Analysis of Parallel VHDL Simulation

Wilco van Hoogstraeten Arlet Ottens
Henk Corporaal
Delft University of Technology
Department of Electrical Engineering
Email: chess@duteca.et.tudelft.nl

Abstract

We propose a performance evaluation scheme for parallel VHDL simulation. This scheme provides accurate information about the performance of various parallel event driven simulation algorithms.

A compiler is used to translate the VHDL source into C++ which is linked with the parallel simulation algorithm and executed using a parallel architecture simulator. Because of the flexibility of the simulation tools, we can exchange simulation algorithms very quickly and can use a wide variety of network types and topologies.

Thanks to a careful design of the tools, they can be used for other simulation problems with only minimal changes.

Keywords: parallel simulation performance, VHDL simulation, network simulation.

1 Introduction

In the framework of the CHESS project[1], we have developed a hardware independent method to evaluate the performance of parallel VHDL simulation. The method enables us to measure the performance of different parallel simulation algorithms on parallel architectures with different characteristics. Instead of running the simulation on existing hardware platforms, our method uses a architecture simulator on a normal workstation. Currently our research is focussed on the simulation of VHDL, but the design can be modified for other kinds of simulation.

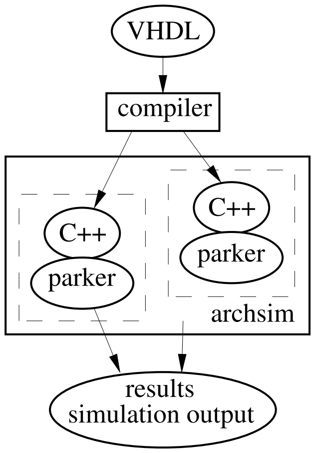

Figure 1: How we collect statistics

Figure 1 shows the method we use to collect statistics of the parallel VHDL simulation. Top-down, the following tools are involved:

compiler The compiler partitions the VHDL source and compiles it into C++. This results in a number of parallel runnable C++ programs that have compiled-in information about other partitions needed for message passing.

parker The parallel kernel (parker) contains the simulation algorithms both for the internal VHDL simulation and the global parallel simulation. One C++ program and a parker form a *partition* (the dashed boxes in Figure 1) that denotes the code for one network

[1]The CHESS project was partly funded by the ESPRIT programme of the European Community.

node.

archsim The architecture simulator (archsim) simulates the underlying network and keeps accounting information of the execution times of the processors. The nodes of the simulated network are simulated by the partitions.

In our framework, all the tools together with the generated C++ are compiled into one executable. The executable can give a wide range of data, both simulation-dependent and network-dependent.

Other partners in the CHESS project have composed a set of benchmarks to measure the performance of parallel VHDL simulation [3]. We will use these benchmarks to evaluate the simulation algorithms and network topologies.

Section 2 provides a short introduction to VHDL, section 3 describes the VHDL to C++compiler, section 4 describes the workings of the parallel simulation kernel, section 5 describes the architecture simulation, and section 6 concludes the discussion.

2 VHDL

VHDL [4] is the official standard for the description of logic circuits that are designed for the DoD, the Department of Defense of the United States. 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.

Because it is impossible to provide a complete overview of VHDL in a short space, we only outline the main characteristics needed to understand the aspects typical to VHDL simulation. Figure 2 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.

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 most important being the *wait statement* and the *waveform assignment*. They are used in the following example:

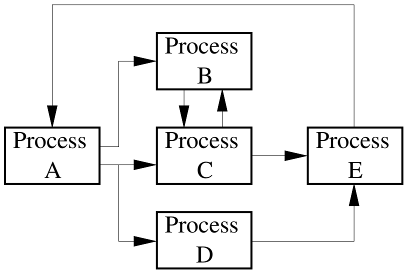

Figure 2: A circuit consisting of a number of processes connected by signals.

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

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

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, an implementation is defined, consisting of one process that contains two statements:

- A waveform assignment denoted by a '<=' 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`.

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.

After the initialisation of all values, and the execution of every process until it suspends, the simulation cycle starts. It consists of two separate phases:

- In the first phase, the simulation time is advanced to that of the earliest event, and signals are updated.
- In the second phase, the processes that have been waiting for signals are executed if the signal has been changed in the first phase. This phase ends if all processes are suspended again.

The VHDL simulation cycle closely resembles that of event driven simulation.

3 The VHDL to C++ compiler

The VHDL to C++-compiler is responsible for translating a VHDL source into a number of C++-sources, that are combined with parallel kernels to form partitions. In order to make the compiler as simple as possible, it uses a high level runtime organisation.

3.1 The Runtime Organisation

The runtime organisation defines the interface between the generated C++ code and the parallel kernel, and consists 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 advantages of the high level runtime organisation are that we are able to develop the VHDL compiler with a minimum of effort.

The three most important classes are called Value, Signal and Process. These are shown in more detail below.

Value The Value class offers a single datatype to represent every possible VHDL type. This greatly simplifies the interface of the runtime environment. Reference counting is used to implement a garbage collector.

Signal The Signal class is used to represent all VHDL signals. Scalar signals contain the current value as an object of type Value. Composite signals consist of an array of Signal.

Process The Process class implements concurrent processes. Process are automatically suspended when they wait for signals, and they are resumed as soon as the signals are updated. Processes are implemented as light weight threads using a special library.

3.2 Code generation

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 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 identifies the hierarchy of VHDL entities. At the bottom of this hierarchy we find the individual processes. The compiler back end divides the processes into a number of partitions. Every partition is meant to be executed on a single node in the network.

The current version of the compiler contains a simple-minded partitioner that introduces a new partition to every process.

4 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. One part of the parker handles the internal simulation algorithm (in this case VHDL simulation), while the other part handles the simulation towards the other parallel kernels in the network. For this reason, the parker is usable for other simulation problems as well. The parallel kernel can host a multi-threaded simulation, which is the case in our VHDL simulation.

Together with the C++-code for the VHDL processes, the parallel kernel forms a *partition*. A

[2]Cad Language Systems Inc.

partition runs on a single network 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.

4.1 The internals

The parallel kernel uses two threads: one handling the incoming messages, and one controlling the VHDL simulation (the *event-handler*). All the specific routines for the VHDL simulation are in the event-handler. Furthermore it comprises a library of functions used by the generated C++ and the event handler. Figure 3 shows a partition with a parker and compiled VHDL code. The

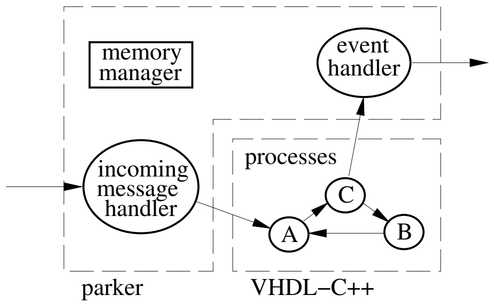

Figure 3: A partition consisting of a parker and C++. The arrows denote message-flow.

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 rollbacks, because they contain information that cannot be reproduced [5].

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.

4.2 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 [1, 6]; optimistic simulation uses *anti-messages* to discard incorrect messages [2]. 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 incoming signals used in the partition but driven on other ones, and another table containing all outgoing signals driven in the partition and used somewhere else.

5 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. The simulator 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 is vital for simulations that have nodes with unbalanced work load.

- 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.

Application code is not interpreted, but run on the host machine as native machine code. *Code augmentation* [7] is used to keep account of the execution cycles of the application. Every basic block of the application code is enhanced with some special instructions that increment the execution cycle counter and checks whether this node has exceeded its allocated running time. This augmentation is done with a special extension of the GNU C++ compiler.

6 Conclusions

The complete integrated system as described in this work is still under construction. However, preliminary results with the separate parts show that the approach is useful. Full results are expected in April 1994.

By replacing the VHDL specific parts in the parallel kernel, the system can be used to get statistics of other parallel event driver simulations. This requires only minimal change, because these parts are concentrated in one area.

Because an underlying architecture simulator is used, there is no need to run the simulation on actual parallel hardware. This removes any bias for a particular type of architecture, and allows us to select the optimal parallel system for a given type of simulation.

References

[1] 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.

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

[3] 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.

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

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

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

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