The CHESS project

Arlet Ottens Wilco van Hoogstraeten

November 11, 1993

Abstract

The CHESS project is funded by the ESPRIT committee to research the feasibility of parallel VHDL simulation. Research is directed into defining a simulation algorithm with accompanying hardware that will have the highest performance.

Two trajectories have been defined to obtain these goals: the *trace analysis* and the *compiler* trajectory. Two of the partners have each followed one of these. Trace analysis is not very accurate, but gives quick results, while the compiler based simulation system is very accurate, but is a large task that has not been completed within the official ESPRIT project duration. Further research will be done without funding.

Trace analysis has shown that centralised-time algorithms can only reach a parallelism of 2, while distributed algorithms can easily reach 10, on the same set of medium-sized benchmarks.

1 Introduction

The CHESS project is part of the ESPRIT funding program of the European Community. CHESS is the acronym for: "A MIPS cruncher for a distributed Concurrent HEterogeneous Simulation System." The project was initiated to define the best distributed simulation algorithms and accompanying hardware for the parallel simulation of VHDL.[1] VHDL is a standard hardware description language that is supported by a large user base throughout the world. The project is a cooperation between five partners in four different countries:

DIBE DIBE is the department of Biophysical and Electronical Engineering of the University of Genova in Italy. DIBE has grown experience in the area of architectural support for VLSI specific problems, like simulation and testing.

DIBE has investigated trace analysis as a tool to determine performance of parallel simulation algorithms adapted to VHDL.

Intergraph Intergraph is an American, so-called, CAD-vendor with a branch in England. Among their wide range of products is a complete design, test and simulation environment for VHDL.

Intergraph has collected benchmarks, and was responsible for investigating the marketing possibilities of the defined simulation system.

[1] VHSIC (Very High Speed Integrated Circuit) Hardware Description Language

Italtel Italtel is a large Italian company that has its main interest in telecommunication. Italtel has long-term experience in designing VLSI, and using VHDL simulation.

Italtel was involved in developing standard methods for writing VHDL and producing realistic benchmarks. Italtel cooperated with Intergraph to set up the business requirements specifications.

SICAN SICAN is a publicly funded institution in Germany that provides VLSI design capabilities to small firms. Lately they have started their own research into the development of better VLSI design tools.

SICAN has developed a tool for partitioning a large VHDL program into smaller subprograms that can run in parallel.

TUD TUD is the acronym of The Computer Architecture group of the Delft University of Technology in the Netherlands. TUD has large experience in the area of special purpose compilers and the architectural design of application specific processors.

TUD has investigated the performance of parallel VHDL simulation by developing a compiler and runtime environment that can accurately measure speedup and other important statistics.

This paper contains a brief introduction of VHDL and simulation algorithms in sections 2 and 3, followed by a discussion of techniques to measure performance in section 4, ended by a more detailed description of the "compiler trajectory" and some results in sections 5 and 6.

2 The VHDL Language

VHDL is the official standard for the description of logic circuits that are designed for the DoD, the Department of Defense of the United States. It performs the same role as Ada does for software, and resembles Ada in many aspects. VHDL is standardised by the IEEE in the IEEE standard 1076, which has been widely accepted by industry. VHDL can be used to describe any logic circuit on any possible level, although it is best suited to descriptions on higher levels of abstraction.

Although it is impossible to provide a complete overview of VHDL in a short space, we will outline the main characteristics needed to understand the aspects typical to VHDL simulation. Figure 1 shows a small example of a circuit. Every circuit in VHDL consists of a number of processes interconnected by signals. A signal can be seen as a wire running from one process to another. The most common signal has one input and one output, but there are also more specialised kinds of signals like buffers, and bidirectional signals. For our purposes it suffices to look at the most basic signals only.

Every process has a description in the form of sequential code. This code describes how the process reacts to changes on incoming signals and how it drives outgoing signals. The code that is used in these sequential parts is very similar to Ada, augmented with a number of typical VHDL features.

The two most important features of VHDL which we do not find in Ada are the *wait statement* and the *waveform assignment*. Both constructs can be used with signals. Consider the following example of a simple VHDL process:

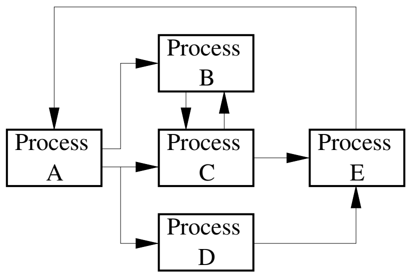

Figure 1: A circuit consisting of a number of processes.

```
entity Nor_Gate is
   port( a, b: in Bit; y: out Bit );
end Nor_Gate;

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

In this example we can see the definition of an entity called `Nor_Gate`. This entity has an interface consisting of two incoming signals and one outgoing signal, as specified by the `port`-clause. Below the definition of the entity, a implementation is defined in an architecture named 'trivial'. In this implementation we can see one process consisting of two constructs:

- A waveform assignment denoted by '`<=`' symbol. On the left of this symbol is the name of the signal to be changed, on the right is the new value, followed by the delay after which the signal change becomes active.
- A wait statement. This statement causes the process to wait for a change on either signal `a` or signal `b`. As soon as one of these signals changes, the process continues executing. The code between `process begin` and `end process` is executed repeatedly.

The waveform assignment does not directly influence the value of the signal, it only schedules a future change. Even if the delay is zero, the signal is only changed in the next simulation cycle.

2.1 Signals and Drivers

Associated with every signal is a driver, as shown in figure 2. The driver contains a number of *transactions*. Every transaction is a pair consisting of a value and a time. For any driver there is exactly one transaction whose time component is not greater

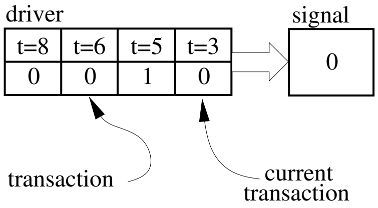

Figure 2: A driver associated with a signal.

than the current simulation time. This transaction is called the *current transaction*. All transactions together make up the *projected output waveform*. When time advances, and becomes equal to the time of the next transaction, the first transaction is deleted from the projected output waveform, and the next becomes the current value of the driver. The current value of the driver is copied into the value of the signal.

2.2 Execution of VHDL

The VHDL execution consists of an initialisation phase followed by the repetitive execution of the statements in the processes that are defined in the VHDL description. The initialisation phase consists of the following steps:

1. All initial values of signals are calculated, and the current value of a signal is set to this value. It is assumed to have been the value of the signal for an infinite length of time prior to the start of simulation.

2. Each process in the model is executed until it suspends, i.e., performs a wait statement.

After the initialisation, the model is executed according to the following algorithm:

1. If no driver is active, then simulation time advances to the next time at which a driver becomes active.

2. Each active signal is updated. A change in value of the signal results in an *event* on that signal.

3. For each process P, if P is currently waiting on a signal S, and an event has occurred on S in this simulation cycle, then P resumes.

4. Each process that has just resumed is executed until it suspends.

As we can see from this description, in the running simulation there are two separate phases: updating the signals and executing the processes.

3 Simulation Algorithms

Figure 3 shows a classification of simulation algorithms. There are two main categories: state-driven and event-driven. In state-driven simulation the entire model is evaluated at every time step. This contrasts with event-driven simulation, which only evaluates the components that are active at the current time step. Considering the definition of the VHDL simulation cycle it appears that event-driven simulation is the only option, because events are a part of VHDL, and in that respect very hard to bypass. We did not find any way to modify state-driven simulation in such a way that it could be successfully applied to the simulation of VHDL. Sequential event-driven simulation is not interesting

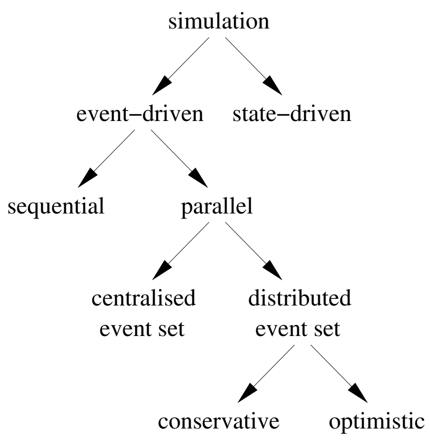

Figure 3: A classification of simulation algorithms.

considering the goals of the CHESS project which dictate the use of a parallel simulation algorithm. This leaves only three options for a simulation algorithm, namely:

PCES In the Parallel Centralised Event Set algorithm all events (change of a signal) are maintained in a central event queue. The main loop of the simulation algorithm removes the first event from the queue (and every other event with the same time stamp) and marks every process waiting on the associated signal. If this has been done, all marked processes are executed in parallel. New events generated by these processes, are placed into the event queue. Of course, the amount of parallelism is limited, and the performance is restricted by the bottleneck that is created by having a central structure.

Conservative PDES In the Parallel Distributed Event Set simulation, there is no longer a central event queue. Every processor has its own event queue, which contains only the events for the signals of the processes that are running on this particular processor. Events on signals that cross the boundary between two processors, are propagated by means of messages. In conservative PDES, every processor only generates events that are correct. An example of a conservative PDES algorithm is the Chandy-Misra-Bryant algorithm [3, 12].

Optimistic PDES Optimistic PDES is comparable to conservative PDES in its use of separate event queues per processor. The main difference is that not only correct events are generated, but that speculative processing of incoming events is allowed. This might lead to uncorrect processing, and to the generation of uncorrect events. The algorithm detects uncorrect events, and has a mechanism to cancel them. Most optimistic algorithms are based on the Virtual Time paradigm, implemented in the Time Warp algorithm described by Jefferson [9].

The choice between those three alternatives is not easy to make. The PCES algorithm has the main disadvantage of a limited amount of parallelism, caused by the central event queue. The conservative PDES algorithm is troubled by the problem of *deadlock*. When a model contains a cycle of processors, waiting for events of each other, a situation arises that no processor can continue with the simulation: deadlock occurs. Avoiding deadlocks is possible but only at the cost of introducing more messages into the network. Recovering from deadlock is a global and sequential operation, imposing a limit on the amount of parallelism.

Optimistic PDES does not have the problems of deadlock, and results in the largest amount of parallelism. However, the overhead caused by introducing uncorrect messages and cancelling them can be larger then any gain in speed caused by the higher parallelism.

The various algorithms, together with a number of refinements, have been tried and tested by researchers [1, 2, 4, 7, 10, 11, 13, 14]. It seems that the optimal algorithm heavily depends on the type of problem that is being simulated. It is not clear which algorithm should be preferred to simulate VHDL. This, of course, does not only depend on the language itself, but also depends on the properties of the VHDL circuit that is being simulated.

4 The Simulation Workbench

The main objective of the CHESS project has become to find out which of the main simulation algorithms leads to the largest speedup. Three alternatives have been proposed to evaluate the different algorithms: trace analysis, the VHDL interpreter, and the VHDL compiler trajectory. These alternatives are outlined below.

4.1 Trace Analysis

The aim of the Trace Analysis trajectory is to estimate the possible speedup by analysis of execution traces produced by a commercial VHDL simulator. Figure 4 shows the trace analysis trajectory. The function of the whole system is to estimate the length of the critical path in a distributed simulation for a given benchmark. As a consequence the average parallelism and the estimated speedup for a given number of processors and interconnection network can be deduced. The critical path is the time necessary to terminate the execution of all instructions, without violating the causality constraints imposed by the simulation semantics. The system works with the following data:

- The VHDL sources of a benchmark.
- The VHDL statement traces; these are the statements that are actually executed by the simulator.
- An architecture description in terms of number of processors, interconnection topology, cost of sending a message.

The signal dependency extractor reads the VHDL sources comprising the benchmark and extracts dependency information for each VHDL statement. The partitioner applies

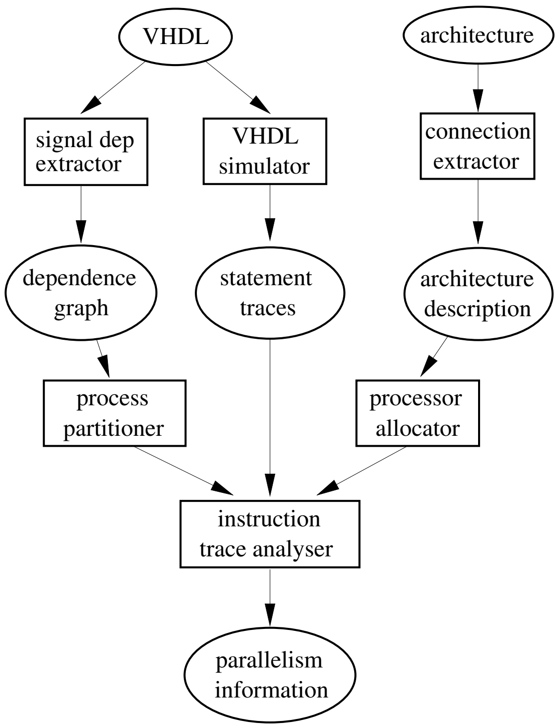

Figure 4: Outline of the trace analysis trajectory.

a Min-Cut algorithm to the circuit description in order to divide the VHDL processes among the processors in the network. The instruction trace analyser is the heart of the simulation system and consists of a trace driven simulator. Given an architecture model, a simulation algorithm and a VHDL benchmark the instruction trace analyser provides information about:

- Execution time of a sequential simulation.
- Execution time of a centralised-time parallel simulation.
- Execution time of a distributed-time parallel simulation.
- Length of the critical path of the benchmark.
- Average parallelism.
- Number of exchanged messages.
- Most used communication channels.

4.2 The VHDL interpreter

The disadvantage of the trace analysis trajectory is that it cannot produce the correct output for the optimistic algorithms. This is caused by the fact that the execution traces only contain the *correct* events, they do not contain the *uncorrect* events that were generated due to speculative execution of parts of the model. They also do not contain the cancellation messages that were sent as a result of correcting the errors. Using trace analysis we can show that in some cases, a number of rollbacks would occur at a certain time, but we cannot accurately predict the amount of rollbacks. Using a

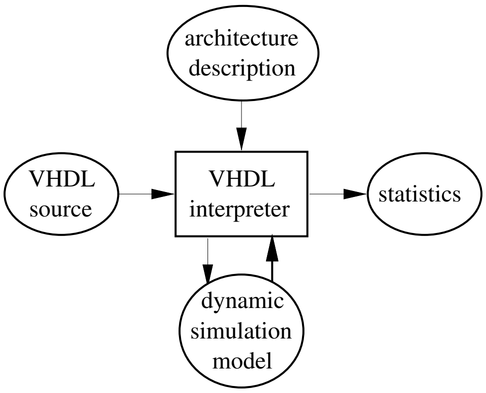

Figure 5: Overview of the VHDL interpreter.

VHDL interpreter, we can simulate VHDL in a realistic way. The core of the simulator is a program (see figure 5) that reads the VHDL source, and interprets it statement by statement, simulating all processes, maintaining an event queue and a list of signals. The interpreter also takes care of simulates the underlying architecture. The interpreter is capable of measuring all aspects of the simulation algorithm and the architecture that are considered important.

One of the functions of the interpreter is to assign a certain cost to every VHDL statement in terms of time necessary to execute it. This cost should be added to the total execution time of the processor to which the execution of this particular VHDL statement is assigned. The problem for the writer of the interpreter is to accurately estimate the cost of every VHDL statement. There are a large number of different VHDL statements, ranging from very simple ones such to very complicated and costly ones. It seems an almost impossible task to estimate these timings.

4.3 The VHDL compiler trajectory

The compiler trajectory, shown in figure 6, is the most comprehensive of the three alternatives. It avoids the problem of estimating the execution times of every VHDL statement. Instead the simulation is done as follows: the VHDL source is translated by a special compiler into C++ code, and this code is subsequently linked together with two libraries:

- The ParKer, or the parallel simulation kernel. This is a runtime environment that includes several constructs from VHDL, like signals, drivers and processes. These constructs are all implemented as C++ classes. The semantics defined by the IEEE standard [8] are all implemented by these classes. The user program only has to define a signal, and an associated driver, and then it can perform waveform assignments on the signal. The simulation kernel takes care of suspending and resuming the processes.

- The ArchSim, or architecture simulator. This is a library that is used by the ParKer. The ParKer itself uses *partitions*. A partition is a set of processes that belong together, and are supposed to be executed on the same logical processor. The ArchSim simulates the underlying parallel architecture, and takes care of process accounting, message passing and providing virtual interrupts.

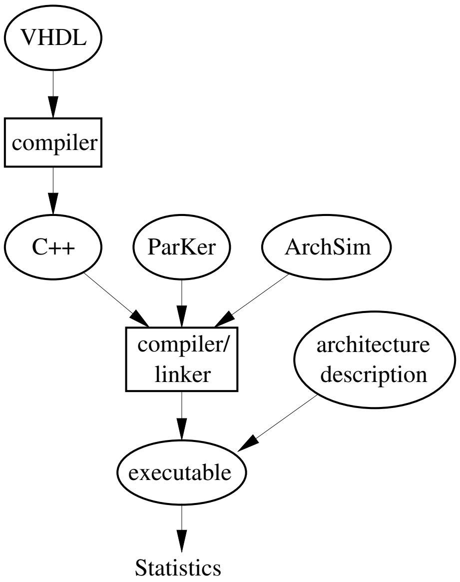

Figure 6: Overview of the VHDL compiler trajectory.

When the generated C++ is combined with the ParKer and the ArchSim, this results in an executable that can be run by the user. The executable reads the description of the architecture, and produces statistics. These statistics include:

- Average parallelism.
- Number of messages per communication link.
- Overhead caused by simulation algorithm, such as cost of rollbacks.
- The resulting speedup.

The time consumed by processing the VHDL statements, and by the overhead of the simulation algorithm are all measured in terms of CPU-cycles of the (sequential) processor that runs the simulation. The aim is to count the number of machine instructions executed per logical processor, not the real or wall-clock time that is needed to execute them.

Apart from providing us with the most accurate simulation results the compiler trajectory also leaves a compiler as a side effect. Such a compiler can serve as a starting point for a commercially viable VHDL simulator.

4.4 Choice between the alternatives

After some consideration it is clear that building a VHDL interpreter is too much work, and does not lead to particularly good results. The extra work involved in designing and implementing a VHDL compiler is worthwhile considering the advantage of accurate results, and the usability of the final product. The trace analysis trajectory is also interesting. Although it does not provide us with the most accurate results, and does not result in a product that has any use apart from gathering statistic, it has the benefit of being relatively easy to implement.

Because there are two partners involved in this particular part of the ESPRIT project, we decided to pursue both alternatives. The trace analysis trajectory was followed by the university of Genova, while the university of Delft started to work on the compiler trajectory.

5 The Compiler Trajectory

This section shows the compiler trajectory, as pursued by our group in Delft, in more detail. Subsections deal with the runtime organisation, the VHDL compiler, the parallel simulation kernel and the architecture simulator.

5.1 The Runtime Organisation

The runtime organisation defines the interface between the generated C++ code and the ParKer, and consists in our case of a number of C++ class definitions. These classes serve to raise the semantic level of C++ to that of VHDL. In fact it is possible, although cumbersome, to write a VHDL-like description in C++, using the runtime environment.

The three main ones are called `Value`, `Signal` and `Process`. These will be shown in more detail below.

The Value class

The Value base-class is able to represent any kind of VHDL value. There are no direct instances of `Value`, but it is possible to derive from it. For example, the `Int` class is used to contain integer values, and the `Real` class is used to contain real numbers. Both inherit from the `Value` class. The main advantage of this approach is that other routines in the runtime environment, can deal with general objects of type `Value`.

For example, there is a function that performs a waveform assignment, it takes pairs consisting of a value and a delay and puts them into the event queue. At the appropriate time, the event queue handler takes the value and assigns it to a signal. The type of the value and the type of the signal, are not important in these steps. If we had chosen to implement VHDL integers using the C++ `int`-type, and reals using the C++ `float`-type, then it would be necessary to provide multiple versions of a large number of routines in the runtime environment. Using a single `Value` class results in a small and clean interface with the simulation kernel, at the price of slower execution.

The following derived classes are defined within the runtime organisation:

- The `Int` class is used to hold a number of VHDL types, like integers, enumerations, characters and subranges.

- The `Real` class is used to hold the VHDL real type.
- The `Null` class does not contain any value, it is used for the value of an unconnected signal in VHDL.
- The `Composite` class is used to hold the two composite types: arrays and records. Because every element of a record can be represented by an object of the `Value` class, a record is just a list of `Value`s, and can therefore be implemented just like an array. Arrays in VHDL can be ascending or descending, depending on the type of the index. In the runtime organisation, descending arrays are implemented as ascending arrays but with negated indices. So, the array with index 10 downto 3 is implemented as an array with range -10 to -3. The compiler takes care of inserting the necessary minus signs.

Objects of the `Value` class are pointers to the actual values; a garbage collection system is implemented using reference counting. It would take too much space to show the `Value` class and its derived classes in detail, but table 1 shows some examples of its use.

VHDL	C++
`variable I: integer := 3;`	`Int I( 3 );`
`I := I + 4;`	`I = I + Int( 4 );`
`A[I] := A[I] + B;`	`A[int(I)] = A[int(I)] + B;`

`A = { 3, 5, 7 };`	`Composite tmp( 1, 3 );`
	`tmp[1] = 3;`
	`tmp[2] = 5;`
	`tmp[3] = 7;`
	`A = tmp;`

`if A < B then ...`	`if( A < B ) then ...`

Table 1: Examples of VHDL and C++

The Signal class

The `Signal` class is used to represent all VHDL signals. It is implemented just like the `Value` class, as a pointer to the actual data. In contrast to the `Value` class there is no garbage collection system provided for signals, because usually they are only destroyed when the entire simulation has finished. There are two important types of signals: scalar signals and composite signals, the latter being semantically equivalent to an array of subsignals. There is also a class of implicit signals, that are not explicitly specified by the user, but are defined by the VHDL compiler to implement resolved signals, and a number of derived signals that result from attributes like `'event` and `'active`. The semantics of the implicit signals differ slightly from the normal ones.

The Process class

The `Process` class is used to represent VHDL processes. Every process is implemented as a *thread* in C++. Threads are implemented using a special library entirely written in

C++. Every thread has its own stack, but shares its data space with the other threads. A context-switch routine is implemented using `setjmp()` and `longjmp()` calls. A thread can suspend its execution by calling the `wait()` function. In the runtime organisation, this feature is used to wait for changes on signals. Built into the threads package is a priority mechanism: as soon as a running task suspends itself, the scheduler picks the next runnable thread with the highest priority. The priority mechanism is used in the runtime organisation by lowering the priority of the event queue handler below those of the VHDL processes. Therefore, the event handler is only active when all other processes are waiting for a signal change. The NOR-gate that was shown on page 3 can be translated into C++ in the following way:

```
class Nor_gate: public Process
{
public:
   Nor_gate( Signal a_in, Signal b_in, Signal y_out );
   int main( );

private:
   Signal a, b, y;
}

Nor_gate::main( )
{
   Driver y_driver( y, Int(0) );

   for( ;; )
   {
      y_driver.waveform_assign( a.nor(b), Time(10) );
      wait_for( a, b );
   }
}
```

To implement a VHDL process, it is necessary to define a new class that inherits from `Process`. This class needs a member `main()` containing the body of the process. Because this member has no parameters, the constructor function (not shown here) takes care of copying the parameters into the private area, and initialising the thread. By declaring variables like this:

```
Nor_gate gate_1( s, q, not_q );
Nor_gate gate_2( r, not_q, q );
```

we create two interconnected NOR-gates to form a simple SR-latch.

5.2 The VHDL Compiler

The VHDL compiler is built using the front end from CLSI[2]. This front-end produces an annotated abstract syntax tree on which the static semantic analysis has already been performed. This analysis includes type checking, scope resolution, and overload resolution. Using a library the abstract syntax tree can be read by the back-end, and subsequently

[2]Cad Language Systems Inc.

manipulated. The back-end itself is written in C, and implements a reasonably large subset of VHDL.

Thanks to the high semantic level of the runtime organisation, the job of the compiler is relatively easy. It builds a directed acyclic graph (DAG) representing the hierarchy of entities and processes, as outlined in figure 7. The bottom level nodes in the DAG are the processes, the other nodes are entities, or environments. The next step is to transform this DAG into a flattened list of separate processes. The compiler then produces a class

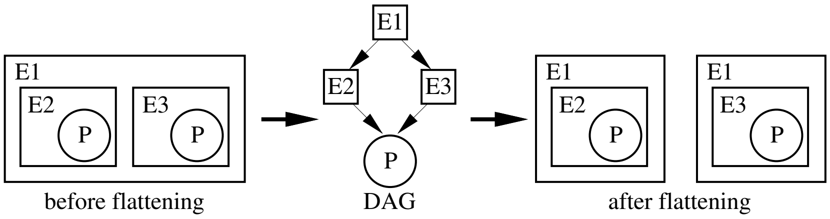

Figure 7: Flattening the processes.

definition for every unique process. In the case of figure 7, only a definition for process 'P' is produced. For every path from the top of the DAG to the leaves, the compiler generates a *partition*, an instantiation of the process with the appropriate combination of environments. Each of these partitions is supposed to run on its own processor. In a future version of the compiler the simple-minded partitioner will be replaced by a version that is capable of assigning multiple VHDL processes to a single partition. This capability is vital to avoid excessive communication overhead between the partitions. The advantage of the current partitioner, apart from its simplicity, is that it allows us to find an upper limit to the parallelism by artificially lowering the communication cost to zero.

5.3 The Parallel Kernel

The parallel kernel (ParKer) was designed with flexibility in mind. It has two well-defined interfaces that allow it to be used for the simulation of a wide variety of simulation algorithms, and network architectures and topologies. It is possible to implement a

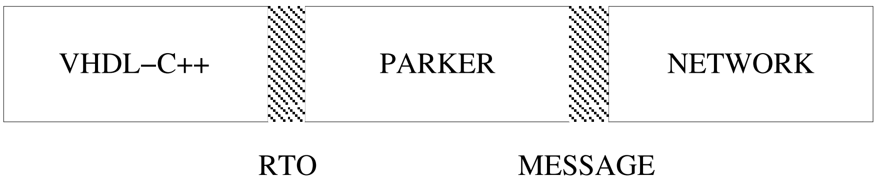

Figure 8: The two interfaces of the parallel kernel. On the left the interface to the generated C++, the runtime organisation interface, on the right the interface to the network, the message interface.

large class of parallel simulation algorithms in this way, without altering the C++ code generator. A flexible message passing interface on the network side is easily adaptable to a wide range of message passing interfaces. In the remainder of this section we will restrict the discussion to the kernel for optimistic simulation, because it has the most and the most difficult features.

Together with the C++-code for the VHDL processes, the parallel kernel forms a *partition*. A partition runs on a single node, which could be either a uniprocessor, or a shared-memory multiprocessor system. Internally the partition uses a sequential or centralised-time algorithm, while externally it uses a distributed one.

The internals

The parallel kernel uses two threads: one handling the incoming messages, and one controlling the VHDL simulation (the *event-handler*). Furthermore it comprises a wide range of functions that are used by the generated C++ and the event handler. The parallel simulation algorithm is located partially in the incoming-message-handler and partially in the event-handler.

An important part of optimistic simulation is the state-save/rollback functionality. State-saving consists of several steps:

- The saving of the stacks of all threads of VHDL processes and the stack of the event-handler.
- The saving of all the memory that has been dynamically allocated by the VHDL process code. A special memory allocator, which keeps track of all allocated memory, takes care of this.
- The saving of any I/O states that have been initiated in the VHDL processes.
- The saving of "global" variables (actually local to this kernel) that are used in the VHDL simulation.

The state can be saved to real memory or to disk, depending on preferences and circumstances. Not saved are the lists of messages that are received and sent. These have to be kept across any rollbacks, because they contain information that cannot be reproduced [9].

The incoming-message-handler has a high priority, and will be called almost immediately on the arrival of a new message. The event-handler has the lowest priority and will only be called when all VHDL processes have suspended. It will then determine the time of the next simulation cycle, process the pending events, and send the associated messages.

The externals

Values of VHDL signals that are shared in two or more partitions are communicated by means of messages. Several types of messages exist, depending on the simulation algorithm. Normal messages communicate the occurrence of an event on a signal; conservative simulation uses *null-messages* to avoid or resolve deadlocks [3, 12]; optimistic simulation uses *anti-messages* to discard incorrect messages [6]. All distributed algorithms need extra messages to update the global simulation time, needed for time-ordered I/O and other tasks that are specific to different simulation algorithms.

Special tables have to be generated by the partitioner for each partition. One table containing all signals that are used in the partition but driven on other ones, and another table containing all signals that are driven in the partition and used somewhere else.

Incoming messages Upon arrival a message is checked for having a timestamp smaller than the local simulation time. This will cause a rollback to the first appropriate time. In both cases the message is put in a list of all incoming messages, or, in case of an anti-message, a message is deleted from it.

Outgoing messages Upon application of a VHDL event on the signal, a message is sent to all other partitions that use that signal. The message is also put in a list, for further use in cancelling uncorrect messages.

The global simulation time, which is the minimum of all local simulation times and timestamps of all unprocessed messages, is used to restrict the size of the message lists, and the amount of states that are saved.

5.4 The Architecture Simulator

The architecture simulator is capable of simulating a distributed system consisting of processing nodes using an interconnection network. The nodes communicate with each other by sending messages over the network. Every node in the architecture simulator corresponds exactly to a partition in the ParKer. The architecture simulator has three tasks:

- Executing the partition at every processor node. This is complicated by the fact that the nodes send asynchronous messages to each other, resulting in some sort of interrupt at the receiving note. In order to simulate this interrupt, the simulator must use some sort of polling mechanism. After executing a few instructions of a node, it checks to see if any message has arrived. If this is the case, the running node is notified of the incoming message.

  The simulator also has to check at regular intervals whether the simulated time for one node is not advancing too much with respect to the other nodes in the system. If a certain margin has been exceeded, the running node is suspended in order to allow the other nodes to catch up. This behaviour is vital in order to correctly simulate the asynchronous message passing used in the ParKer.

  The execution time of the processor nodes is modelled using a technique called *code augmentation*, as described by Weinberger [15].

- Handling the messages that are sent from one node to another. To accurately calculate the communication performance, the simulator uses a realistic model of the architecture. Message passing is based on the OSI model for computer networks, and covers the transport, network and data link layers. The simulator allows the user to vary a number of parameters for each of those layers. The current version of the simulator uses E-cube routing in a 2-D mesh topology, but more topologies and routing strategies are expected to be developed in the near future.

- Performance analysis of the processor nodes and communication channels. By means of a configuration file, it is possible to selectively monitor a number of events in the simulator. The results are written to a trace file, which can subsequently be analysed by further tools, allowing the user to extract many different aspects of the architecture performance.

6 Conclusions

Trace analysis [5] has shown that centralised-time algorithms will not play an important part in high-performance parallel VHDL simulation. Because the maximum obtainable parallelism stabilises around 2. It has also shown that the obtainable parallelism for distributed simulation algorithms can easily reach 10, and this just for medium-sized realistic benchmarks.

The trace analysis will be refined to use more accurate data on execution times of VHDL statements. This will give still better results.

The compiler trajectory was a very large task, and has unfortunately not completed. Results are expected in the beginning of 1994. These results will give solid data on which to base the final conclusions of the project.

The compiler will be extended to generate C++ for all VHDL statements. This will allow us to use any benchmarks that are available. The parallel kernel will be refined and optimised for better performance; also a range of simulation algorithms will be implemented for better comparison. The architectural simulator will be made more flexible, with different network topologies and routing algorithms.

References

[1] R. Ayani. A parallel simulation scheme based on the distance between objects. In *Proceedings of the SCS Multiconference on Distributed Simulation 21*, pages 113–118, March 1989.

[2] W. Cai and S.J. Turner. An algorithm for distributed discrete-event simulation — the "carrier null message" approach. In *Proceedings of the SCS Multiconference on Distributed Simulation 22*, pages 3–8, January 1990.

[3] 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.

[4] K.M. Chandy and R. Sherman. The conditional event approach to distributed simulation. In *Proceedings of the SCS Multiconference on Distributed Simulation 21*, pages 93–99, March 1989.

[5] A. De Gloria and P. Faraboschi. A trace-driven evaluation system for VHDL parallel simulation. Technical report, University of Genoa, Dept. of Biophys. and Elect. Eng. (DIBE), September 1993.

[6] R.M. Fujimoto. Parallel discrete event simulation. *Communications of the ACM*, 33(10):30–53, October 1990.

[7] B. Groselj and C. Tropper. A deadlock resolution scheme for distributed simulation. In *Proceedings of the SCS Multiconference on Distributed Simulation 21*, pages 108–112, March 1989.

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

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

[10] Y.-B. Lin, E.D. Lazowska, and J-L. Baer. Conservative parallel simulation for systems with no lookahead prediction. In *Proceedings of the SCS Multiconference on Distributed Simulation 22*, pages 144–149, January 1990.

[11] B.D. Lubachevsky. Efficient distributed event-driven simulations of multiple-loop networks. *Communications of the ACM*, 32(1):111–131, January 1989.

[12] J. Misra. Distributed discrete-event simulation. *Computing Surveys*, 18(1):39–65, March 1986.

[13] J.K. Peacock, J.W. Wong, and E.G. Manning. Distributed simulation using a network of processors. *Computer Networks*, 3(1):44–56, February 1979.

[14] R.C. De Vries. Reducing null messages in misra's distributed discrete event simulation method. *IEEE Transactions on Software Engineering*, 16(1):82–91, January 1990.

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