evaluate the elements at the output part of the circuit. The simulation is called *conservative* because only correct events are generated.

  In conservative simulation *deadlock* can occur when no single processor can continue. Deadlock avoidance and deadlock detection and resolution schemes are therefore of prime interest for these algorithms.

- Optimistic distributed time simulation is a further improvement, because it allows speculative execution of events. When a processor is finished handling all it completely known events, it generates new events based on incomplete information of its inputs. As soon as all inputs are known, the processor verifies the correctness of the generated events. If these turn out to be correct, time has been saved. If the generated events are incorrect, then the processor rolls back to an earlier virtual time and re-evaluates the events.

  Besides this, deadlock cannot occur in optimistic simulation. This is another important point in its favour.

At first sight it may seem that optimistic distributed time simulation is more expensive because it creates incorrect events which have to be corrected, leading to an avalanche of corrective actions. This intuition is wrong. Speculative evaluations are performed only when the same processor would be waiting for additional inputs in the conservative simulation, so no time is wasted. Other studies using optimistic algorithms report good results.

The application of optimistic algorithms to VHDL simulation is new. Results from the CHESS-1 project have already shown promising results. The circuits shown in 6 and 7 serve as vehicles for demonstrating in more detail why optimistic algorithms perform better than their conservative counterparts. The first circuit shows a stratified circuit that has no cycles, these

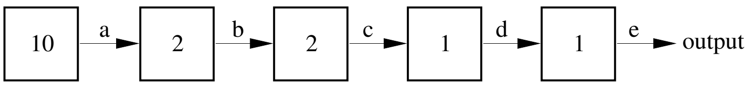

Figure 6: A stratified circuit (no cycles)

circuits are quite common and can be found in the form of digital signal processors for example. The second circuit is similar to the first one, except that a feed-back signal has been added. We want to stress that these circuits are tiny, and only serve as a means of demonstrating the principles of distributed simulation. In real-life circuits, with much larger size and complexity, the difference between the various simulation methods will be much larger.

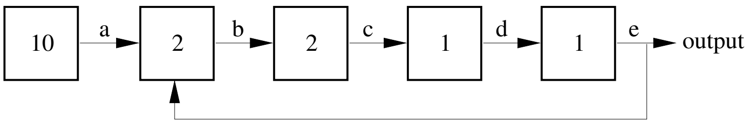

Figure 7: A circuit with a cycle

3.0.1 Centralized time simulation

Table 3.0.1 shows the simulation of the above circuit using a centralized time simulation algorithm. The circuit is divided in 5 elements, every element is evaluated on a dedicated processor. These processors are numbered P_1 to P_5, the first column of the table shows the real time, the time that it takes for the actual simulation to complete. Every row in the table corresponds to a simulation cycle. For the sake of simplicity we assume that every simulation cycle takes an equal amount of time. The expression $\langle$ 10,a, d $\rangle$enotes an event that takes place at virtual time 10, and has an effect on signal a. Table 3.0.1 shows clearly that the events are evaluated sequentially. Although

REAL TIME	P_1	P_2	P_3	P_4	P_5
0	$\langle$ 10, a $\rangle$				
1		$\langle$ 12, b $\rangle$			
2			$\langle$ 14, c $\rangle$		
3				$\langle$ 15, d $\rangle$	
4					$\langle$ 16, e $\rangle$
5	$\langle$ 20, a $\rangle$				
6		$\langle$ 22, b $\rangle$			
7			$\langle$ 24, c $\rangle$		
8				$\langle$ 25, c $\rangle$	
9					$\langle$ 26, e $\rangle$
10	$\langle$ 30, a $\rangle$				
11		$\langle$ 32, b $\rangle$			
12			$\langle$ 34, c $\rangle$		
13				$\langle$ 35, c $\rangle$	
14					$\langle$ 36, e $\rangle$
15	$\langle$ 40, a $\rangle$				
16		$\langle$ 42, b $\rangle$			
17			$\langle$ 44, c $\rangle$		
18				$\langle$ 45, c $\rangle$	
19					$\langle$ 46, e $\rangle$

Figure 8: Centralized time simulation of both circuits

5 processors are used, they never operate concurrently. In other circuits it may be possible that some events happen at exactly the same simulation time, these events will be evaluated in parallel using this simulation method. Note that it does not matter whether there are cycles in the circuit or not. The total simulation will always take 20 units to complete.

3.0.2 Conservative distributed time simulation

Table 3.0.2 shows the results of conservative distributed time simulation applied to the first circuit. We can see the effect of 'pipelining' in this case. The remark made in the introduction that processors further away from the input signals operate in earlier virtual times is demonstrated nicely in this table. The second table (3.0.2) shows the results when the second circuit is simulated

REAL TIME	P_1	P_2	P_3	P_4	P_5
0	$\langle$ 10, a $\rangle$				
1	$\langle$ 20, a $\rangle$	$\langle$ 12, b $\rangle$			
2	$\langle$ 30, a $\rangle$	$\langle$ 22, b $\rangle$	$\langle$ 14, c $\rangle$		
3	$\langle$ 40, a $\rangle$	$\langle$ 32, b $\rangle$	$\langle$ 24, c $\rangle$	$\langle$ 15, d $\rangle$	
4		$\langle$ 42, b $\rangle$	$\langle$ 34, c $\rangle$	$\langle$ 25, d $\rangle$	$\langle$ 16, e $\rangle$
5			$\langle$ 44, c $\rangle$	$\langle$ 35, d $\rangle$	$\langle$ 26, e $\rangle$
6				$\langle$ 45, d $\rangle$	$\langle$ 36, e $\rangle$
7					$\langle$ 46, e $\rangle$

Figure 9: Conservative distributed time simulation of circuit 1

using a conservative algorithm. Because the presence of a cycle, the event $\langle$ 22, b $\rangle$ cannot be calculated in the second simulation cycle, instead processor P_2 has to wait until the results from P_5 up till time 22 are known.

REAL TIME	P_1	P_2	P_3	P_4	P_5
0	$\langle$ 10, a $\rangle$				
1	$\langle$ 20, a $\rangle$	$\langle$ 12, b $\rangle$			
2	$\langle$ 30, a $\rangle$		$\langle$ 14, c $\rangle$		
3	$\langle$ 40, a $\rangle$			$\langle$ 15, d $\rangle$	
4					$\langle$ 16, e $\rangle$
5		$\langle$ 22, b $\rangle$			
6			$\langle$ 24, c $\rangle$		
7				$\langle$ 25, c $\rangle$	
8					$\langle$ 26, e $\rangle$
9		$\langle$ 32, b $\rangle$			
10			$\langle$ 34, c $\rangle$		
11				$\langle$ 35, c $\rangle$	
12					$\langle$ 36, e $\rangle$
13		$\langle$ 42, b $\rangle$			
14			$\langle$ 44, c $\rangle$		
15				$\langle$ 45, c $\rangle$	
16					$\langle$ 46, e $\rangle$

Figure 10: Conservative distributed time simulation of circuit 2

3.0.3 Optimistic distributed time simulation

Table 11 shows the results that may be generated by an optimistic distributed time algorithm. We can see that processor P_2 emits an event at time 22, although the event on the feedback signal has not arrived yet. The processor assumes that this signal will not affect signal b. This turns out to be correct. The event $\langle\ 36,\ e\ \rangle$ has an effect that was not predicted correctly by P_2 and causes a roll back, the incorrect events are redistributed in the system.

REAL TIME	P_1	P_2	P_3	P_4	P_5
0	$\langle$ 10, a $\rangle$				
1	$\langle$ 20, a $\rangle$	$\langle$ 12, b $\rangle$			
2	$\langle$ 30, a $\rangle$	$\langle$ 22, b $\rangle$[a]	$\langle$ 14, c $\rangle$		
3	$\langle$ 40, a $\rangle$	$\langle$ 32, b $\rangle$	$\langle$ 24, c $\rangle$	$\langle$ 15, d $\rangle$	
4		$\langle$ 42, b $\rangle$	$\langle$ 34, c $\rangle$	$\langle$ 25, d $\rangle$	$\langle$ 16, e $\rangle$
5			$\langle$ 44, c $\rangle$	$\langle$ 35, d $\rangle$	$\langle$ 26, e $\rangle$
6				$\langle$ 45, d $\rangle$	$\langle$ 36, e $\rangle$[b]
7		$\langle$ 42, b $\rangle$			
8			$\langle$ 44, c $\rangle$		
9				$\langle$ 45, d $\rangle$	
10					$\langle$ 46, e $\rangle$

[a] Speculative
[b] Causes rollback

Figure 11: Optimistic distributed time simulation of circuit2

4 The Optimistic Kernel

The optimistic kernel is the key part of the distributed VHDL simulation system. Each node in the distributed network has one optimistic kernel, which is responsible for the VHDL simulation on its node, for the communication with other nodes, and for the optimistic synchronisation of its node.

4.1 VHDL simulation

The simulation kernel is responsible for the correct VHDL simulation on its processor node. The VHDL simulation algorithm consists of two passes:

1. The application of events that are pending in the current simulation cycle on the signals; this includes the events that have been sent in messages. The determination of the processes that have become active owing to a signal change.
2. The processes that are active in the current simulation cycle are executed.

At the start of these two passes the current simulation time is determined based on the smallest timestamp in the event set.

The VHDL simulator in ADVISE! uses multi-threading to solve some of the problems that VHDL simulation poses. Each VHDL process is contained in a thread. The simulation kernel is also a separate thread. This threading solves an important problem: on resumption, a VHDL process always restarts in the correct place, even when deeply nested in a procedure, owing to its program stack that is preserved. Furthermore, the introduction of multi-threading gives an elegant solution to activation and execution of the processes. When the processors are activated they are automatically placed on the list of runnable processes. When the simulation kernel starts pass two, it suspends itself with a low priority; the priority scheduler restarts the kernel thread when it has highest priority, which is after all VHDL processes have stopped. This implements the two pass algorithm in a very elegant manner using standard features of multi-threading libraries.

Figure 12 shows the dataflow in the simulator. The circles in the figure are the different threads, in this example the kernel thread and two processes. The arrows from the signals to the processes show their dependencies: an event on signal 3 means that process 1 is activated.

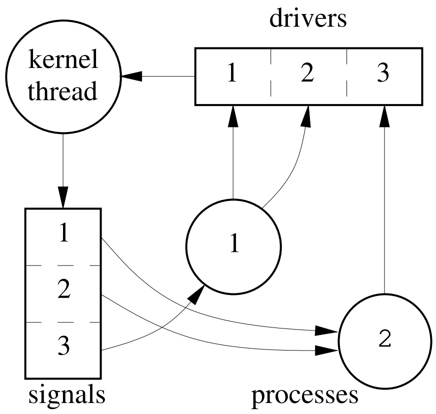

Figure 12: Dataflow in the VHDL simulator.

4.2 Optimistic synchronisation

Although optimistic algorithms promise highest speedup, they come at a cost. Optimistic schemes involves high overhead, both in time and in space; especially its memory consumption is much higher than the alternatives. The following items are very important for an optimistic algorithm:

Message cancellation Speculative execution can lead to the sending of a message that will later turn out be incorrect. This necessitates a message cancellation scheme to undo incorrect messages.

State saving and restoring Optimistic synchronisation can be guaranteed to give correct results by saving its state periodically. When the processor is proven to be incorrect, an old, correct state is restored (a *rollback*), and correct execution can proceed.

Global time It is important to have a good (conservative) approximation of the global simulation time on each processing node. It can discard old states smaller than the global time to free resources (*fossil collection*).

All these mechanisms must be implemented in an optimistic simulator, although the last could be left out in the unrealistic case of infinite resources.

The optimistic simulation kernel in ADVISE! has the optimistic functionality built on top of the VHDL simulation kernel. For the handling of messages there is an extra thread with high priority that is called as soon as a new message arrives. This emulates an interrupt-driven message handling routine in a real system. Figure 13 shows how a graphical representation of the simulation kernel. Besides the two threads in the kernel there is a special module that takes care of memory allocation. This memory manager has two special functions, save and restore, that perform a save or restore of the memory that it has allocated. Using these functions dynamically allocated memory can be saved.

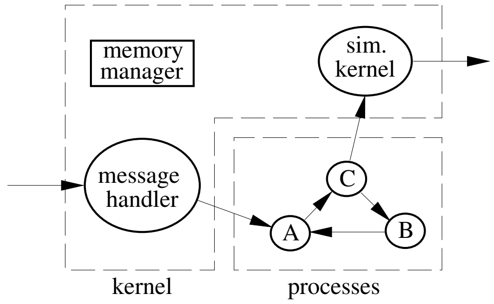

Figure 13: The layout of the optimistic simulator.

4.3 Tuning

The tuning of the optimistic algorithm is very important for a good performance. One of the most influencing factors in the algorithms is the number of simulation cycles that a processor node can execute before it must stall until old states can be discarded. If this number is small, then the system is tightly-coupled, and the local simulation times on the nodes are relatively close. If the number is large, then the system is loosely-coupled and large variations can exist between local simulation times. In general tightly-coupled systems generate many stalls and few rollbacks, and loosely-coupled systems the opposite.

In the ADVISE! simulator, this *lookahead time window* is determined by two separate parameters: (1) N, the number of states saves allows and (2) f, the frequency of state saving. The frequency f is an integer number that denotes that one state save is performed every f simulation cycles. The lookahead time window W is then given by:

$$W = N \times f$$

Future work Future work in tuning will be mostly concerned with general optimisations in the compiler, i.e. make the generated code more efficient. Some excellent possibilities in this area are discussed by Willis and Siewiorek [11]: temporal analysis, waveform propagation, input desensitisation, concurrent evaluation, and statement compaction. Notice that part of these optimisations also speedup sequential simulation.

Another interesting optimisation for optimistic simulation is the detection of stateless VHDL processes [12]. Stateless processes always restart at the same place and have no local variables, so that its state does not need to be saved. The often used *concurrent waveform assignment* belongs to this class, which makes this optimisation certainly worth its while.

5 Results

The results in this section have been obtained with a medium-sized benchmark containing both high and low level constructs. Because the VHDL to C++ compiler is not yet finished at this time, the benchmark has been compiled by hand. Partitioning has been done fully automatic.

Characteristics of the benchmark are listed in Table 1. The benchmark implements a 24x24 bit butterfly operation, part of a set of benchmarks for VHDL performance evaluation [4]. For non-VHDL users, "concurrent waveform assignments" can be viewed as simple processes. VHDL "transactions" are a superset of the events, and a transaction becomes an event if certain conditions have been met.

number of	quantity
Processes	162
Concurrent waveform assignments	146
Signals	2844
Simulation cycles	432
Events	17421
Transactions	56723

Table 1: Some characteristics of the benchmark.

Both the network simulator and the optimistic kernel have a large set of parameters that influence the simulation speed. The influence of the most important parameters on the speedup is shown in this section. These parameters are:

State saving time Because the state in our VHDL is fairly large, optimisation would decrease this size to a large extent. Influence of smaller states can be measured with the framework with a special parameter.

Resources Resources are in general finite. One of the most important resources for optimistic distributed simulation is the memory used for state saves. It is possible to change both the number of state save slots available and the number of simulation cycles between subsequent state saves.

Link bandwidth The link bandwidth is a parameter of the network simulator and shows whether the simulation will gain from a higher bandwidth.

The first measurements are devoted to the generic speedup that can be obtained with a standard set of parameters without optimisations. After that the measurements with altered parameters are shown and their influence on the speedup.

5.1 Speedup

Speedup has been determined for mesh-type networks containing 1, 2, 4, 8, 16, and 32 processor nodes. Speedup is calculated against a network with just one node. The network simulator and the simulation kernel have a special option for sequential simulation. In this way code augmentation is used for sequential simulation as well. The numbers that are obtained with the one node network are therefore a good approximation of the simulation time of our sequential simulator.

Figure 14 shows a graphical presentation of the speedup obtained with the mentioned number of nodes. The line marked “unoptimised state saving” is the speedup that is obtained on a network with normal characteristics, and without any optimisations in the simulation kernel. The line marked “optimised state saving” is shown for comparison and incorporates one important optimisation that is discussed more deeply in the next section, namely that the state of each node can be saved and restored 10 times as cheap as the unoptimised state. The following section shows that this is a reasonable assumption for a real simulator.

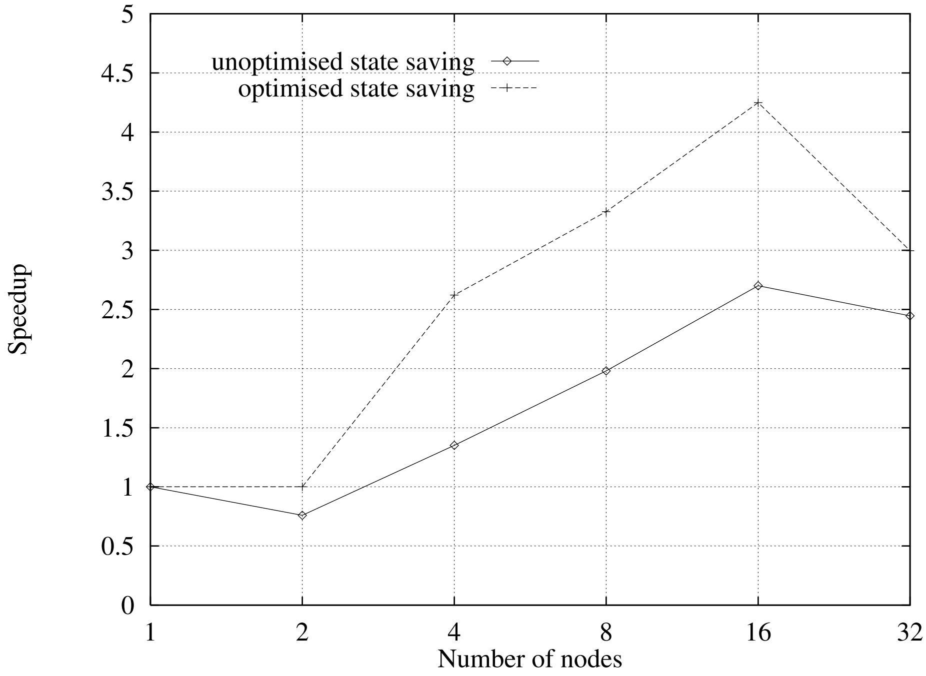

Figure 14: Speedup relative to 1 node.

This figure shows some important characteristics of optimistic distributed simulation that will occur in most simulation instances. The peak speedup is reached around 16 nodes; this means that somewhere between 16 and 32 processor nodes the optimal grain size of the VHDL application has been reached, and further splitting up will only worsen the speedup. Heuristics have to be found that can guide partitioning to the right amount of processors.

5.2 The influence of state saving time

State saving time has a large impact on the performance of the parallel simulator. State saving in the ADVICE! simulator is not optimised, leading to large states that take a long time to save and restore. However, it should be possible to decrease the size of the state in a real simulator and to save it faster. This section shows the influence on the speedup when state saving and restoring is made faster.

Figure 15 shows the speedups that have been obtained by experiments with the state save and restore costs. The figure plots the speedup versus the number of bytes that can be saved per processor cycle. With the number of bytes that can be saved in a single cycle, the cost of state saving and restoring decreases, and speedup is higher. The figure shows that 4 bytes/cycle would give a very decent speedup with the current large states. Because most computers nowadays can save at least 4 bytes/cycle, and some even more, and the state of the simulator can be made much smaller by some simple optimisations, it seems that speedup obtained with 10 bytes/cycle is a reasonable estimate for an optimised simulator.

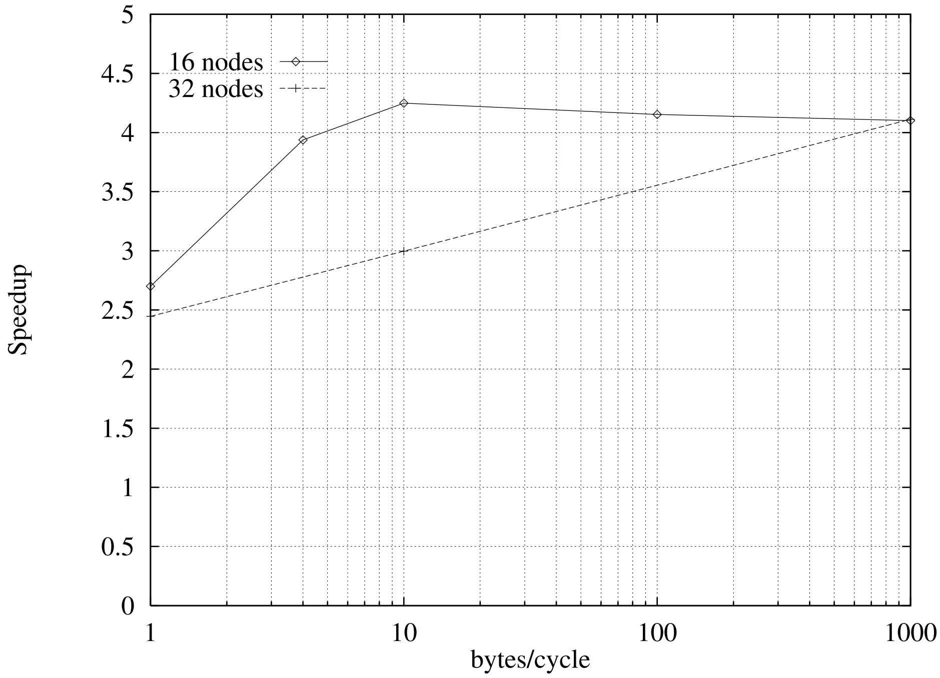

Figure 15: The influence of state save/restore cost on speedup.

Below a certain cost for state saving and restoring, the influence is negligible. This means that the state save/restore cost is a bottleneck for the simulator, but once small enough, it poses no further performance penalty. It can also be seen that the influence on a 32 node network is larger than on a 16 node network; this is again due to the fact that 32 nodes is too many for this VHDL application. There are just too many rollbacks and cancellations with this many nodes for the simulation to be efficient.

5.3 The influence of resources

Resources are very important for optimistic distributed simulation, because the more resources there are, the more each processor can calculate in advance, potentially raising the speedup. This section investigates the size of the lookahead time window as defined in section 4.

Figure 16 shows the influences of resources on the speedup of the simulator for a 16 node network. Each point in the figure has a label in the form "state save slots/frequency", where 25/2 means that there were 25 state save slots, and 1 state save is performed per 2 simulation cycles. All the experiments have been done with optimised state saving enabled, taking away this bottleneck. The influence is not shocking, but is definitely present; especially for small resources the influence is rather large.

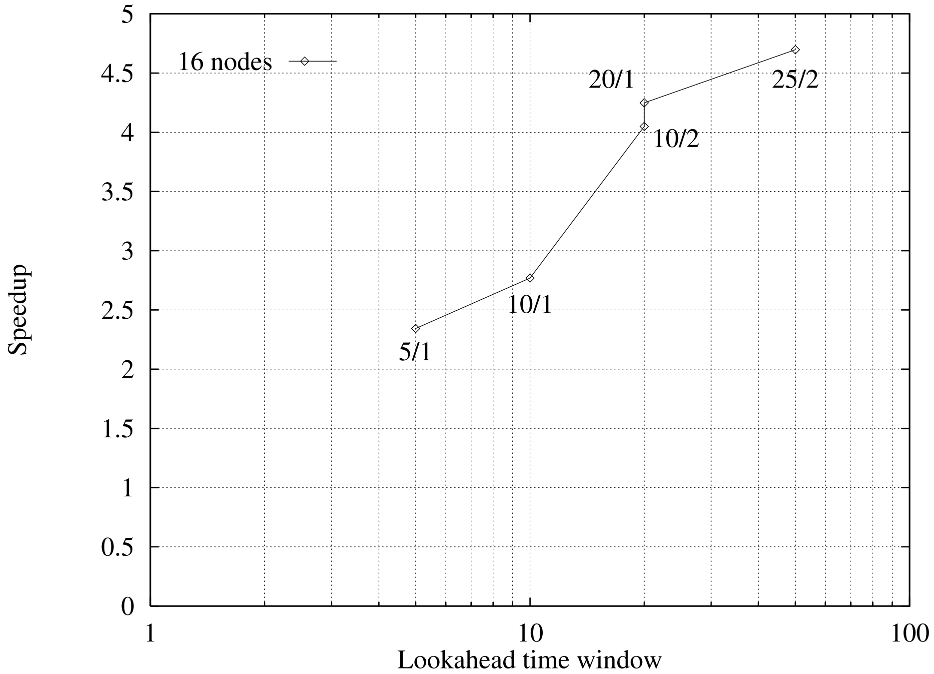

Figure 16: The influence of lookahead time window size on speedup.

5.4 The influence of link bandwidth

Link bandwidth is an interesting issue, because it can show whether the message passing is a bottleneck for the simulator. If it is, then it is not necessary to optimise the simulator, because it can only be faster when less messages are sent.

Figure 17 shows the influence of the bandwidth on the speedup. The numbers on the x-axis show how many communication slots must be used for a message atom of 32 bits. This means that the bandwidth decreases with a higher number on the x-axis. A realistic value for the bandwidth is 10, which has been used in all the other experiments.

The figure shows that bandwidth is not a limiting factor for the simulator, and will not be for a long time. This means that many optimisations in the simulator can be performed without encountering the bandwidth bottleneck.

6 Conclusions

Our simulation environment gives very accurate results, combined with a wealth of simulation options. This allows us to evaluate performance of VHDL programs and evaluate best suited architectures for it.

With the current VHDL simulator, which has not been optimised, we are able to obtain speedups up to 4 with 16 processor nodes. Because the bandwidth of the network is large enough, it is possible to optimise the simulator to a large extent before the bandwidth will become a limiting factor.

It is our conviction that an optimised simulation kernel and larger VHDL programs will lead to a higher speedup, at least up to 8.

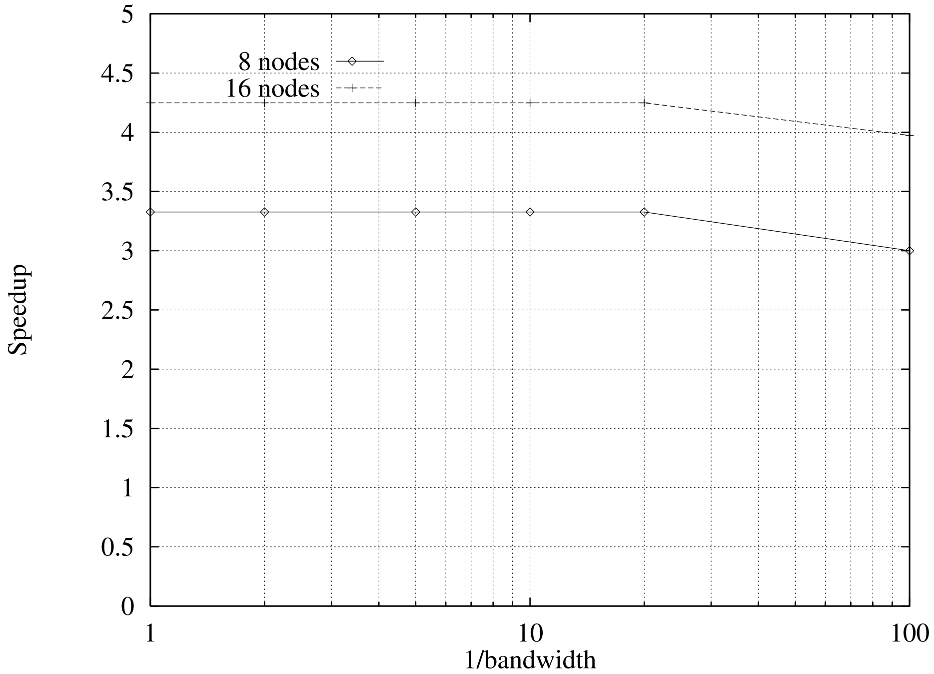

Figure 17: The influence of bandwidth on speedup.

Future work

References

[1] P.J. Ashenden, H. Detmold, and W. McKeen. Execution of VHDL models using parallel discrete event simulation algorithms. *VLSI Design*, to appear.

[2] P. Chawla, H.W. Carter, and P.A. Wilsey. An investigation of the performance of a distributed functional digital simulator. In *Proceedings of the 32nd Midwest Symposium on Circuits and Systems*, pages 470–476, August 1989.

[3] A. De Gloria and P. Faraboschi. Massive parallelism in multi-level simulation of VLSI circuits. *Integration*, 14(2), December 1992.

[4] G. Gorla, A. De Gloria, and P. Faraboschi. A VHDL benchmark suite for the evaluation of simulation performance. Technical report, ITALTEL — University of Genoa, DIBE, June 1993.

[5] M. Hueber, S. Laserre, and A. de Monteville. VHDL experiments on performance. In *Proceedings of EURO-VHDL*, pages 282–292, 1991.

[6] Institute of Electrical and Electronics Engineers. *IEEE Standard VHDL Language Reference Manual, IEEE Std 1076–1987*, March 1988.

[7] B.W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *The Bell System Technical Journal*, 49(2):291–307, February 1970.

[8] Eddy Olk. Parse: Simulation of message passing communication networks. In *Proceedings of the 27th Annual Simulation Symposium*, pages 115–124, 1994.

[9] B.L. Vellandi. *Parallelism Extraction and Program Restructuring for Parallel Simulation of Digital Systems*. PhD thesis, University of Colorado, Department of Computer Science, 1990.

[10] P.J. Weinberger. Cheap dynamic instruction counting. *AT&T Bell Laboratories Technical Journal*, 63(8), October 1984.

[11] J.C. Willis and D.P. Siewiorek. Optimizing VHDL compilation for parallel simulation. *IEEE Design & Test*, pages 42–53, September 1992.

[12] P.A. Wilsey, A.C. Palaniswamy, and S. Aji. Rollback relaxation: A technique for reducing rollback costs in an optimistically synchronized simulation. In Wilsey and Rhodes, editors, *Proceedings of the International Conference on Simulation and Hardware Description Languages (ICSHDL)*, pages 143–148, January 1994.