REFERENCES

[1] D. Abramson, R. Francis and M. Rawling, “Performance Bounds for the Conservative Parallel Discrete Event Simulation of VLSI Circuits and Systems”, *Australian Computer Science Communications*, Vol. 14, No. 1 (January 1992), Part B, pp. 753–767.

[2] P. J. Ashenden, *The VHDL Cookbook*, The University of Adelaide (Adelaide, South Australia, 1990).

[3] K. M. Chandy and J. Misra, “Asynchronous Distributed Simulation via a Sequence of Parallel Communications”, *Communications of the ACM*, Vol. 24, No. 4 (April 1981), pp. 198–206.

[4] B. A. Cota and R. G. Sargent, “An Algorithm for Parallel Discrete Event Simulation Using Common Memory”. *Proceedings of the 22nd Annual Simulation Symposium* (March 1989), pp. 23–31.

[5] A. Dewey and A. J. de Geus, “VHDL, Toward a Unified View of Design”, *IEEE Design & Test of Computers*, Vol. 9, No. 2 (June 1992), pp. 8–17).

[6] R. M. Fujimoto, “Parallel Discrete Event Simulation”, *Communications of the ACM*, Vol. 33, No. 10 (October 1990), pp. 30–53.

[7] J. L. Hennessy & D. A. Patterson, *Computer Architecture: A Quantitative Approach* , Morgan Kaufmann Publishers (San Mateo CA, USA, 1990).

[8] The Institute of Electrical and Electronic Engineers, *IEEE Standard VHDL Language Reference Manual*, Std. 1076–1987, IEEE (New York NY, USA, 1988).

[9] D. R. Jefferson, “Virtual Time”, *ACM Transactions on Programming Languages and Systems*, Vol. 7, No. 3 (July 1985), pp. 404–425.

[10] P. Konas and P. Yew, “Parallel Discrete Event Simulation on Shared–memory Multiprocessors”, *Proceedings of the 24th Annual Simulation Symposium* (April 1991), pp. 134–148.

[11] R. Lipsett, C. Schaefer and C. Ussery, *VHDL: Hardware Description and Design* , Kluwer Academic Publishers (Norwell MA, USA, 1989).

[12] A. Silberschatz, J. L. Peterson and P. B. Galvin, *Operating Systems Concepts* , 3rd ed., Addison–Wesley (Reading MA, USA, 1991).

[13] L. P. Soulé, *Parallel Logic Simulation: An Evaluation of Centralized-Time and Distributed-Time Algorithms*, Ph. D. Thesis, Department of Electrical Engineering, Stanford University (June 1992).

[14] J. C. Willis and D. P. Siewiorek, “Optimizing VHDL Compilation for Parallel Simulation”, *IEEE Design and Test of Computers*, Vol. 9, No. 3 (September 1992), pp. 42–53.

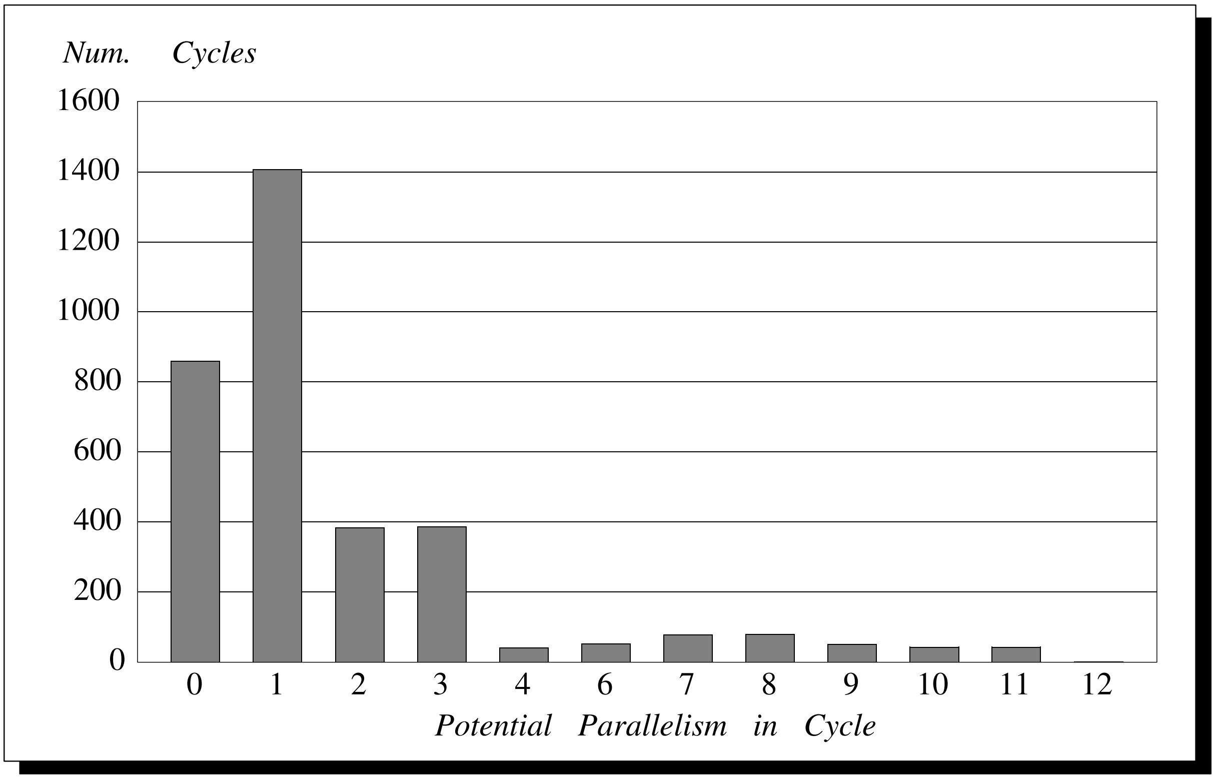

Figure 15. Histogram of cycle frequencies vs potential degree of parallelism for the DLX model.

speedup, other algorithms may be appropriate when all engineering considerations are taken into account.

As a first step towards understanding the trade-offs, we have implemented a simulator using the centralized event queue algorithm. Our initial measurements lead us to an improvement on the basic algorithm, namely parallelizing the signal update phase of the simulation cycle. With this modification in place, measurements of two sample models indicate that the technique is appropriate for use in simulating high-level and medium-level models on small-scale multiprocessors. This is exactly the case in a development environment early in the design cycle, where engineers using multiprocessor workstations need fast turn-around for proof of concept models. Measurements of a low-level model indicate that this algorithm is less appropriate for such models, since the overheads of the run-time system dominate over VHDL process execution. A simulator using an optimistic algorithm on a massively parallel multicomputer may be more appropriate at this later phase of the design cycle. Also, the development of compiler optimization techniques to cope with this problem deserves further research. Further measurement is needed using a more extensive benchmark suite, both on our simulator and on simulators using other PDES algorithms, to gain a clearer understanding of the trade-offs.

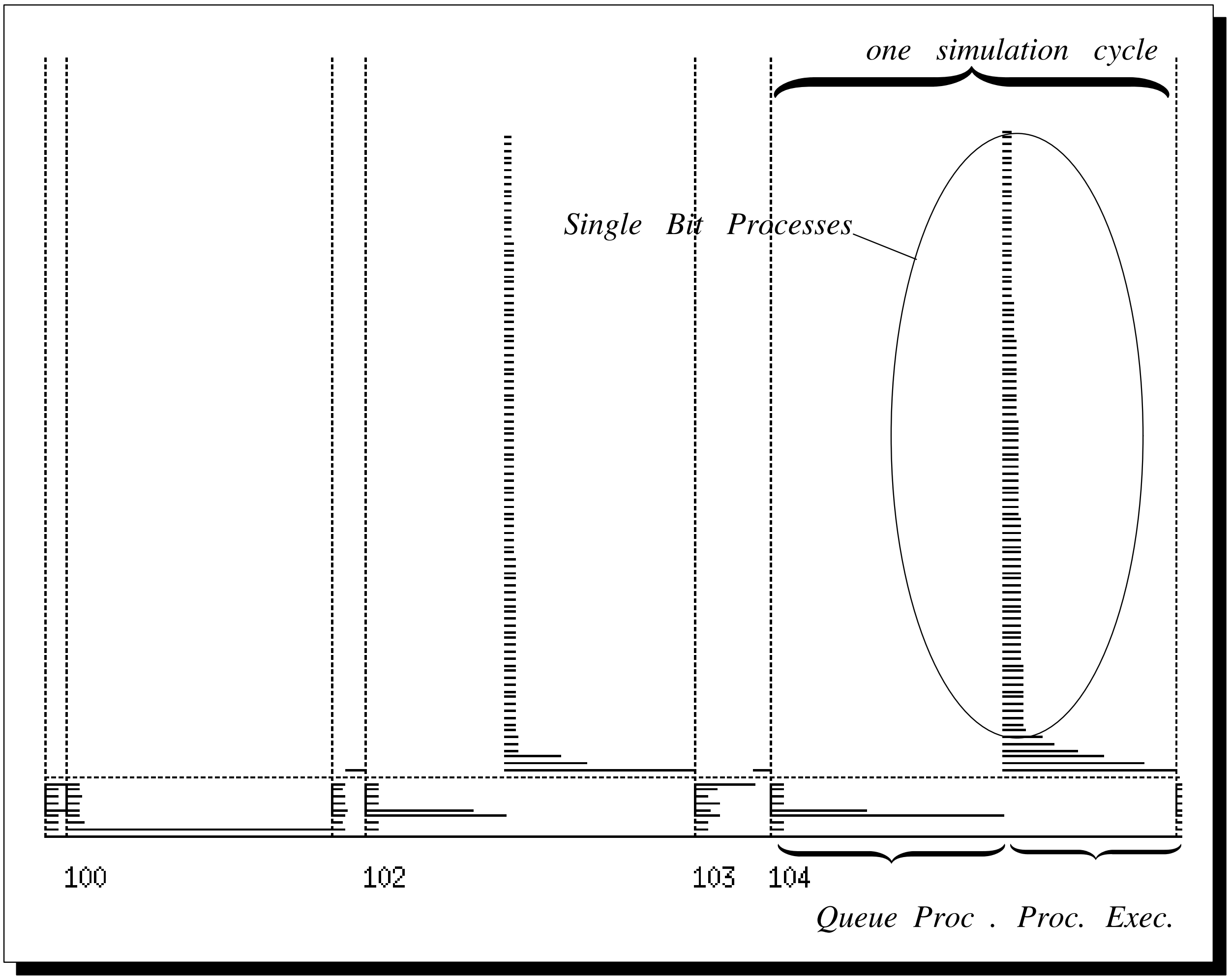

Figure 14. Excerpt of the parallelism profile for the multiplier model. Individual simulation cycles are shown along the horizontal execution time axis, each consisting of a queue processing phase and a process execution phase. The vertical axis represents the number of processes potentially executing in parallel. VHDL processes are above the dashed line, and queue update processes below.

5 CONCLUSIONS

In this paper we have surveyed a number of PDES algorithms, which serve to improve the performance of general simulation problems by making use of multiple processors. We have pointed out the potential advantages of using these algorithms in the context of executing VHDL simulation models. In particular, the productivity improvements and the reduced time to market that can be gained from shorter simulation run times makes parallel execution attractive. We have also pointed out some of the ways in which the algorithms must be adapted to form the core of a VHDL simulator , and suggested implementation strategies for both VHDL and VHDL-92.

An implementor of a VHDL simulator choosing a PDES algorithm must be aware of a number of factors: (1) the characteristics of the models expected to be simulated (implicit parallelism and data dependencies), (2) the relative complexities of implementing the algorithms, and (3) the characteristics of the target machine. While it is generally believed that optimistic algorithms give the best performance and

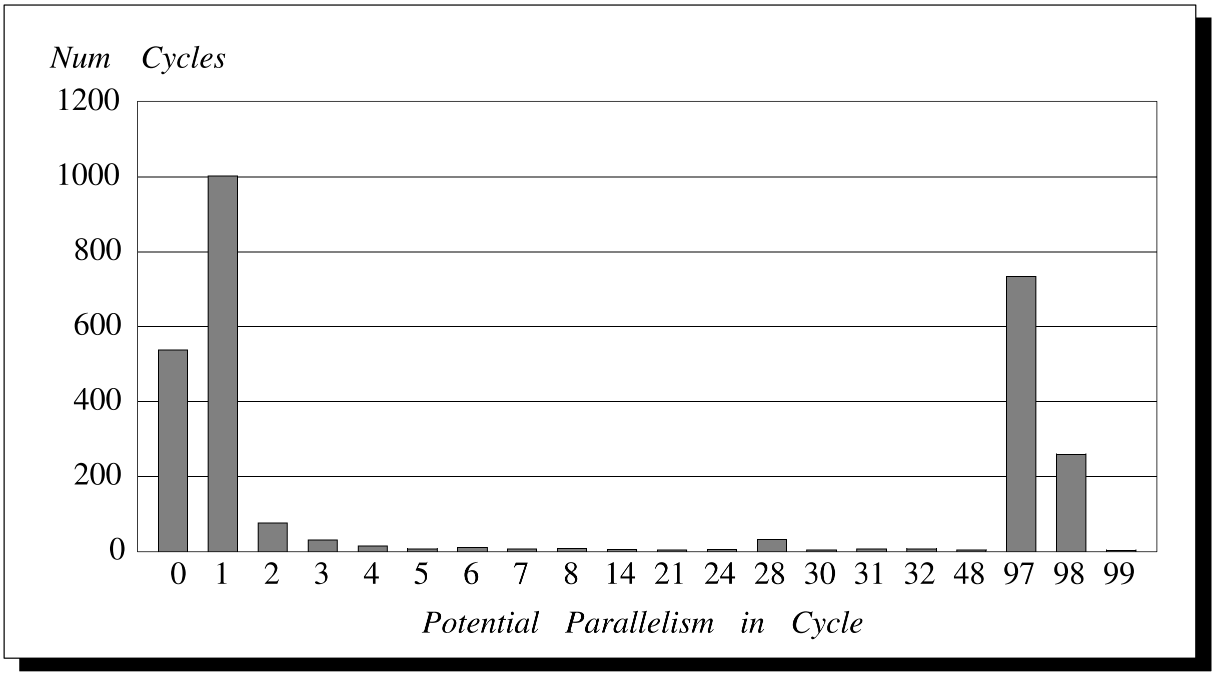

Figure 13. Histogram of cycle frequencies vs potential degree of parallelism for the multiplier model.

more even. However, as Figure 15 shows, the amount of concurrency obtained is very small. Two points are worth noting about this benchmark. Firstly, the model contains only 12 processes, since the system is modeled at a high level of abstraction. That is, the problem lies partially in the lack of parallelism implicit in the model rather than in the algorithm. Secondly, even though the amount of parallelism achieved is small, it is still significant. We consider this benchmark to be suitable for execution on small scale multiprocessors, such as the multiprocessor workstations and servers that have appeared recently.

4.3 Modified Implementation

In light of the proportion of simulation time spent in the signal-update phase, as revealed by Figure 12, we made a VHDL specific modification to the simulation algorithm. In addition to running each VHDL process as a separate thread, we multi-threaded the signal update phase. We replaced the single queue by multiple queues, each encapsulated in a thread which updates elements in the queue during the update phase of the simulation cycle. This corresponds to multiple concurrent 'kernel processes', to adopt the nomenclature of the language definition. Within a given cycle, there must be no dependencies between elements in different queues. We achieved this by using the object identification of the driven signal as a key to determine the queue into which to place transactions. Thus all transactions scheduled for a given signal are entered into the same queue, even if the transactions arise from different drivers in the model.

The multi-threaded signal update phase performs quite well, indeed for models such as the DLX, more concurrency is obtained in an 8-threaded update phase than in process execution. Two areas specific to multi-threaded signal update which could be further investigated are improving the distribution of elements across the queues, and determining an optimal number of queues and update threads.

ter dominates, in VHDL the reverse is the case, as Figure 12 clearly shows. This result lead us to a modification of the general algorithm, to be described in section 4.3.

The results obtained for the low-level 32-bit multiplier are superficially encouraging. Figure 13 reveals a large percentage of cycles with more than 90 processes resumed out of a total of 99 in the model. However, the section of the parallelism profile for this model given in Figure 14 reveals a problem. This section is typical of the entire parallelism profile. It reveals that the vast majority of processes executing within a cycle in which 90 or more processes are resumed only execute for a very short time; usually they perform a single signal assignment. There are two problems with this kind of behavior: the scheduling overhead is likely to dominate the execution time for such processes, and, within such cycles, a small number of processes execute for a significantly longer time than the majority. The execution of these processes must be completed before the next simulation cycle may begin. It should be noted that the first problem is a result of mapping each VHDL process to a separate logical process. Even in a sequential simulator, logical processes still need to be scheduled. The best strategy for a solution to both problems appears to be a modified code generator which ‘bunches’ several small VHDL processes into a single logical process, resulting in a decrease in the number of processes, but an increase in their granularity to a more useful size. The effectiveness of such a strategy remains to be explored.

The parallelism profile for the DLX model does not exhibit this behavior to the same degree: the granularity of processes in this model is greater on average and also

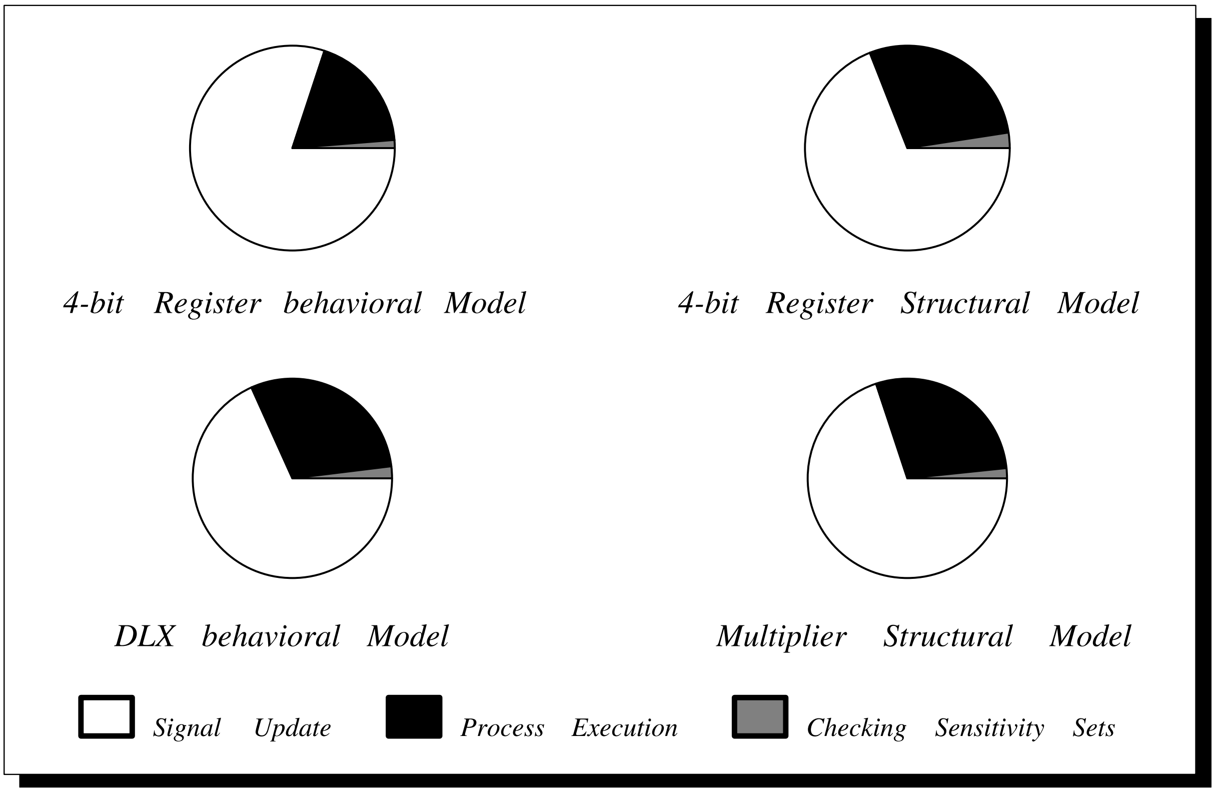

Figure 12. Proportion of execution time spent in the different phases of the simulation cycle. The 4-bit register models describe a simple transparent latch register. The behavioral models consists of two processes, and the structural model consists of D-flipflop and gate elements. The DLX and multiplier models are larger high-level and low-level models respectively.

To enable full concentration on the development of the kernel we chose to utilize an off-the-shelf front-end analyzer and code generator. This saved in development cost, but also precluded research in a number of interesting areas. For example, it made it difficult to perform static analysis of models for model-specific adaptation in the kernel. It also precluded bunching of several (small) VHDL processes into a single simulation process, to control the granularity of the final processes.

The front-end we chose was the Cad Language Systems Inc. (CLSI) Retargetable VHDL Code Generator (RVCG). Kernels using this system are library files which are linked with the generated code object files for a given model to produce a standalone executable for that model. In broad outline a simulation run proceeds as follows:

1. Elaboration: the kernel initially has control, and calls functions provided by the generated code to elaborate the model. This elaboration process (which is single-threaded) includes calls to kernel entry points to create process, signal, driver and other VHDL objects. Control returns to the kernel at the completion of elaboration.
2. Initialization: the kernel executes all VHDL processes. Each VHDL process executes as a separate thread. Since the sequential code of each VHDL process is part of the generated code, the kernel must callback into this code as necessary.
3. Execution: the kernel performs the simulation cycle as given in the Language Reference Manual[1], re-activating VHDL process threads as required.

The simulator currently runs on a sequential machine (a single processor SparcStation), and is being used to collect measurement in accordance with the first aim stated above. The kernel design is structured in such a way that porting to a parallel architecture should present minimal problems. We have developed the prototype kernel to stage where it supports a subset of VHDL sufficient to represent any elaborated model. Some structural modeling aspects are not supported, but they can be manually translated into the behavioral subset which is supported. At this stage the instrumentation necessary to report structural dependency information is still to be developed.

4.2 Results

The results shown here are derived from two benchmark models. The first is a register transfer level model of the non-pipelined DLX processor, described in [7]. This serves as a sample high-level model, as the datapath is modeled using delta-delay components, and the control section and test bench memory are behavioral. The second benchmark measured was a low level model of a 32-bit shift-and-add multiplier, consisting of flip-flop, full-adder and gate components, with a simple state machine controller. This benchmark includes simple timing for each of the components in the model.

The first result of interest concerns the proportion of execution time spent in updating signal values and determining the set of processes to resume in a given cycle, versus actually executing those processes. Whereas in general simulations the latknown resource allocation techniques used by operating systems and described in texts on the subject (for example, [12]). However, a simulator using a distributed PDES algorithm must also allow for the fact that different processes requesting access to shared variables may be at different simulation times. To preserve the language semantics, the implementation must synchronize processes that use shared variables, so that accesses occur in the correct order in simulation time. An obvious way to do this is to encapsulate each shared variable in a monitor process, which is responsible for granting access to one process at a time, and for serializing requests in the correct simulation time order. A process requests access to a shared variable by sending a time-stamped request message to the monitor process. When exclusive access is granted, the monitor process replies with the value in another message with the same time-stamp. When the requesting process has completed its access statement, it relinquishes exclusive access by sending a third message, possibly containing an updated value.

---

[1] [8], Chapter 12.

In a simulator using a conservative algorithm, the monitor process treats the request message channels as input links, complete with link clocks, and applies the conservative approach to serialize requests correctly. On the other hand, in a simulator using an optimistic algorithm, a monitor process simply queues requests and grants exclusive access provided requests arrive in non-decreasing time-stamp order. When a straggler request arrives, it rolls the value of the shared variable back, and sends anti-messages for the grant messages it has since sent to other processes. These, in turn, must roll back and re-execute the access statements.

4 A CENTRAL QUEUE PDES IMPLEMENTATION

The Vsim Project at the University of Adelaide aims to develop a number of concurrent simulation kernels employing various of the different algorithms surveyed above. The aim is to use these simulators to fulfil a number of goals:

- to measure of the amount of parallelism implicit in typical VHDL models,
- to determine types of structural dependencies via kernel instrumentation, and
- to use the information obtained from prototype kernels to guide the design and development of more advanced kernels.

4.1 Initial Implementation

For the production of the first simulation kernel we chose to adopt the centralized queue approach. A number of factors motivated this choice. Firstly, the way in which VHDL semantics are expressed seems to favour simulators with a central queue. While this might be regarded as a deficiency of the VHDL definition in terms of abstraction, it makes for a relatively simple mapping of the general algorithm to the specifics of VHDL. Our experience is that the process of development of such a kernel leads to a greater understanding of VHDL semantics, which can then be applied in the development of kernels for which the mapping is non-trivial. Secondly, using the centralized queue approach identifies a useful lower bound on the amount of parallelism available for a given model. Thirdly, a centralized kernel is easiest to instrument, whereas instrumentation of a distributed system would present additional problems.

case is $\delta_{min} = 0$ fs, in which case the process cannot send a transaction until its local clock passes the time-stamp of the transaction.

If a process uses inertial delay, deletion of a scheduled transaction with time-stamp earlier than $T+\delta_{min}$ depends on the values in transactions scheduled by subsequently executed signal assignments. In general, these are determined at run time, and cannot be predicted, thus a transaction may not be sent until the local clock passes the time-stamp of the transaction. However, in many cases it may be possible to determine these values through static analysis of the model code (e.g., using the techniques suggested by Willis and Siewiorek [14]), and thus guarantee that transactions won't be deleted. In these cases, a transaction may be sent before the local clock reaches its time-stamp, thus increasing the parallelism achievable. This is an example of the use of *lookahead* in a conservative algorithm. The literature on conservative algorithms suggests that the quality of lookahead information used is important in improving the performance of these algorithms (see [6]).

The proposal for VHDL-92 to include a *pulse rejection limit* in an inertial delay specification provides opportunity to improve on the above the behavior. The pulse rejection limit is a time interval, counting backwards from the delay value specified in a signal assignment, in which previously scheduled transactions may be deleted. Any transactions between the current time and the beginning of this interval are not deleted. Thus if the simulator can determine a lower bound on delays (δ_{min}) and an upper bound on pulse rejection limits (ρ_{max}), transactions with time-stamps between T and $T + (\delta_{min} - \rho_{max})$ can be sent. However, the difficulty mentioned above also applies to determining ρ_{max}, namely that, in general, the interval is determined at run time.

While considerable complexity is added to conservative algorithms in handling these semantics, time warp algorithms can deal with transaction deletion much more simply. When a transaction is scheduled in a time warp simulator, the process can send it immediately. If the transaction must subsequently be deleted, the process can simply send an anti-message to cause cancellation of the transaction and roll back of any optimistically executed computation. Note, however, that this does not prevent application of lookahead techniques to avoid sending a transaction. If the simulator can determine that a transaction will be deleted, it can avoid the cost of sending and subsequently cancelling the transaction.

3.4 Shared Variables

The proposal for shared variables in VHDL-92 creates considerable extra complication for distributed PDES algorithms. The semantics are that shared variables may be declared in blocks that enclose processes, and the processes may refer to shared variables using *access statements*. Each access statement names a set of shared variables, and the runtime system is required to guarantee exclusive access to these variables while the sequential statements in the access statement are executed. The mechanism used by the runtime system to provide mutual exclusion between processes is required to be deadlock free. This is made simpler by the fact that processes may not suspend whilst within an access statement, nor may they dynamically nest access statements.

In a sequential simulator, or a parallel simulator using a central event queue algorithm, shared variables can be managed by the kernel. It can apply any of the well

```
initialize
loop
    set local-clock to the earliest input link-clock
    if any input link with link-clock = local-clock is not empty then
        receive first event from one of these links and update signal
        if process is triggered then
            while there is an input link with link-clock.time = local-clock.time loop
                set local-clock to the earliest input link-clock
                if any input link with link-clock = local-clock is not empty then
                    receive first event from selected link and update signal
                else
                    wait until an event arrives on any link with link-clock = local-clock
                end if
            end loop
            — after loop, all input links have link-clock at later time-step than local-clock
            resume the postponed process — may involve scheduling events on output links
        end if
    else
        wait until an event arrives on any link with link-clock = local-clock
    end if
end loop
```

Figure 11. Modified conservative PDES algorithms that correctly handles postponed processes.

3.3 Transport and Inertial Delay Semantics

The semantics of transport and inertial delay in VHDL necessitate further modifications to the basic PDES algorithms, due to the fact that a transaction scheduled by one signal assignment may subsequently be deleted by another. If a signal assignment specifies transport delay, any previously scheduled transactions on the signal with time-stamps later than or equal to that of the new transaction are deleted. If inertial delay is used, any previously scheduled transactions with earlier time-stamps are also examined, and some of those may be deleted. In both cases, the transactions to be deleted can not, in general, be determined until run-time, since a model may include dynamically variable delay values in signal assignments. We compare this to other simpler hardware description languages, such as that used by Soulé in his parallel simulators [13], in which a fixed statically determined delay is specified for each process, and simple transport delay semantics are used. This means that each signal assignment schedules a transaction on a signal for a later simulation time than transactions from previously executed assignments, and no transactions need be deleted.

For parallel simulators using conservative algorithms, the prime concern in this context is that transactions be sent over message links in non-decreasing time-stamp order. If transport delay is used, the simulator needs to determine a lower bound on the delay value for any assignments to a signal in a process (δ_{min}), since transactions later than the current simulation time (T) plus this delay may be deleted, and therefore should not be sent. (This delay is the same value used for time-stamping null messages.) The process should hold a transaction scheduled on an output link until $T+\delta_{min}$ passes the time-stamp of the transaction. The worst way, but does not necessarily resume execution in the simulation cycle in which a triggering event occurs. Instead, it is resumed after all of the delta cycles for that time-step have been completed. While these semantics are relatively straightforward to implement in a sequential simulator or a central event queue parallel simulator, care must be taken in distributed parallel implementations.

If a conservative algorithm is used, a postponed process may not respond to a transaction until it can guarantee that no other transaction at the same time-step will arrive. Figure 10 illustrates a scenario where the normal conservative algorithm would fail for a postponed process.

A possible variation to the conservative algorithm of Figure 9 for use by a postponed process is shown in Figure 11. In this case, the inner loop performs the conservative algorithm on input links, receiving transactions and updating signals, so long as their link clocks remain in the current time-step. When all input links have progressed to a later time-step, the process can guarantee that there will be no more delta cycles, so it can resume the postponed VHDL process body. The process body may schedule transactions on output links, but they must have non-zero delay, otherwise further delta cycles in the same time-step would result, violating the VHDL-92 semantic requirements.

A simulator using the time warp algorithm can deal with postponed processes relatively easily. A postponed process may resume immediately it is triggered, on the optimistic assumption that there will be no further delta cycles in the time-step. (The requirement that transactions it schedules have non-zero delay still applies, however.) An input transaction arriving subsequently with the time component of the time-stamp equal to the time component of the local clock when the process was resumed should be treated as a straggler. A simple expedient for implementing this is to reserve a special value of delta cycle count which is deemed to be greater than any other value, and setting the delta component of the local clock to this value when the postponed process is resumed.

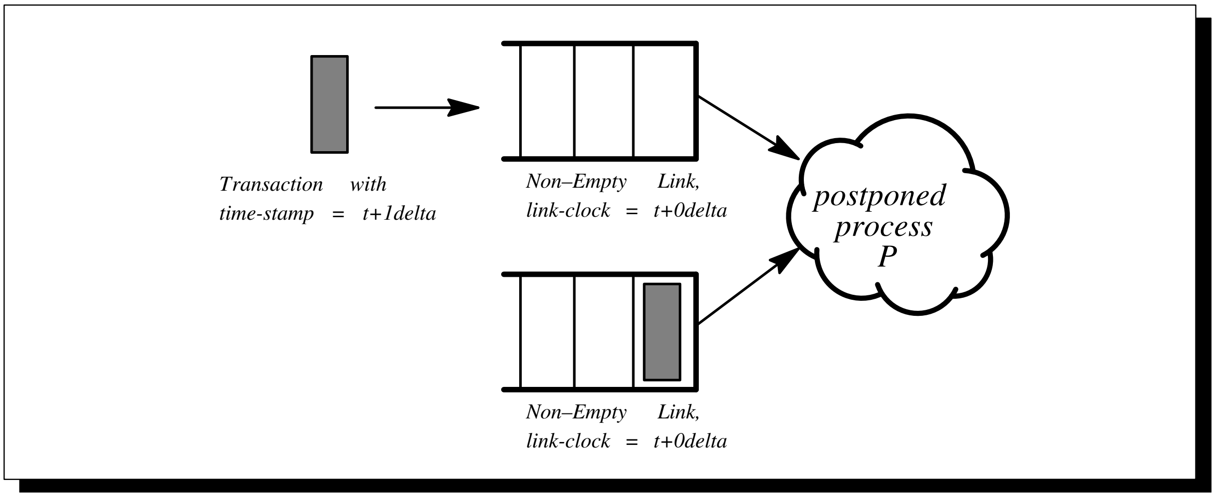

Figure 10. A scenario where the normal conservative algorithm would fail for a postponed process. The postponed process P has two input links, one of which is empty, with a link clock at *t*+0delta. The other contains a queued transaction with time-stamp *t*+0delta, which causes a VHDL event that triggers the process. Immediate resumption of P would be incorrect, since the subsequent arrival of the transaction at simulation time *t*+1delta should cause the corresponding signal to be updated first.

The semantics of the VHDL simulation cycle also require another variation to the general PDES algorithms. If two transactions with the same time-stamp are sent to a logical process, one of which causes a VHDL event resulting in resumption of the VHDL process, both transactions must be received and the signal value updated before the VHDL process is resumed. Compare this with the general conservative algorithm shown in Section 2.2, in which a process responds to an event as soon as the local clock reaches the link clock. The variation to the general algorithm shown in Figure 9 avoids the possiblity of the VHDL process being resumed in response to one transaction, only to have another transaction with the same time-stamp arrive later .

The outer loop performs the normal conservative algorithm, so long as the process is not triggered by a VHDL event on a signal in its sensitivity set. When it is triggered, the inner loop performs the conservative algorithm on input links, receiving transactions and updating signals, so long as there are link clocks equal to the local clock. When all input links have progressed to a later time, the process can guarantee that there will be no more transactions for the current time, so it can resume the VHDL process body.

A simulator using the time warp algorithm requires relatively little modification to deal with this aspect of the semantics of VHDL process resumption. When a VHDL process is triggered by a VHDL event on a signal, it may resume immediately, on the optimistic assumption that no further transactions will arrive with the same time-stamp. If an input transaction subsequently arrives with time-stamp equal to the local clock when the VHDL process was resumed, the transaction should be treated as a straggler.

A further complication arises when ‘postponed processes’ proposed for the VHDL-92 language, are considered. Such a process is sensitive to VHDL events in the normal

```
initialize
loop
    set local-clock to the earliest input link-clock
    if any input link with link-clock = local-clock is not empty then
        receive first event from one of these links and update signal
        if VHDL process is triggered then
            while there is an input link with link-clock = local-clock loop
                if any input link with link-clock = local-clock is not empty then
                    receive first event from selected link and update signal
                else
                    wait until an event arrives on any link with link-clock = local-clock
                end if
            end loop
            — after loop, all input links have link-clock at later time than local-clock
            resume the VHDL process — may involve scheduling events on output links
        end if
    else
        wait until an event arrives on any link with link-clock = local-clock
    end if
end loop
```

Figure 9. Modified conservative PDES algorithms that correctly handles VHDL process resumption, based on the VHDL simulation cycle semantics.

4. waiting until the resumed processes have all suspended.

If a process that is resumed in step 3 schedules a transaction with zero delay, the corresponding signal is not updated until the next iteration of the simulation cycle. No process sees the effect of the transaction until the next cycle. This form of delay is called a *delta delay*, since the simulation time is the same for the next cycle, but some apparent time has passed. It is quite possible to write a VHDL model whose meaning depends on the correct ordering of process executions in successive simulations cycles after a delta delay, as shown in Figure 8. Indeed, zero delay behavioral models, and some low level models for switched circuit elements, require these semantics to be observed by a simulator.

The potential problems that arise lead simulator implementors to treat simulation time as a pair, consisting of a time value in fs, and a count of delta cycles at that time step. This presents no great difficulty in a sequential simulator, or in a parallel simulator using the central event queue technique, since the kernel can implement the VHDL simulation cycle as described above. However, in a simulator using a distributed algorithm, local clocks and time-stamps in transaction messages must include both the time value and the delta cycle count, and comparisons must be made using both elements.

```
architecture a of e is

    signal s1, s2 : bit := ’0’;

begin

    p1 : process
    begin
        wait for 1 fs;
        s1 <= ’1’ after 0 fs;
        wait;
    end process p1;

    p2: process
    begin
        wait for 1 fs;
        s2 <= ’1’ after 0 fs;
        wait;
    end process p2;

    p3: process
    begin
        wait until s1 = ’1’;
        assert s2 = ’1’ report ”Wrong order!”;
        wait;
    end process p3;

end a;
```

Figure 8. A simple model that depends on correct ordering of process resumptions in the presence of delta delays. If correct VHDL semantics are observed, p1 and p2 are resumed in the first cycle at time 1 fs, then p3 is resumed after a delta cycle and no assertion violation occurs. On the other hand, if a simulator were to resume p1 first, then p3 before p2, p3 would see the value ’0’ on s2, and raise an assertion violation.

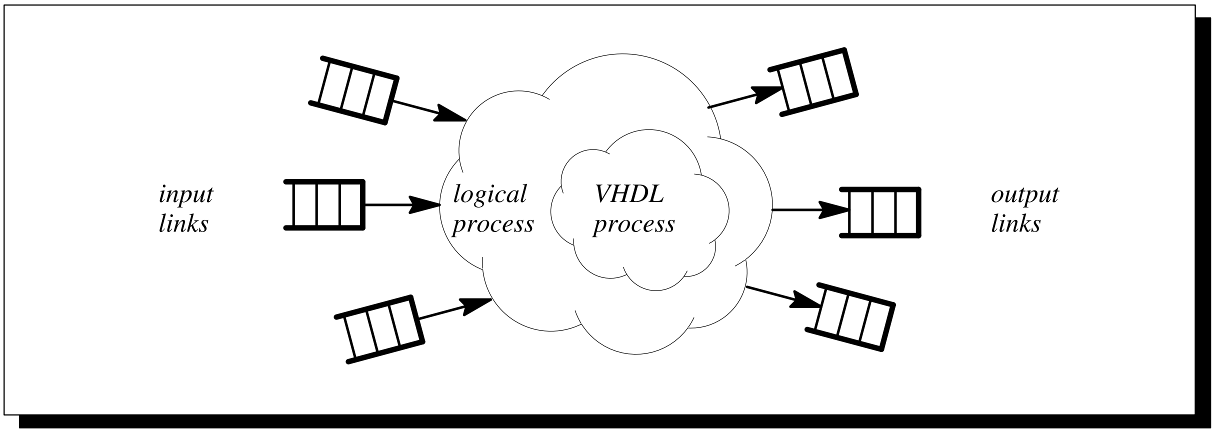

Figure 7. Implementation of a simulation process as a VHDL process encapsulated in a logical process.

In addition to the processes described above that define the behavior of a model, VHDL also allows the designer to specify a block of code for resolving the value of a signal driven by a number of sources. When a transaction occurs on any of the sources, a resolution function is invoked, which calculates the new value from the values of the sources. One way to implement resolution functions in a PDES simulator is to encapsulate the resolution function in a PDES process. The process has input links to receive transaction messages from each of the sources for the resolved signal, and has output links to processes that use the resolved signal value. The process responds to input messages by computing the new signal value and scheduling a transaction message on its output with no delay. (This must be a delay of 0fs+0delta to preserve VHDL semantics; see discussion below on the VHDL semantics of simulation time.)

A related issue is the implementation of VHDL GUARD signals, which require the evaluation of a guard expression. The implementation can be done similarly, with the guard expression being encapsulated in a PDES process. In this case, the input links to the process receive transaction messages for signals denoted in the guard expression, and the output link corresponds to the GUARD signal itself. Again, the process must evaluate the expression and schedule an output transaction with no delay .

3.2 Simulation Time and Delta Delays

Whereas the general PDES algorithms discussed in Section 2 use a simple view of simulation time, VHDL simulators require a structured representation for simulation time. This comes about from the specification of the run-time semantics of VHDL in terms of a sequential iteration of a simulation cycle. The cycle basically involves the following steps:

1. advancing simulation time to the time of the earliest scheduled transaction,
2. updating signals with the values of transactions scheduled for this simulation time,
3. resuming processes sensitive to VHDL events resulting from the signal updates,

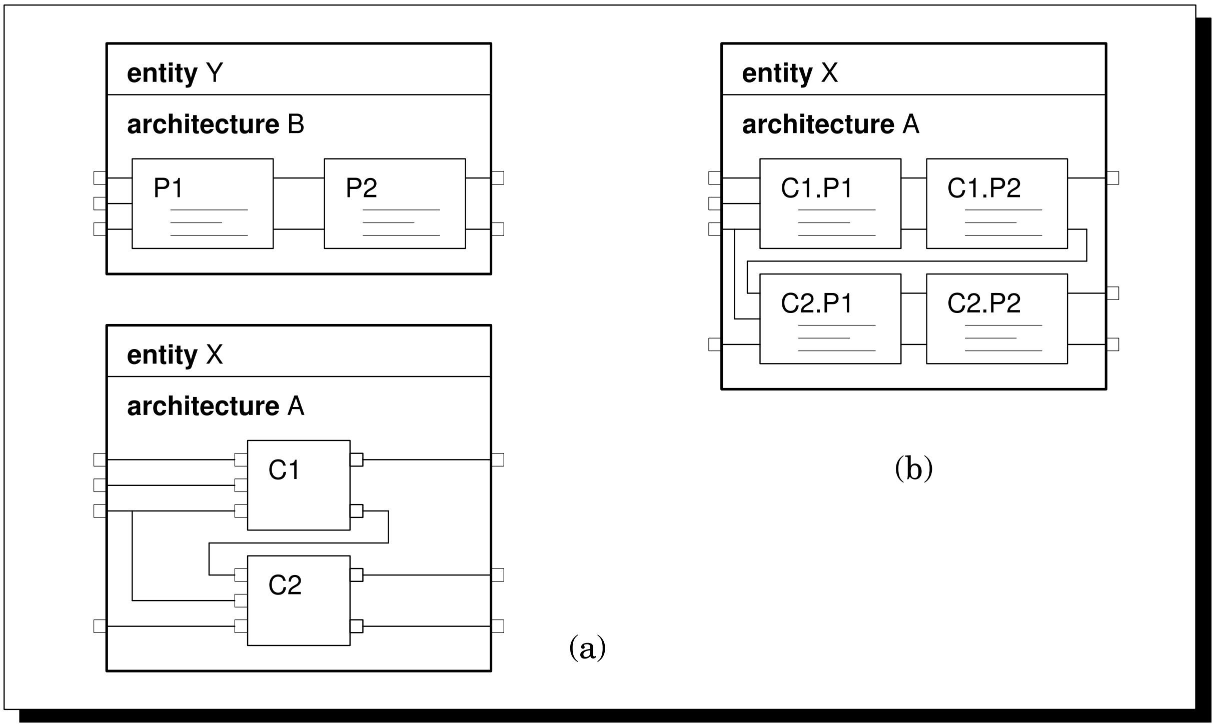

Figure 6. Elaboration of a simple example hierarchical model. In (a), the entity Y has an architecture B, which contains two processes P1 and P2. The entity X has an architecture A, which contains two component instances C1 and C2, both of which are bound to the pair Y/B. In (b), the structure is elaborated, with architecture body B substituted for each of C1 and C2. The result is a network with two instances of P1 (C1.P1 and C2.P1) and two instances of P2 (C1.P2 and C2.P2).

event message is received, and executes code to respond to that message. A VHDL process, however, only resumes when a wait statement terminates. This may occur as a result of:

- a VHDL event on a signal in the sensitivity set on which the process has suspended (if no wait condition is specified),
- a VHDL event on a signal in the sensitivity set, and the wait condition being true (if a condition is specified), or
- the time-out period expiring (if one is specified).

However , this does not fully describe the conditions to which the process must respond. It must also respond to any transaction that arrives on any input link and update the input value accordingly. In the case of conservative PDES algorithms, the process must update its local clock using the link clocks of all input links, even if the VHDL process is not waiting on some of the signals. In the case of optimistic algorithms, the process must respond to straggler messages on nput links for signals the VHDL process is not waiting for. These semantics suggest an implementation of a process in terms of a logical process, which executes the PDES algorithms, and which encapsulates a VHDL process (see Figure 7). The logical process acts as a local kernel for the VHDL process, filtering input messages, and resuming the VHDL process when the wait statement on which is is suspended terminates. The VHDL process uses the services of the local kernel to send output messages as described throughout the rest of this paper.

sponds to a VHDL 'transaction', that is, a time-stamped communication of information from one process to other processes. VHDL distinguishes the special case of the new value on a signal being different from the old value with the term 'event'. To avoid confusion in the following discussion, we will use the terms 'transaction' when referring to a PDES event in the context of VHDL, and the term 'VHDL event' when referring to a transaction that changes a signal value. The reason for making this distinction is that the PDES algorithm used must communicate transactions, not just VHDL events, between processes connected by signals.

3.1 Processes and Signals in VHDL Models

PDES algorithms share a common view of a model to be simulated: they treat it as a collection of logical processes, which schedule events. Each logical process is a piece of sequential code, which is triggered by events scheduled for it by other logical processes. In a central queue PDES simulator, events are sent to a logical process by the kernel process. In a distributed PDES simulator, events are sent from one logical process to another via event message links.

In a VHDL model, the processes are explicitly defined, and the links are defined by the signal nets. (Compare this with some other modeling languages, in which the logical processes and event scheduling links are implicit, and must be extracted by code analysis.) A VHDL model consists of a hierarchical structure of component instances and processes, interconnected with signals. These are specified using VHDL entities (which represent objects to be used in the design), and corresponding architecture bodies (which implement the functionality of the entities). An architecture body can contain component instances, with ports that are interconnected by signals. Figure 6(a) shows an example of a hierarchical model.

Each component instance is bound to an entity/architecture pair (by a configuration specification), and can be elaborated by substituting the corresponding architecture body. The signals of the substituted architecture body are connected with those of the enclosing body via the component instance's ports. The component instances in the substituted body can then be recursively elaborated. The recursive elaboration bottoms out when an architecture body is reached that contains only processes, and no further component instances. A process is a sequential body of code that can affect the values of signals using signal assignment statements. A process containing a signal assignment for some signal s is said to contain a driver for s.

This elaboration procedure is carried out statically on a design as a precursor to executing the simulation. The result of elaboration is a collection of VHDL processes, each containing drivers for one or more signals (Figure 6(b)). This corresponds to the view used by PDES algorithms as follows. Each VHDL process corresponds directly to a PDES logical process, and each signal net is implemented as a collection of PDES event message links. Whereas in the general PDES algorithms, a logical process sends an event message directly to a destination process, in a VHDL simulation a VHDL process schedules a transaction on a signal. This involves the driver for the signal duplicating ('fanning-out') the transaction to all other VHDL processes which reference the signal, namely, all those VHDL processes which connect to the signal net for input.

The semantics of VHDL process resumption also require an extension of the general PDES algorithms. In the general algorithms, a logical process is resumed when an only store enough state history to roll back to GVT. State for earlier times can be discarded. GVT is also used to determine when operations such as I/O that cannot be rolled back may be committed.

There are a number of optimizations to the time warp algorithm that may be used in a VHDL simulator. For example, when a process rolls back, instead of immediately cancelling events it has generated, it may first check to see if the new events it generates are the same as the old ones. If so, the original events need not be cancelled. This is known as *lazy cancellation.* In the best case, lazy cancellation may allow execution to complete faster than critical path analysis would suggest. This may occur if the optimistic execution of a process turns out to have produced the correct results when a straggler arrives. The successor processes will already have completed correct execution, and need not be rolled back. In the worst case, delaying cancellation may allow incorrect computation to spread further through the network than normal cancellation, thus requiring more roll back in other processes and prolonging execution time for the simulation. The actual behavior can be expected to be somewhere between these extremes, depending on the model and input data.

Another optimization of time warp which may be effective in circuit simulation is *lazy re-evaluation.* If processing a straggler event does not change the state of a process, then it is not necessary to re-process the events that were rolled back, since they would produce exactly the same results. Instead, the process can ‘jump forward’ again to the virtual time to which it had previously made progress. This optimization may prove valuable in models that do not rely on internal state to respond to events, such as combinatorial circuit elements.

Optimistic algorithms rely on the frequency of roll back being sufficiently low that it does not significantly affect the progress of the simulation. However, in extreme cases, the processes could spend more time performing roll backs than useful computation, which may cause the parallel simulation to execute slower than a sequential simulation. Optimistic algorithms also presume that the space need to save state information is not excessive. Compared to conservative algorithms, optimistic algorithms avoid the possibility of deadlock, thus providing more opportunity for parallelism, albeit at the expense of performing some computation which is subsequently discarded. The time warp algorithm is significantly more complex than conservative approaches, and there is no guarantee that it will induce a better speedup. Experimental evidence is needed to determine which technique produces best results for different types of models.

3 CHARACTERISTICS OF VHDL MODELS

Each of the PDES algorithms surveyed in the previous section are general purpose, and can thus serve as the basis for a parallel simulator for VHDL models. However, there are a number of aspects of the semantics of VHDL that affect how these algorithms are used. In this section, we comment on some of these aspects and discuss there implications for simulator implementations. We assume that the reader is familiar with VHDL, and refer the less familiar reader to a number of sources that describe the language [5, 2, 11] and to the IEEE Standard Language Reference Manual [8].

One important distinction to draw when discussing PDES algorithms in the context of VHDL simulation is the different use of the term ‘event’. A PDES ‘event’ corre-

Secondly, the extra message traffic generated by sending null messages may be considerable. However, this can be reduced by sending null messages on a demand basis. If a process must block, it first requests null messages from those processes on the sending sides of the links on which it is about to block.

Deadlock detection and recovery involves a mechanism (such as that described in [3]) to detect deadlock in the distributed computation. Deadlock can then be broken by processing the messages with the smallest time-stamp. An advantage of this technique is that cycles in which the sum of the minimum time increments is zero are permissible. A disadvantage in typical implementations is that the mechanism to detect and recover from deadlocks is centralized and therefore may become a bottleneck. Furthermore, using that mechanism, only global deadlocks are detected, allowing local deadlock in a subnet of the simulation to remain undetected and reduce parallelism.

Shared memory implementations of conservative algorithms make possible several optimizations [4, 6, 10]. For example, deadlock avoidance can be implemented by having a process which is about to block on an empty link directly inspect the local clock of the process at the other end of the link. This obviates the null message traffic that would otherwise be required.

2.3 Optimistic Distributed Algorithms

One major drawback with conservative algorithms is the fact that they may not fully exploit the potential parallelism in a model. If there is any chance that one process may affect another, then the process executions are serialized. On the other hand, optimistic algorithms, using the *Time Warp* paradigm [9], allow execution to proceed without regard to causality violations, and take corrective action when violations are detected.

As with conservative algorithms, each process includes a local clock which measures simulation time for that process (called virtual time), and a queue of time-stamped event messages to be processed. However, with optimistic algorithms, each process proceeds in simulation time at its own pace, on the assumption that no causality errors result. If at some point in simulation time an event arrives with time-stamp less than the local clock (a *straggler* event), then the receiving process must roll back. It must undo all state changes back to a simulation time at or before the time-stamp of the straggler, and recommence. In order to be able to perform roll back, a process must save its state from time to time, and must also keep a list of all messages it has received so that it can process them again after rolling back. Furthermore, it must keep a record of event messages it has sent to other processes, since some of them may result from execution which is subsequently rolled back. When roll back occurs, the process must send an ‘anti-message’ for each message sent with time-stamp greater than the straggler. If the message to be cancelled is still in the destination’s input queue, it is simply deleted. On the other hand, if the destination process has already processed the message, it too must be rolled back. This process continues recursively throughout the network of processes until all incorrect computation has been undone.

In order to prevent unbounded growth of saved state, time warp simulators use the notion of Global Virtual Time (GVT), which is less than or equal to all processes’ local virtual time clocks. Since no process need ever roll back past GVT, a process need

```
initialize
loop
    set local-clock to the earliest input link-clock
    if any input link with link-clock = local-clock is not empty then
        remove first event from one of these links
        process the event — may involve scheduling events on output links
    else
        wait until an event arrives on any link with link-clock = local-clock
    end if
end loop
```

Figure 4. Conservative PDES algorithm.

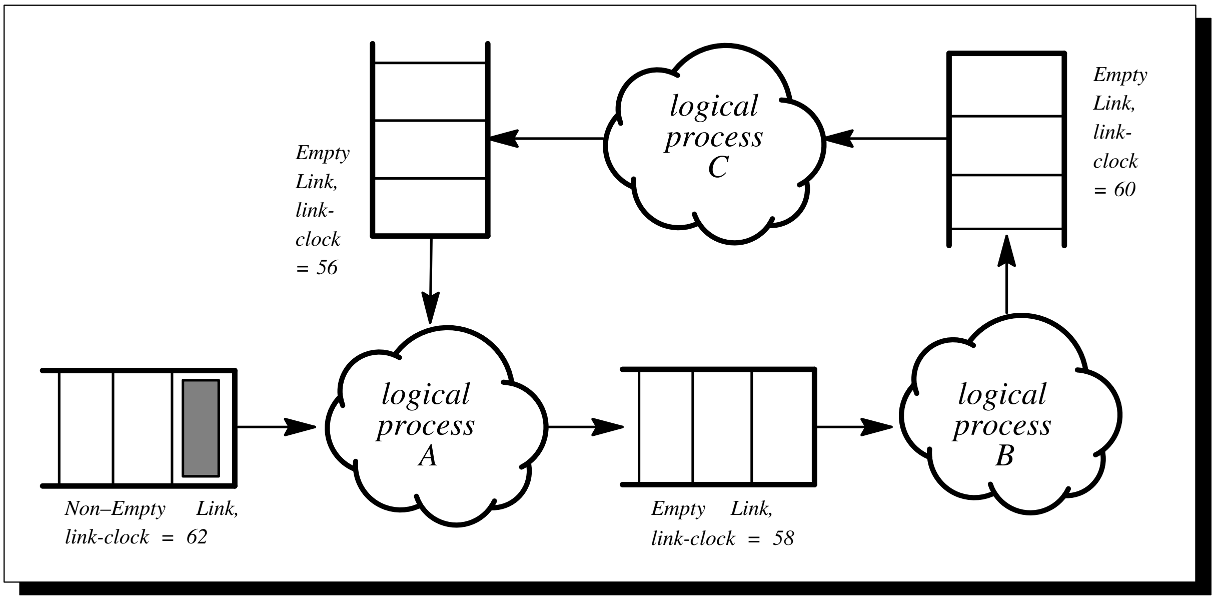
logical process C
Empty Link, link-clock = 60
Empty Link, link-clock = 56
logical process A
logical process B
Non–Empty Link, link-clock = 62
Empty Link, link-clock = 58

Figure 5. A deadlocked network of logical processes. Process A is waiting for a message from C, since the link from C has earlier link clock than A's other input link. Processes B and C are also waiting on empty links in the cycle. No process can proceed, so the network is deadlocked.

dealing with deadlock include deadlock avoidance, and deadlock detection and recovery.

Deadlock avoidance involves the use of null messages, which carry no data and are sent solely for synchronization purposes. When a process finishes processing an event which arrived at time t it sends a null message on all of its output links with time-stamp $t+\delta_{min}$, where δ_{min} is the lower bound on the time-stamp increment for a message through the process. This increment can be thought of as the minimum propagation delay through the corresponding physical process. The null message indicates to the receiver that the sender will not generate an output event with time-stamp earlier than $t+\delta_{min}$, so the receiver may safely update its local clock. It also generates null messages on its output links, adding its minimum time-stamp increment. In this way, null messages are propagated around a cycle until some local clock value reaches the link clock of some other link not in the cycle.

There are two possible problems with this deadlock avoidance technique. Firstly, it will only work if the sum of the time-stamp increments around any cycle of processes is non-zero. Thus it is not suitable for zero-delay abstract behavioral models.

```
initialize all logical processes    — this should cause some events to be scheduled
while event queue not empty loop
    set the global simulation time to the time-stamp of the first event in the queue
    repeat
        remove the first event in the queue and resume destination process
    until the time-stamp of the first event in the queue is greater than the current time
    wait until all processes have suspended
end loop
```

Figure 3. Centralized queue PDES algorithm.

be spread more evenly through simulation time. Hence the processes which receive these events will not be executed in parallel, even if the events are independent. Another problem is that the central queue is a shared resource between the processes executing during one simulation cycle, and is thus a potential bottleneck.

There are a number of possible ways to increase parallelism using a centralized event queue algorithm. One simple technique is to avoid serializing access to the central queue, so that it does not become a bottleneck when new events are added. An example of how this may be done is by dividing the queue into a number of subqueues, and placing events destined for different processes on different queues, using the destination process identification as a hash key. More elaborate techniques involve searching for independent events beyond the current time step. This search involves some model specific knowledge about the behavior of processes.

2.2 Conservative Distributed Algorithms

In the conservative distributed PDES technique, information about scheduled events is distributed amongst the receiving processes. Different processes may respond in parallel to events occurring at different simulation times, but a process only receives an event message when it can determine that no causality error will result. This is the meaning of the name ‘conservative’ for this technique.

Each logical process in the simulation maintains its own local clock recording the simulation time it has reached. Events are sent between logical processes via links, which act as FIFO queues. The links are created at simulation startup, connecting communicating logical processes. The conservative algorithm must ensure that a process does not receive an event with time-stamp earlier than the current local clock. This can be ensured in part by having processes send event messages in non-decreasing time-stamp order. In addition, each link has a clock associated with it, recording the time-stamp of the earliest message queued in the link. If the link is empty , the link clock records the time-stamp of the last message sent over the link. Each process repeatedly selects a link with earliest clock, and processes the first event message from the link. If all of the links with earliest link clock are empty, the process block until a message arrives on one of them. The algorithm executed by each process is shown in Figure 4.

Unfortunately this algorithm may lead to deadlock of part or all of the simulation. If, as in Figure 5, a set of logical processes are connected by a cycle of empty links and those links are the ones which are selected by each process in the above algorithm, then all processes in the cycle will be blocked permanently. Techniques for

```
initialize all logical processes    — this should cause some events to be scheduled
while event queue not empty loop
    set the global simulation time to the time-stamp of the first event in the queue
    remove the first event in the queue and resume the destination process
    enqueue any new events scheduled by the process
end loop
```

Figure 1. Discrete event simulation algorithms used in a sequential simulator.

There are three major classes of parallel discrete event simulation (PDES) algorithms that have been published in the literature: central queue PDES, conservative distributed PDES, and optimistic distributed PDES. Each of these has different approaches to dealing with parallelism, synchronization and communication of data within a model. Figure 2 shows a taxonomy of these algorithms.

2.1 Centralized Queue PDES

One simple way to achieve parallelism in executing a simulation model is to modify the sequential algorithm shown above, to process all events selected in one time-step in parallel. Since these events all have the same time-stamp, the causality constraint is not violated. The basic simulation algorithm, shown in Figure 3, is implemented by a kernel process, which maintains the central queue of event messages. Logical processes are implemented as parallel software processes. They execute a loop that receives an event message from the kernel, processes the message, then suspends. A process schedules an event message for another process by calling back to the kernel process, which updates the event queue.

The amount of parallelism available using this method is unclear, and may vary significantly between models in different applications. If many events occur with exactly the same time-stamp, this method may give us significant speedup. However, in real models which include device propagation delays, we expect that events will

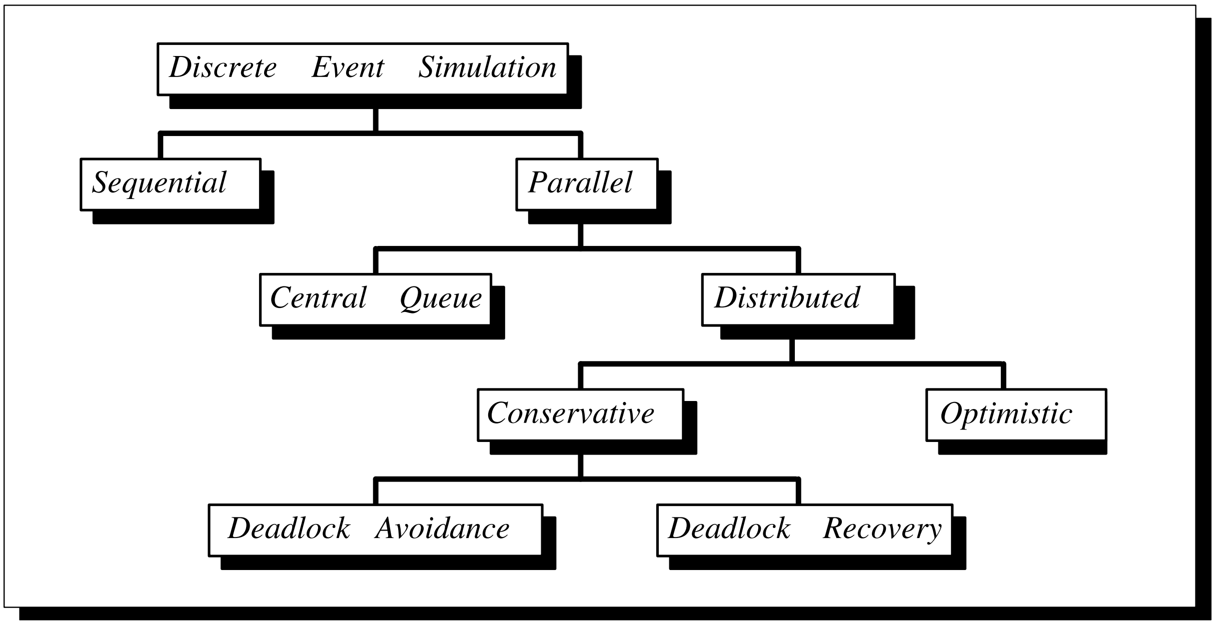

Figure 2. A taxonomy of discrete event simulation algorithms.

mining which techniques give best speedup for different kinds of VHDL models. It is expected that VHDL simulation will be amenable to speedup using PDES algorithms, but the degree of parallelism implicit in real VHDL models is currently not clear, nor are the patterns of data dependencies relevant to this problem well understood.

As a first step, this paper identifies a number of PDES algorithms used for discrete event simulation in general, and evaluates their applicability to executing VHDL simulation models. In Section 2, we survey a number of PDES algorithms, then in Section 3 we discuss the characteristics of VHDL that are relevant when considering parallel simulation, and consider the applicability of the various PDES algorithms to execution of VHDL models. In Section 4, we describe an initial experiment performed using one of the PDES algorithms and present measurements made during execution of some benchmark models.

2 SURVEY OF PDES ALGORITHMS

Discrete event simulation algorithms view a physical system as a collection of interacting physical processes. A simulation model consists of a number of logical processes, each corresponding to a physical process, which communicate by exchanging time-stamped event messages. Each logical process contains local state, which includes a local clock to measure the passage of simulation time. A logical process responds to an event message when its local clock reaches the time-stamp of the message. It then simulates the action of the corresponding physical process by modifying its internal state using the information in the message, and by sending further messages with later time-stamps to other logical processes.

Correct execution of a simulation model can be guaranteed if the following causality constraint is satisfied: that a process receives and acts upon events in non-decreasing time-stamp order. Since a process may modify its local state when it acts upon an event, observing this constraint guarantees that the process' response to a later event cannot affect the response to an earlier event, as we would expect.

In some cases, this constraint is stricter than necessary to achieve correct execution. We say two events are independent if processing one has no effect on the outcome of processing the other. Independent events can be processed in any order, irrespective of their time-stamps.

A typical sequential implementation of discrete event simulation uses a time ordered central queue of events remaining to be simulated at later simulation times. The logical processes are implemented as coroutines, being resumed in turn as the simulation progresses. Conceptually, the local clocks of all the logical processes are subsumed by a global simulation time clock. Simulation proceeds by the following algorithm shown in Figure 1. The body of the loop implements what is called a simulation cycle. In each cycle, an event at a single instant in simulation time is processed. Since the queue is ordered by the time-stamp of the events, the events are processed in non-decreasing time-stamp order, and thus the causality constraint is satisfied. Note that the order of processing of events scheduled for the same simulation time is not specified, nor need it be. Since such events correspond to simultaneous events in the physical system being modeled, there is no inherent ordering.

1 INTRODUCTION

Over the course of the last decade, VLSI circuits being manufactured have escalated in complexity. This is a direct result of advances in manufacturing technology. Simulation is now a vital tool used by VLSI engineers, both for verifying that a design meets its functional requirements, and for generating test vectors against which behavior of a manufactured circuit can be compared.

In order to simulate a design, the structure and intended behavior must be described. Structure can be described using circuit schematics, but behavior must be described using a hardware description language (HDL). An HDL is essentially a specialized programming language, designed for this purpose. In 1987, the IEEE standardized a hardware description language for digital circuits called VHDL [8]. Since then, VHDL has gained widespread use internationally, with simulators being made available by the main vendors of computer aided engineering software. Its adoption has been accelerated by a requirement by the United States Department of Defence that circuits designed for it under contract be delivered with documentation in VHDL. Independently, many engineers world-wide have seen the need for a common design language to allow them to interchange design information, and to use design tools with standard interfaces.

The current problem in VLSI engineering is that simulation runs take excessive amounts of time to execute. Run times of several days are not uncommon for complex circuits, causing increased design costs and longer time-to-market. The trend is for run times to lengthen, since the combinatorial complexity of simulation is outstripping performance improvements in computers. The delays induced may make a circuit uncompetitive when it is finally manufactured. The temptation is to reduce the amount of simulation done, at the risk of releasing a circuit with functional errors or having to re-iterate through the prototyping cycle. The underlying cause of the problem is that most of the simulation tools available commercially perform sequential execution of a model. However, VHDL is not inherently a sequential language; indeed, it has been designed to allow parallel execution with deterministic results.

The Vsim Project at the University of Adelaide is investigating means of speeding up simulation by executing VHDL models in parallel on multiprocessors and multicomputers. This can be achieved by processing a circuit's responses to numerous signal events in parallel. Parallel simulation is not a trivial problem, as the components of a model executing on different processors must be kept in synchronization to avoid causality errors. The success of this approach depends on the degree of parallelism allowable within a simulation model, which in turn depends on the data dependencies within that model.

There is a large body of research dealing with general techniques for managing parallel discrete event simulation (PDES). See [6] for a survey of the field. These techniques model a system as a collection of logical processes, which interact by scheduling and reacting to events that occur at instants in simulation time. This framework is appropriate for simulating VHDL models, since such models consist of processes that represent the behavior of circuit components, and that schedule and react to changes of values of circuit signals.

The experiments being performed in the Vsim Project involve developing VHDL simulators using an appropriate selection of the published techniques, and deter-

CONTENTS

1 Introduction 1

2 Survey of PDES algorithms 2
- 2.1 Centralized Queue PDES 3
- 2.2 Conservative Distributed Algorithms 4
- 2.3 Optimistic Distributed Algorithms 6

3 Characteristics of VHDL models 7
- 3.1 Processes and Signals in VHDL Models 8
- 3.2 Simulation Time and Delta Delays 10
- 3.3 Transport and Inertial Delay Semantics 14
- 3.4 Shared Variables 15

4 A central queue PDES implementation 16
- 4.1 Initial Implementation 16
- 4.2 Results 17
- 4.3 Modified Implementation 19

5 Conclusions 20

References 22

Parallel Execution of VHDL Models

Peter J. Ashenden, Henry Detmold and Wayne S. M^{c}Keen
{petera,henry,wayne}@cs.adelaide.edu.au

Department of Computer Science,
University of Adelaide,
GPO Box 498, Adelaide SA 5001,
Australia

Technical Report No. 93–01

January 1993

ABSTRACT

In this paper, we discuss the use of parallel discrete event simulation (PDES) algorithms for execution of hardware models written in VHDL. We survey central event queue, and conservative and optimistic distributed PDES algorithms, and discuss aspects of the semantics of VHDL and VHDL-92 that affect the use of these algorithms in a VHDL simulator. Next, we describe an experiment performed as part of the Vsim Project at the University of Adelaide, in which a simulation kernel using the central event queue algorithms was developed. We present measurements taken from this kernel simulating some benchmark models. It appears that this technique, which is relatively simple to implement, is suitable for use on small scale multiprocessors (such as current desktop multiprocessor workstations), simulating behavioral and register transfer level models. However, the degree of useful parallelism achievable on gate level models with this technique appears to be limited.