PERFORMANCE BOUNDS FOR THE CONSERVATIVE PARALLEL DISCRETE EVENT SIMULATION OF VLSI CIRCUITS AND SYSTEMS

MARK RAWLING, RHYS FRANCIS, DAVID ABRAMSON
High Performance Computation Project
CSIRO Division Of Information Technology
723 Swanston Street, Carlton, Vic. 3053, Australia

ABSTRACT

Parallel Discrete Event Simulation (PDES) is gaining popularity as a means of overcoming the design bottleneck traditionally suffered by commercial VLSI chip/circuit designers. There is much debate, however, as to the potential speedups available in such simulations and the best ways of achieving these speedups.

The two main approaches to PDES are the so-called conservative and optimistic strategies, of which the latter is generally thought to yield better potential speedups. However, optimistic simulation has potentially costly rollback requirements and in any event must be restricted to run conservatively in a non-functional environment such as that considered in this study.

The deterministic nature of conservative simulation makes it possible to conduct an accurate trace-driven analysis of an existing sequential simulator in order to establish upper bounds on a conservative PDES implementation. This paper describes such an analysis carried out on a commercial VLSI digital circuit simulator. The results add to the current knowledge of the potential concurrency characteristics of such simulators and have been used to assess the viability of conservative PDES in this field.

1. Introduction

Discrete Event Simulation (DES) is used in a wide variety of disciplines. It typically involves the simulation of complex systems made up of many interconnected modules. The changing state of a system over (discrete) time is modelled through state changes within modules as they are executed, and the transmission of timestamped events between modules. Events are timestamped since the simulation does not occur in real time.

The task of a simulator is to manage the execution of modules, as well as the routing and scheduling of events, in a way that produces results consistent with the corresponding system running in real time. A traditional (sequential) DES simulator achieves this by maintaining a time ordered *event queue* through which all event scheduling takes place. The basic algorithm is as follows:

```
while event queue is not empty do
    set the current time to that of the first event in the queue
    remove all events at the head of the queue with that time
    route the events to any sensing modules and execute those modules
    queue any new events according to their scheduled times
end-while
```

Although it is not mandated by the above algorithm, a simple DES simulator will execute all modules active at the current time sequentially. However, there is apparently no reason why independent modules can't be executed in parallel. This is the motivation behind Parallel Discrete Event Simulation (PDES), i.e., to detect and exploit the parallel execution of independent modules.

The performance of PDES is limited by the available concurrency in the system being simulated, and the presence of overheads greater than those of simple DES. One way to measure PDES

performance would be to adopt a 'look-and-see' approach, whereby a simulator could be designed for, or adapted to, a particular problem and the performance of that simulator then observed. This approach has the advantage of yielding precise results, but only for the particular implementations tested. In particular, this technique does not reveal the theoretically optimum performance of any given PDES model, and there are several. Furthermore, if one does not have ready access to the type of parallel machine that a successful implementation might require, or is not able to invest the time and effort into implementing a PDES simulator, then it would be essential to establish the likely performance in advance.

The next section looks at the basics of PDES. Remaining sections look more closely at the Funsim model and describe several levels of analysis designed to determine if Funsim can be sped-up significantly by applying the techniques of PDES. The analysis is illustrated using a simulation of a dynamic RAM controller in a microprocessor subsystem, 'Mullet'. This work is the result of a study carried out by the High Performance Computation Project of CSIRO's Division of Information Technology for an Australian VLSI chip/system design house, Austek Microsystems. The aim of the study was to investigate the application of parallel discrete event simulation as a means of speeding up Austek's digital circuit simulator, Funsim.

2. Parallel Discrete Event Simulation

Parallel discrete event simulation is gaining popularity as a means of improving performance. There is much debate however, as to the potential speedups available in such simulators, and the best ways of achieving these speedups. This section looks at three approaches to implementing PDES. The first method is a simple adaptation of sequential DES, while the second and third methods allow for fully asynchronous module execution.

2.1. The Centralised Event-Queue Approach

The simplest approach to PDES is suggested by the sequential algorithm given in section 1.. The centralised event queue and simulation kernel of sequential DES are retained, but at each time step all modules that receive an event are dispatched in parallel:

while *event queue is not empty* **do**
    set the current time to that of the first event in the queue
    remove all events at the head of the queue with that time
    in-parallel *route the events to any sensing modules and execute those modules*
    queue any new events according to their scheduled times
end-while

This method has the advantage that an existing sequential simulator could be adapted to it fairly readily. In the case of Funsim, it would involve relatively minor changes to the simulator kernel. It would also be necessary to ensure that the Funsim modules involved in such a simulation are sufficiently independent to allow this sort of parallel scheduling (see section 4.2.).

2.2. The Conservative Distributed Approach

The simple PDES scheme is restricted by a scheduling mechanism that limits parallelism to that which can be obtained by executing all modules in the *current* time step in parallel. A more promising approach is to remove the need to synchronise module execution at each time step, thus allowing modules from *different* time steps to be run in parallel. This leads to the so-called distributed PDES techniques.

One approach to distributed PDES is to manage module execution conservatively, i.e., modules can only execute when it is 'safe' to do so [2]. Each module maintains its own local clock, which is the

time to which it has executed. In this context, safety means that a module must not receive an event that would change its state at a time *prior* to its current clock. This effectively precludes the arrival of events in non-timestamp order. Events are therefore sent between modules via channels which act as FIFO queues, fan-in must be handled specially. Each channel also has a clock associated with it, which is equal to either the timestamp of the first event in the channel, or the time of the last processed event (plus any *lookahead*—see below), should the channel be empty. Channel clocks can be updated by different simulator components, including: the sending module, when it posts events; the receiving module, as events arrive; the channel itself, if it exists as an entity in its own right; or by an entirely asynchronous network manager (which could even be based around a centralised event queue). Modules execute based on the arrival of new events, the times of the channel clocks, and the time of the local clock:

```
(for each module)
while the system is active do
    wait until all input channel clocks are in advance of the module clock
    remove the first event (if any) from the earliest input channel
    update the module clock to that of the selected channel
    if the input has changed then
        reevaluate module state based on old state and current inputs
    end-if
    update output channels
end-while
```

Under certain circumstances, a conservative simulation may *deadlock*. This can happen whenever two or more mutually dependent modules, in a cyclic arrangement, are blocked while waiting on events from each other. Deadlocks can be avoided, however, if such cycles have a non-zero time delay. In this case, it is necessary and sufficient to keep outputs constantly up to date, both in time and value, hence the updating of output channels is not conditional in the above algorithm. Channel clock updates can be achieved by posting *null* events on outputs that do not change upon module reevaluation, although it is generally agreed that excessive use of null events can have a negative impact on performance because of the extra message traffic involved. An alternative scheme is for a blocked module to request of its predecessors the most up to date times of their connected outputs. Direct access to the channel clocks of these outputs would also be possible by maintaining this information in shared memory, thus reducing the need for null messages. Other conservative approaches allow deadlocks to occur; a deadlock recovery scheme is then employed to allow the simulation to continue. This avoids null messages but requires a centralised deadlock detection scheme. More detailed discussions of these concepts can be found in [2, 6, 4, 3, 9].

In a simple DES simulator, deadlocks cannot arise because the event at the head of the queue is guaranteed to be the earliest of all currently unprocessed events in the system. Similarly, optimistic PDES simulators do not deadlock (see next section); conservative PDES deadlock avoidance and recovery schemes therefore represent an overhead unique to this approach. In addition to causing potential deadlocks, the blocking characteristics of the conservative algorithm can also restrict the parallelism available at run time.

One way to increase parallelism is to keep the channel clocks further advanced in time than what is required for correct simulation and/or deadlock avoidance. This is done by exploiting a fundamental module property called *lookahead*. For example, in an 'inverter' module of a digital circuit simulator, it might be assumed that the output cannot change twice within the propagation delay of the inverter.[1] Output events can therefore be sent with a lookahead equal to the inverter's minimum propagation delay attached to them, or an extra null message, advanced by the lookahead, could be generated. Sensing modules may now know the state of their inputs further into the future than would otherwise be possible. If lookahead is attached to general events, rather than being sent

[1] Such a change would represent a glitch, worthy of special consideration in its own right.

as special null events, then it is possible for modules to execute more than once upon receiving a single event (as the event's lookahead could overlap other channel clocks). The algorithm shown will handle this situation correctly, but it requires modules to repeatedly scan their inputs, until the local clock cannot be advanced, rather than simply wait for an event to be present on every input and process the earliest one.

There are many variations on the basic conservative algorithm, most of which are designed to exploit the characteristics of the machine that the simulator runs on, or of the particular model being simulated. The shared memory and attached lookahead techniques for avoiding null messages are typical optimisations, others can be found in [6].

2.3. The Optimistic Distributed Approach

Any simulation in which modules may be executed before it is known to be safe to do so is said to be *optimistic*. Of particular interest to this study is *Virtual Time* or *Time Warp* simulation because of its close relationship to the conservative method described above [8]. Time Warp has become a very important and popular technique because of its potential to extract more parallelism from a given simulation model than either of the two previous approaches. In a Time Warp simulator there are no scheduling constraints, so module execution is entirely asynchronous. Events are processed in the order they are received, on the assumption that all other inputs are up to date with respect to the current event. As long as this assumption holds, the simulation will progress correctly, however, when the assumption breaks down, appropriate error recovery action must be taken.

Time Warp simulation has many interesting properties that other methods do not have. Perhaps the most bizarre of these is the ability of Time Warp to actually exceed the theoretical critical path performance of a given simulation [7]. This can happen because a module might be executed at an unsafe time, i.e., too early, and yet still produce correct results, which would not have to be undone on roll back. On the other hand, the very nature of conservative simulation means that it can never exceed critical path performance. Unfortunately, Time Warp cannot help in the case of modules that perform actions that can't be rolled back; such modules *must* be scheduled conservatively. This is particularly limiting in the case of simulations involving large amounts of I/O; many Funsim examples fall into this category.

3. Funsim

Austek Microsystems is an Adelaide based commercial VLSI chip/system design house. The company maintains an extensive suite of in-house CAD software, part of which is the digital circuit simulation package, Funsim.

Funsim is both a compilation system and a language for producing design-specific discrete event simulators from cell-based circuit layout and implementation description files. It is currently based on a sequential simulation kernel, with runs lasting anywhere from seconds for trivial circuits with short test vectors, to many hours for more complex simulations.

Funsim models circuits as collections of hierarchically connected cells, each having its own simulation model or being defined in terms of sub-cells, or both (allowing cells to verify the collective behaviour of their children). The compilation driver builds a design-specific simulator by compiling individual cell descriptions into reusable modules which are then linked with a sequential DES kernel and a run-time library. The actual cell connectivity can be compiled into the final simulator or can be read in from a definition file at run time. This approach allows a simulator to be used for many different arrangements of the same set of basic circuit cell definitions.

Run-time control of simulations is achieved either interactively or through command script files; in either case the resulting simulations are quantised in a way that limits their potential concurrency to that which can be obtained between control statements that interrupt the simulation kernel, e.g., breakpoints and time-limits. I/O plays an important role in Funsim—it is used extensively for

reading and writing simulation control and trace vectors, as well as interactive debugging, screen monitoring, etc.. This has a potentially large impact on the performance of a parallel Funsim implementation because of the extra synchronisation required.

Once configured, a particular simulation consists of *instances* of those circuit cells that have a simulation model, connected via *nodes* that handle fan-in and fan-out. Events represent signal transitions and carry information such as output drive strength and status (for high-impedance drives, non-zero transit times, etc.). A node's state is determined by combining the drives of all of its sources; inconsistent node source and sink states are checked extensively, for example, multiple sources, excessive loading, etc..

The sequential DES Funsim kernel maintains a single event queue and is responsible for the advancement of simulated time, (sequential) instance evaluation, and new event scheduling (by sorted queue insertion). The simple DES algorithm given in the introduction is modified to include the evaluation of all nodes that events occur on at the current time. Node status is currently evaluated directly by the kernel, as the routine involved is common for all nodes.

4. Experimental Setup

4.1. Aims and Related Research

The aim of this study is to establish upper-bounds on the potential parallelism of different PDES techniques when applied to real, representative simulations, in order to establish the viability of a PDES implementation of Funsim.

A relatively small number of studies into chip-level parallelism have been performed in the past, but the results remain largely inconclusive with regards to the ultimate value of PDES in VLSI logic simulation. It has been common practice to estimate parallelism based on event simultaneity [1, 13]. This approach is inappropriate for the Funsim case, however, since the bulk of the simulator's work load is due to instance evaluations, not event handling. Other related research has analysed the expected parallelism of special machine architectures with results not directly applicable to the more general purpose Funsim environment [5, 12]. A group at Stanford University looked at several PDES algorithms and measured real speedups based on a general-purpose parallel machine, but, like all the other studies mentioned, they made no attempt to establish upper-bounds on PDES performance [11].

4.2. Limitations and Scope

In addition to instance parallelism, a complete analysis would also take into account the overheads of the actual PDES scheme chosen; this would require either prior knowledge of these overheads, or a test to establish the overheads for the final system once implemented. Unfortunately, the former approach is impractical because sensitivity to the actual simulation environment makes past results largely inapplicable, while the latter solution is not possible in a feasibility study such as this. For these reasons, the results presented in this paper largely ignore the effects of scheduling overheads, but concentrate instead on *optimum* performance.

Although several levels of analysis are presented, the non-deterministic nature of optimistic scheduling makes it impossible to conduct an accurate trace-driven analysis of this technique when applied to Funsim. In addition, the conservative paradigm can be more readily applied to the existing sequential Funsim simulator than an optimistic one. For these reasons, and the requirement that non-functional modules *must* be scheduled conservatively, the scope of the analysis has been limited to establishing performance bounds on conservative PDES schemes only. Of course, the parallelism profiles and statistics produced are still relevant to the general characterisation of VLSI circuit simulation.

4.3. *Instrumentation*

The existing Funsim kernel was modified to log vital statistics when simulations were run in statistics gathering mode. The resulting 'log' files contain entries for each time step of the corresponding simulations. The time slots on the global event queue and the activity that results from processing the events in these slots are described, in part, by the following statistics:

Fundamental statistics

N_{sl}	The sequential number of this event queue time slot.
S_{sl}	The simulated time of this slot.
N_{ev}	The number of events in this slot.
N_{in}	The number of instances dispatched in this slot.
N_{nev}	The number of new events generated by instances dispatched in this slot.
N_{qev}	The total number of unprocessed events in the event queue, including those in this slot.
N_{qsl}	The total number of distinct time slots in the event queue, including this slot.
N_{sk}	The total number of future slots skipped over by events generated during this slot.
$T_{in(max)}$	The maximum number of real time ticks for any instance in this slot.
$T_{in(tot)}$	The total number of real-time ticks for all instances in this slot.

Derived statistics

$F_{ave} = N_{in}/N_{ev}$	The average fan out of events to instance dispatches in this slot.
$N_{ev(all)} = \sum_{i=1}^{N_{sl}} N_{ev_i}$	The total number of events processed so far, including this slot.
$N_{in(all)} = \sum_{i=1}^{N_{sl}} N_{in_i}$	The total number of instances dispatched so far, including this slot.
$T_{in(all)} = \sum_{i=1}^{N_{sl}} T_{in(tot)_i}$	The total number of real-time ticks for all instances so far, including this slot.

In addition to these statistics, which are lumped over whole time slots, a second, more comprehensive, set of data is also recorded. The resulting 'dependency' file contains enough information to carry out a full trace driven simulation of the corresponding Funsim simulator running in conservative PDES mode. This data is used in the 'Distributed Time Slot Analysis' and 'Timestamped Data-Flow Analysis' experiments described below.

Simulator activity during each time slot is described in a dependency file by four types of entries. A time entry gives the simulated time of its slot; all events in this slot were scheduled for this time. Event entries identify the node on which each event is scheduled. Node entries identify the instances dispatched by each node, if any. Finally, instance entries give, for each instance dispatched in this time slot: all new events scheduled by the instance, including their destination nodes and timestamps; and the instance execution time, in real-time 20μS 'ticks'.[2] Events are identified by an internally generated event number, and nodes and instances by their unique internal data structure virtual memory addresses.

The Funsim kernel ensures that a given instance will be dispatched at most once in any time slot, and only as a result of a node actually *changing* its state. Similarly, a given node cannot occur in more than one node entry in a time slot. It is possible for a dispatched instance not to generate any new events, usually, but not necessarily, as a result of outputs not changing. Also, not all nodes are necessarily sensed by any instances (unconnected outputs, etc.). Events of unknown origin are signalled by special entries, such events are typically generated by interactive simulator commands. Unknown events are given special treatment in the analysis that follows.

[2] All measurements were taken on a Sun Sparc Station 2 (40MHz clock) running SunOS 4.1. Instance real-times include both USER and SYSTEM time.

4.4. A Sample Simulation—Mullet

Mullet is an hierarchically structured simulation of a microprocessor/memory subsystem designed to test a proprietary DRAM controller chip. The system includes models of a 386 CPU, generic coprocessor interface, system interface (non-DRAM memory, I/O, interrupt acknowledge and DMA cycles), cache memory controller, cache memory SRAMs, DMA controller, main memory DRAMs and assorted control logic. The DRAM controller chip has 3092 instances in its simulation model, most of which are at the CMOS inverter/NAND gate/flip-flop/etc. level.

Table 1 shows the component/cell instances involved at the top level of the Mullet simulation. As can be seen, several of these instances model complex components, but the simulation also includes numerous lower-level instances. This is a characteristic of the type of simulations involved in this field, i.e., there is little uniformity in the size and complexity of the simulation modules. In this case, only two of the top-level instances, the cache and DRAM controllers, have nested simulation models, the numbers of leaf instances involved are also shown in the table.

Table 1: Top-level Component/Cell Instances in Mullet

Component/Cell	Number	Sub-cells	Leaf Instances
CPU (*i80386*)	1	0	1
Coprocessor interface	1	0	1
System interface	1	0	1
Cache controller	1	5	205
DRAM controller	1	5	3092
SRAM (8kx8)	8	0	8
DRAM (256kx8)	16	0	16
Miscellaneous	17	0	17

The Mullet simulation goes through several stages: an initial reset and 50nS run; next, the three system clocks—a 70nS oscillator for DMA refresh, a 25nS system control clock, and a 120nS DMA clock—are enabled, followed by a 300nS run to reset and initialise the CPU; and finally, a 200μS period of random cycle generation and simulation to exercise the entire system.

5. Results

The results are presented graphically where there is a significant element of variability over a run, and as global statistics where appropriate. We begin by looking at the sequential DES characteristics of Mullet and move on to more sophisticated levels of analysis in order to establish realistic upper bounds on parallelism.

5.1. Sequential DES Behaviour

Figures 1 to 3 illustrate the fundamental event queue and instance evaluation activity during the first 20,000 time steps of Mullet (a simulated time of 2977.77nS). Two distinct phases of the simulation are clearly visible: an initialisation phase which features interactive control of the simulation (through a command script file); and a repetitive random cycling phase which is representative of the long hands-off mode of processing typical in large simulations. Note that the trace has been truncated from 200μS to make each phase approximately equal in time; the second phase would normally continue with very little variation from the short profiles shown.

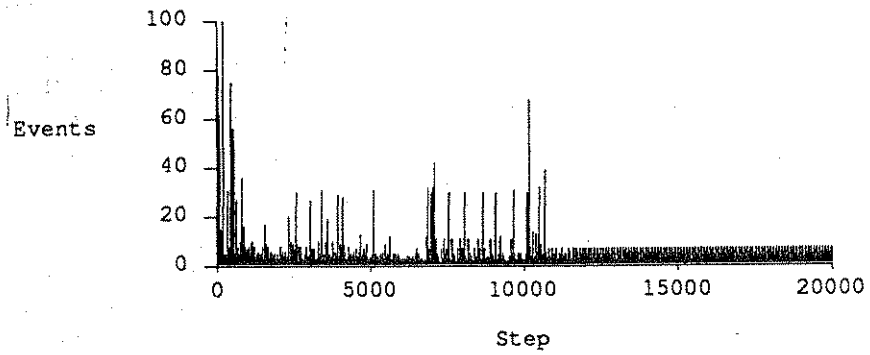


Figure 1: Events processed in each time slot (N_{ev} vs N_{sl}),

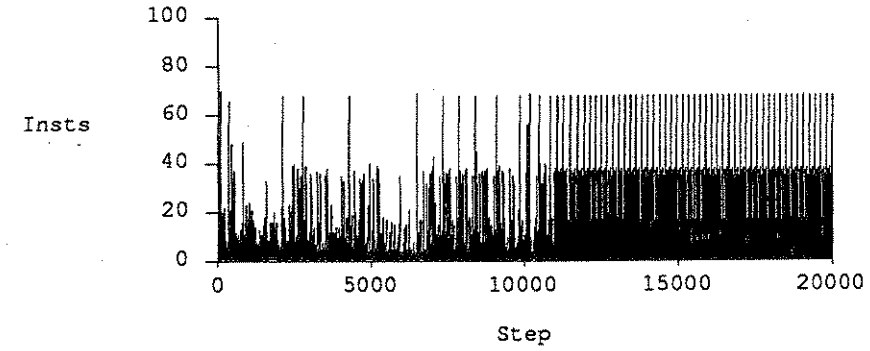


Figure 2: Instances dispatched in each time slot (N_{in} vs N_{sl})

5.2. *Simple PDES Analysis*

The number of instances dispatched during each time slot (Figure 2) is a direct measure of the potential concurrency of the simple centralised event queue PDES approach. Not all instances in a slot take the same time to execute, therefore, this graph should be interpreted as the *maximum* concurrency in each slot. If all instances were naively allocated to separate processing elements, then this graph would also show the number of processing elements needed to execute the simulation. In practice, either the number of PEs would be limited, with resulting machine saturation and stretching of the profile shown; or an alternative scheduling strategy, such as allocation of instances to currently idle PEs, could be used; in either case, some speedup may be lost.

Figure 4 gives the best over-all measure of potential simple PDES performance. The measure used is the ***weighted instance concurrency***, $W_{in} = T_{in(max)}/T_{in(tot)}$, which is the 'average' number of active instances in each time slot. This is plotted against the real time measure $T_{par(all)} = \sum_{i=1}^{N_{sl}} T_{in(max)}$. It would be reasonable to expect this performance in practice on a MIMD machine with enough processing elements to utilise the available concurrency, discounting the overheads of dispatching and synchronising instance evaluations. The area under this graph is a direct measure of the total processing done by the simulator—profiles with this property, but from different analyses, can be compared directly.

The average concurrency is remarkably low, even given the naive nature of the simple PDES algorithm. The highly irregular nature of the first phase is reflected by very low weighted concurrency, rarely exceeding three or four parallel instance dispatches, and often as low as one. This result is consistent with other real simulations studied and all but eliminates this technique as a viable

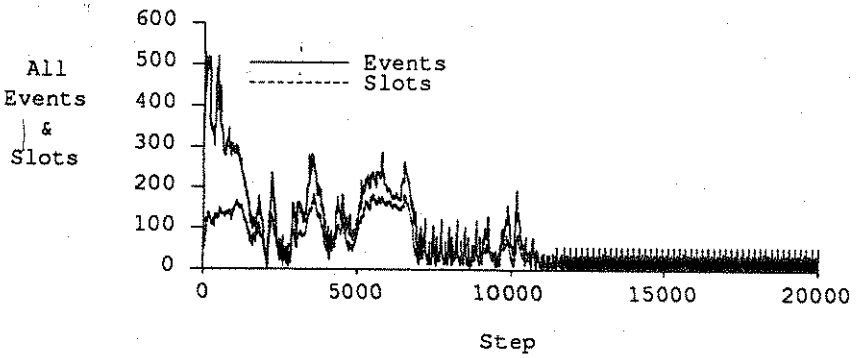


Figure 3: Event queue characteristics (N_{qev} & N_{qsl} vs N_{sl})

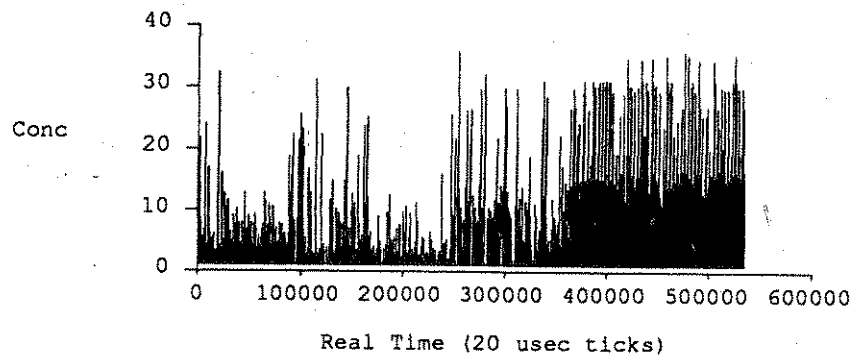


Figure 4: Weighted instance concurrency for simple PDES (W_{in} vs $T_{par(all)}$)

option for parallelising Funsim [10].

5.3. Distributed Time Slot Analysis

As a first step towards fully asynchronous PDES, the distributed time slot analysis considers the effect of advancing the scheduling of instances to an earlier slot than they are actually scheduled for. This is possible if it can be determined that no events generated between the new time and the originally scheduled time can affect this instance (as required by conservative PDES).

To study the effect of varying lookahead, a ***windowed*** analysis was devised.[3] This analysis consists of sweeping a fixed width window over the time slot based execution trace in the dependency file. Any instance visible in the current window that has no dependencies is counted as executable in that window. As the analysis proceeds it is possible that time slots become empty as their instances are moved forward into windows starting in an earlier slot. Windows are slid forward so that they do not start with an empty slot. As a result there are fewer windows, with more parallel instance dispatches in them, than there are time slots in the original trace.

Unknown events are given special consideration in the windowed analysis. If a time slot contains unknown events and is not the first active slot in that window (i.e., the first slot with any instances not already counted), then the window is immediately terminated. This process repeats for all windows that include this time slot, until this slot becomes the first active slot of a window, at this point window termination by this slot ceases and full windowing resumes. This technique prevents instances from being brought forward past a time slot with unknowns in it. This is necessary

[3] Windowing also limits the processing required to produce profiles.

because unknowns are actually interactive events, prior to which the simulation must have *paused* at a break-point, or time-limit (see section 3.). The windowed time slot analysis algorithm is as follows:

```
Build a (non-windowed) dependency tree:
    for all instances in the trace do
        locate dependent instances by 'following' events and nodes
        increment the dependency count on all dependent instances
    end-for

Analyse the dependency tree:
    for all time slots in the trace do
        skip this time slot if it has no active instances (slide the window forward)
        else start a new window beginning with this time slot and extending for the window size

        First Pass:
        for all time slots in this window do
            if this is not the first active slot in this window and it contains unknown events
            then terminate this window and go to second pass
            else mark any instances with zero dependency counts that haven't already
                 been counted in this or any earlier window as executable in this window
        end-for

        Second Pass:
        for all independent instances counted in this window do
            decrement the dependency count of all dependent instances
        end-for
    end-for
```

The analysis phase of the algorithm is performed in two passes. Dependency counts cannot be decremented in the first pass because child instances may then be counted as independent as the time slot is advanced in the current window. An associative hashing scheme based on instance identifiers is used to detect self-dependencies within a window, thus the same instance cannot be counted more than once in a window.

Figure 5 shows the windowed Mullet instance dispatch profile. The effect of the unknown events are visible as discontinuities at simulated times of 101nS and 351nS, in agreement with the commands in the Mullet test vector input file (not shown). The tails at the end of each phase in this profile consist of instances that fall on the critical path, and descendents of these instances.

The distributed time slot analysis predicts the performance of a simulation that uses lookahead to advance instance dispatches, but it is still based on a centralised event queue. This means that the critical path will be extended in real time because the time required for a window will be that of the *slowest* instance it contains, be it a 'critical instance' or not.

Figure 6 shows the weighted large windowed Mullet profile in real time (recall that the weighting expresses the average concurrency in each window). This is indeed the performance one would achieve with a real simulator (neglecting kernel overheads), but it is not yet an upper bound on conservative PDES. To establish a true upper bound it is necessary to remove the central event queue altogether—this requires abandoning any form of time slot based analysis and looking instead at *individual* instance evaluation times. This is achieved by the data-flow analysis covered next.

5.4. Timestamped Data-Flow Analysis

Distributed PDES can be achieved with message passing or its equivalent directly between modules. In the case of Funsim, instances could send each other event messages as they are generated; it

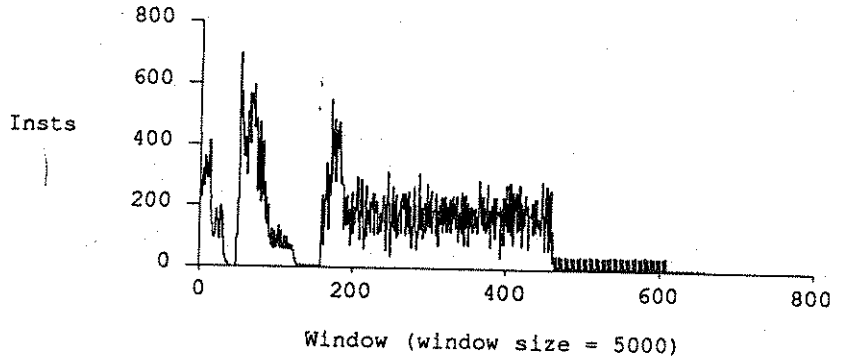


Figure 5: Distributed instance dispatches in Mullet (large window)

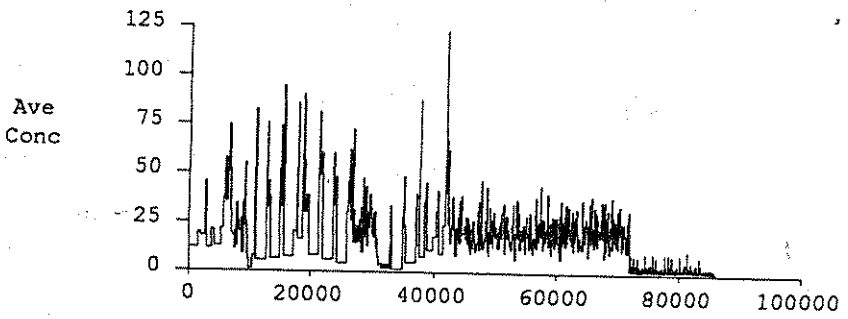


Figure 6: Weighted distributed instance dispatches in Mullet (large window, real time)

would be up to individual instances, or surrounding *shells*, to handle the evaluation of their input node states, as currently done by the central Funsim kernel. This approach bypasses any centralised control (and associated bottlenecks), apart from that needed to detect and avoid deadlocks, if necessary. The dependency traces include individual instance timings which can be used to simulate this protocol. Unknown events are again given special treatment, as their implications remain unchanged. The timestamped data-flow analysis algorithm is similar to the time slot algorithm except that the start and end times of individual instances are used, rather than the start and end times of time slots:

Build a timestamped data-flow dependency tree:

First Pass: link self-dependencies
for *all instances in the trace* **do**
place all occurrences of the same instance on a doubly-linked self-dependency list
end-for

Second Pass: determine earliest instance dispatch times
earliest_global_start = 0
latest_global_finish = 0
for *all time slots in the trace* **do**
if *this slot contains any unknown events*
then *earliest_global_start = latest_global_finish*
end-if

```
for all instances in this slot do
    Update self and next self earliest start times first:
    inst.earliest = max(earliest_global_start, inst.earliest)
    latest_global_finish = max(latest_global_finish, inst.earliest + inst.ticks)
    next_self.earliest = max(next_self.earliest, inst.earliest + inst.ticks)
    Now update all other descendants:
    for all descendant instances do
      descendent.earliest = max(descendent.earliest, inst.earliest + inst.ticks)
    end-for
  end-for
end-for
```

Generate instance dispatch list:

```
for all instances in the trace do
    insert on and off times into a sorted instance schedule list
end-for
```

Earliest instance start times begin at zero and are gradually pushed back as dependencies are detected; when an instance is finally scheduled its start time will be the latest finish time of any of its parents.[4] A data-flow parallel instance profile is generated by traversing the instance dispatch list and plotting the number of active instances against time. Each 'on' entry increments the active instance count and each 'off' entry decrements the count—elapsed time is equal to the real time component of each entry. Weighting is not required here since the profile is a plot of precisely how many instances are active against time. Figure 7 shows this profile for Mullet.

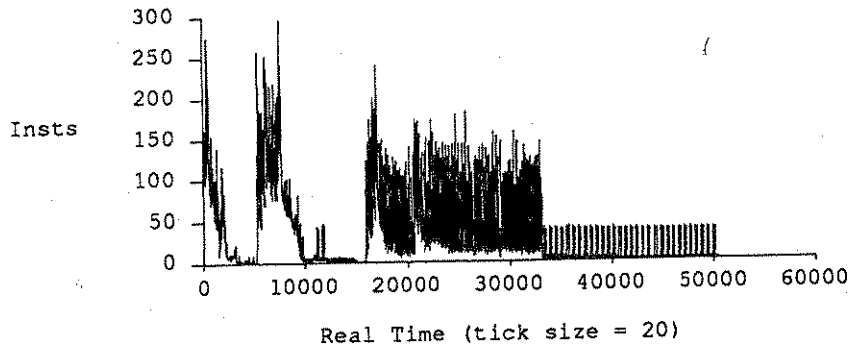


Figure 7: Data-flow instance concurrency for Mullet

Note the similarity between the data-flow profile and the large window distributed time slot profile (figure 5). In fact, the only difference between the two is the extra parallelism gained by removing the restriction that all instances in a window must start at the same time. The data-flow profile is in fact optimal for a given simulation since it is based on only *critical* run-time dependencies. Thus, it represents a true upper bound on the parallelism available to a conservative PDES simulation.

5.5. *Realistic Performance Expectations—Lookahead*

The performance profiles shown so far are optimal for the PDES models used to derive them, but the distributed time slot and data-flow results, in particular, give little indication as to the

[4] The data-flow analysis could be made *slightly* more accurate by removing the assumption that all events are posted at the *end* of instance evaluation (though this assumption turns out to be quite reliable).

real expected performance of actual implementations. Clearly, the parallelism obtainable by these methods depends on whether or not it can be determined that future instances are independent of any events that may be scheduled between the current time and the scheduled times of those instances. The extent to which this determination can be achieved at run time will depend mainly on lookahead, as defined in section 2.2..

The effect of lookahead is determined by varying the window size in the distributed analysis. Figure 8(a) shows speedup curves for execution times versus window size. Speedups are expressed relative to sequential DES execution times. Although the number of windows monotonically decreases as window size increases (not shown), the time taken can actually increase. This is because total execution time is the sum of the maximum instance times in each window, so that if several slow instances from a given time slot or window are separated by an increased window size (independent ones can be brought forward, whilst dependent ones cannot), then execution efficiency may be reduced and the total time increase. This effect can be observed in figure 8(a) for phase 2 at a window size of 192.

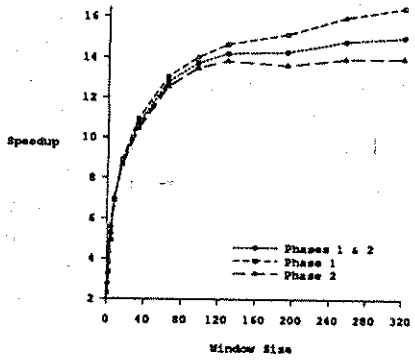
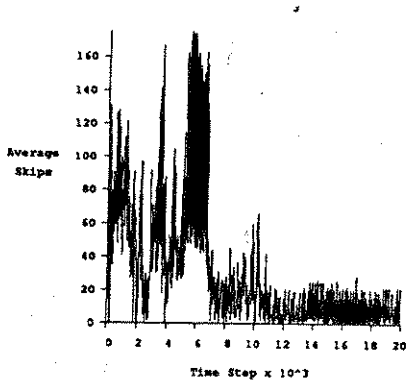


Figure 8: (a) Speedup over sequential DES



(b) Average event time slot skips (N_{sk}/N_{nev})

A quantitative lookahead estimate can be made by observing the ratio N_{sk}/N_{nev}, i.e., the average number of future time slots skipped by the new events created at each time step. This figure is plotted in figure 8(b) (slots with no new events are ignored). The graphs in figure 8 can be cross-referenced to estimate speedups at different stages of the simulation, this is done in the next section. Notice that in phase 2, the large-window speedup is lowest, whilst the average lookahead is also lowest; this is a particularly discouraging result as a 'real' Mullet run will spend nearly all of its time in an extended phase 2.

5.6. Run-Times and Speedups

Table 2 shows the run-times and speedups (over sequential DES) obtained from the different analyses. These figures assume no kernel overheads.

The 'Real Estimate' column is based on distributed time slot performance with window sizes derived from the average lookahead profile of figure 8(b). The lookahead estimates used were 30 for phase 1 and 10 for phase 2. These window sizes give the run-times and speedups shown in the table. A similar windowed lookahead analysis could also be applied to the data-flow profiles; we would expect very similar results to those obtained for the distributed time slot analysis if this was done.

The percentage of time spent in the kernel versus that spent in actual instance evaluation will affect the speedup available due to parallel instance dispatching. The actual overheads are unknown for a given parallel implementation except that the simple PDES scheme, and the distributed PDES scheme based on a central event queue, would be expected to have similar overheads to the existing

Table 2: Predicted simulation run-times/speedups for Mullet

Trace	DES	PDES	Distributed[1]	Data-Flow	Real Estimate
Mullet entire	29.41	10.67/2.8	1.72/17.1	1.01/29.2	3.3/8.9
Mullet phase 1	13.35	5.80/2.3	0.66/20.3	0.32/42.2	1.2/11.1
Mullet phase 2	16.05	4.86/3.3	1.06/15.1	0.69/23.2	2.1/7.6

[1] large window

sequential DES Funsim simulator, as well as any extra overheads due to synchronisation, etc.. Current Funsim kernel overheads have been observed as low as 0.4% and as high as 40% of sequential execution time. The actual figure for Mullet averages 12% during phase 2, which would degrade the estimated speedup for this phase from 7.6 to about 4.2. This figure still excludes any *extra* overheads introduced by the actual PDES implementation.

6. Conclusions

The parallelism results for Mullet and other representative Funsim simulations are generally discouraging in terms of PDES applied to VLSI simulation. Chips with many thousands of instances are routinely found to have maximum parallelisms in the tens rather than thousands or even hundreds. The data-flow profile for Mullet indicates an expected peak average parallelism of around 40 for the critical phase 2 section, but even this is effectively canceled by the long sequential tail on this profile; the average over-all best case parallelism in this region is 23.2. It was further shown that a realistic speedup estimate, taking into account the effects of limited lookahead opportunities and extra kernel overheads (for the distributed case), could be as low as 4 or less for the Mullet test simulation. This result (and other test cases, not discussed here) force us to conclude that there is no conservative scheduling policy that would generate truly significant speedups for Funsim simulators.

There is much interest in Time Warp simulation and it may well be suitable for the Funsim model, despite the fact that operations that can't be rolled back must be scheduled conservatively. Unfortunately an analysis of Time Warp's potential was outside the scope of this project (see section 4.2.), however, it would make an excellent subject for future research, as would an investigation into alternative ways of speeding up Funsim simulators through parallel computation, based on the peculiarities of the Funsim model itself.

7. Acknowledgements

We would like to thank the people at Austek Microsystems for their support of this project. In particular, David Johnstone and Mark Pulver, for providing accommodation at Technology Park and access to Austek's impressive VLSI CAD simulation package, Funsim.

The High Performance Computation Project is a joint project between the Commonwealth Scientific and Industrial Research Organisation (CSIRO) Division of Information Technology and the Royal Melbourne Institute of Technology (RMIT).

References

[1] M.L. Bailey and L. Snyder. An empirical study of on-chip parallelism. In *Proceedings of the 25th ACM/IEEE Design Automation Conference*, pages 160–165, 1988.

[2] K.M. Chandy and J. Misra. Asynchronous distributed simulation via a sequence of parallel communications. *Communications of the ACM*, 24(4):198–206, April 1981.

[3] B.A. Cota and R.G. Sargent. An algorithm for parallel discrete event simulation using common memory. In *Proceedings of the 22nd Annual Simulation Symposium*, pages 23–31, March 1989.

[4] N.J. Davis, D.L. Mannix, W.H. Shaw, and T.C. Hartrum. Distributed discrete-event simulation using null message algorithms on hypercube architectures. *Journal of Parallel and Distributed Computing*, 8:134–148, 1990.

[5] E.H. Frank. Exploiting parallelism in a switch-level simulation machine. In *Proceedings of the 23rd Design Automation Conference*, pages 20–26, 1986.

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

[7] D. Jefferson and P. Reiher. Supercritical speedup. In *Proceedings of the 24th Annual Simulation Symposium*, pages 159–168, 1991.

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

[9] P. Konas and P. Yew. Parallel discrete event simulation on shared-memory multiprocessors. In *Proceedings of the 24th Annual Simulation Symposium*, pages 134–148, April 1991.

[10] M. W. Rawling. Potential parallelism in vlsi circuit simulation. Technical Report TR-DB-91-10, CSIRO Division of Information Technology, 1991.

[11] L. Soule and T. Blank. Parallel logic simulation on general purposes machines. In *Proceedings of the 25th ACM/IEEE Design Automation Conference*, pages 166–171, 1988.

[12] K.F. Wong and M.A. Franklin. Parallel discrete event simulation. *Journal of Parallel and Distributed Computing*, 7(2):416–440, July 1989.

[13] K.F. Wong, M.A. Franklin, R.D. Chamberlain, and B.L. Shing. Statistics on logic simulation. In *Proceedings of the 23rd Design Automation Conference*, pages 13–19, 1986.

Fifteenth Australian Computer Science Conference

ACSC-15

29-31 January, 1992

Australian Computer Science Communications
Volume 14, Number 1
PART B

Department of Computer Science
University of Tasmania
GPO Box 252C
Hobart, Tasmania
AUSTRALIA 7001