

Potential Performance of Parallel Conservative 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, Victoria 3053, Australia

Abstract

The deterministic nature of conservative simulation makes it possible to conduct an accurate trace-driven analysis of an existing sequential simulator in order to predict concurrency profiles and speedup bounds for several parallel implementations. This paper describes such an analysis carried out on a commercial VLSI digital circuit simulator with results based on real instance evaluation times at microsecond resolutions. We consider a central event queue based model with and without lookahead, as well as the distributed message passing model.

The results help to answer the important questions, "Just how much concurrency is there in real commercial VLSI simulations?" and "What type of machine would be best suited for a parallel implementation?"

1 Introduction

The work described in this paper 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 conservative parallel discrete event simulation as a means of speeding up Austek's VLSI circuit simulator, Funsim. The deterministic nature of conservative simulation makes it possible to conduct an accurate trace-driven analysis of some representative Funsim simulations in order to predict concurrency profiles and establish upper bounds on a conservative parallel discrete event simulation (PDES) implementation. We are particularly interested in the study of real, representative, commercial simulations and, if there is concurrency in these simulations, then the type and scale of parallel machine that would be best suited to extracting it.

Funsim is both a compilation system and a lan-

guage for producing design-specific discrete event simulators from hierarchical, 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. Run-time control of simulations is achieved either interactively or through command script files. The simulation kernel supports breakpoints, time-limits and other asynchronous event sources. 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 fanin and fanout. Signal transitions on instance outputs are represented by *events* which carry information such as output drive strength and status (for high-impedance drives, non-zero transit times, etc.). A node's state is evaluated directly by the simulation kernel by combining the drives of all of its sources. Node sources and sinks are checked extensively for erroneous conditions, e.g., multiple sources, excessive loading, etc.. Nodes can be viewed as mini-instances with zero time delay which require an extra level of synchronisation at the kernel level (they are thus a potential source of rollback in an optimistic simulation model).

A relatively small number of studies into chip-level parallelism have been performed in the past and the results remain somewhat inconclusive with regards to the ultimate value of parallel discrete event simulation in VLSI logic simulation. It has been common practice to estimate parallelism based on event simultaneity [1, 17], but this approach is inappropriate for the Funsim case since the bulk of the simulator's work load is due to instance evaluations which are rather

decoupled from event arrivals due to Funsim's node evaluation strategies and optimisations. For example, under Funsim, many events may arrive at a node to be trivially processed (and discarded) before one arrives that causes any connected instances to be evaluated.

Other related research has analysed the expected parallelism of special machine architectures with results not directly applicable to the more general purpose Funsim environment [6, 16]. A group at Stanford University looked at several PDES algorithms and measured real speedups based on a general-purpose parallel machine, but in this case an optimal critical path performance analysis was not conducted [14, 15]. Fujimoto reported on the effects of lookahead in conservative PDES simulations in [7], but these results were largely specific to queueing network simulations on a BBN Butterfly multiprocessor. Reed *et al.* showed that even on a shared memory architecture, Chandy-Misra style PDES can suffer from some very alarming inefficiencies [13]. Finally, the idea of using a critical path data-flow analysis to establish an upper bound on conservative simulation performance is not new, e.g., [2], but our analysis is more complete with respect to the peculiarities of the Funsim model. In addition, we investigate more models than just the asynchronous Chandy-Misra algorithm [3].

An analysis of optimistic simulation was outside the scope of this study, but there is no doubt that Time Warp has become a very important and popular technique due to its potential to extract more parallelism from a given simulation model than conservative techniques [10]. 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 [9]. 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 instances that perform actions that can't be rolled back; such instances 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, although we are certainly not ruling out Time Warp as being potentially effective in this field.

2 Experimental Limitations and Scope

A complete analysis would take into account all of the overheads of the actual PDES scheme chosen. This would require either prior knowledge of these over-

heads, 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; the corresponding parallelism profiles are therefore 'optimal' but somewhat impractical. An effort is made to account for some of the effects of scheduling overheads in sections 5.4 and 5.5, and in the conclusion.

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. For this reason, and the requirement that non-functional instances 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.

3 Sample Simulations

Several sample simulations were obtained from Austek for the purposes of gathering statistics on Funsim. The three simulations studied in this paper are briefly described in this section. We will concentrate mainly on the Mullet and Mars simulations since they are real simulations of commercial importance.

3.1 Mesh

Mesh is a specially constructed simulation of an array of 10 x 12 cells, each with two data input lines (with acknowledge) and two data output lines (again, with acknowledge). The timing parameters of the cell array and the conditions for data transfer, i.e., the logic of the cells, are adjustable to allow the effects of various system parameters to be studied. In this study, the mesh is set-up as a pipelined, endless twisted ring. Signals circulate the ring twelve times before returning to their starting cells. This simulation is included because it is expected to produce high levels of concurrency.

3.2 Mullet

Mullet is an hierarchically structured Funsim 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 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.

Component/Cell	Number	Sub-cells	Leaf Insts
CPU (i80386)	1	0	1
Coprocessor iface	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

Table 1: Top-level component/cell instances in Mullet

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 200 μ S period of random cycle generation and simulation to exercise the entire system.

Mullet exhibits a very important characteristic of the type of simulations involved in this field, i.e., there is little uniformity in the size and complexity of the cell instances. 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.

3.3 Mars

Mars is a complex simulation of a four-cpu SPARC multiprocessor system. It is a purely *architectural* simulation as can be seen from table 2; i.e., the leaf cells in this simulation are virtually all high level components apart from some trivial interface 'glue'. Compare this with Mullet, which includes transistor level models for many of its components. The Mars system can be simulated in either uniprocessor or multiprocessor mode. We will observe the multiprocessor results here.

Component/Cell	Number	Sub-cells	Leaf Insts
Processors	4		
ATC	1	0	42
CPU	1		
Cntrl	1	0	12
SPARC	1	0	1
Misc	16	0	16
Cache	24	0	24
Bus Watcher	19	0	19
Misc	3	0	3
IO Server	1	1	99
Miscellaneous	28	0	124

Table 2: Top-level component/cell instances in Mars

4 Instrumentation

The existing Funsim kernel was modified to log vital simulation statistics. 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:

Time Slot Analysis 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_{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}	The average fan out of events to instance dispatches in this slot ($= N_{in}/N_{ev}$).
T_{seq}	Sequential real time — the sum of real-time ticks for all instances so far ($= \sum_{i=1}^{N_{sl}} T_{in(tot)_i}$).
T_{par}	Parallel real time — the sum of the maximum instance times for each slot ($= \sum_{i=1}^{N_{sl}} T_{in(max)_i}$).

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 analysis of the corresponding Funsim simulator running in different PDES modes. 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 the dependency file by four types of entries: time, event, node and instance. A time entry gives the simulated time of its slot. 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 20 μ S ticks.¹

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 due to unconnected outputs, manually desensed nodes, etc..

Events can be of unknown origin, these are typically generated by interactive simulator commands, internal clocks, etc.. They occur when the simulation pauses at a break-point or time-limit, or, more importantly, as a result of the way signal clocks are handled. Clock events are generated whenever global time reaches the scheduled time for an asynchronous clock to fire. This situation is detected between kernel iterations, not as part of the normal event queue driven activity. Unknown events are given special treatment in the following sections.

5 Results

5.1 Simple PDES Analysis

The simplest approach to conservative PDES is suggested by the sequential DES algorithm itself. The centralised event queue and simulation kernel are retained, but at each time step all instances that receive an event are simply dispatched in parallel (figure 1).

This method has the advantage that an existing sequential simulator can be adapted to it quite easily. 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 instances involved in

¹All measurements were taken on a Sun Sparc Station 2 (40MHz clock) running SunOS 4.1. Instance real-times include both USER and SYSTEM time.

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

Figure 1: Central event queue PDES algorithm

such a simulation are sufficiently independent to allow this sort of parallel scheduling, i.e., a suitable coding discipline would have to be imposed on circuit cell design.

The number of instances dispatched during each time slot for Mullet and Mars are shown in figure 2. These profiles are a direct measure of the potential concurrency of the simple PDES approach.

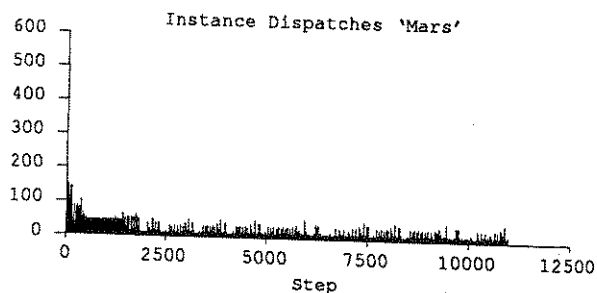
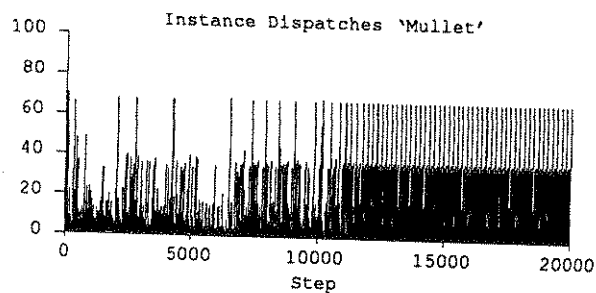


Figure 2: Instance dispatch profiles for Mullet and Mars (N_{in} vs N_{sl})

Not all instances in a slot take the same time to execute, therefore, these graphs should be interpreted as the maximum concurrency in each slot. If all instances were naively allocated to separate processing elements, then these graphs would also show the number of pro-

cessing 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; 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.

Only the first 20,000 time steps of Mullet are shown (a simulated time of 2978nS). Two distinct phases of the simulation are clearly visible: an initialisation phase, which features external 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. The trace has actually been truncated from 200 μ S to make each phase approximately equal in time; the second phase would normally continue with very little variation from the short profiles shown.

Figure 3 shows the Mars profile in sequential real time. There is significant distortion of the horizontal axis due to some unusually slow instances executed at the beginning of the simulation; this is a rare occurrence which was not observed in any other test studies. Note the two peaks at the beginning of the Mars profile; it is common for these simulations to exhibit a great deal of activity upon initialisation.

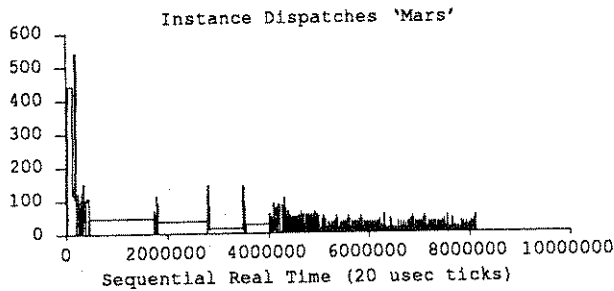


Figure 3: Instances dispatched in each time slot for Mars (real time) (N_{in} vs T_{seq})

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 parallel real time measure, T_{par} (section 4). 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 (which would be very low for this algorithm). The area under this graph is a direct measure of the total processing done by the simulator —

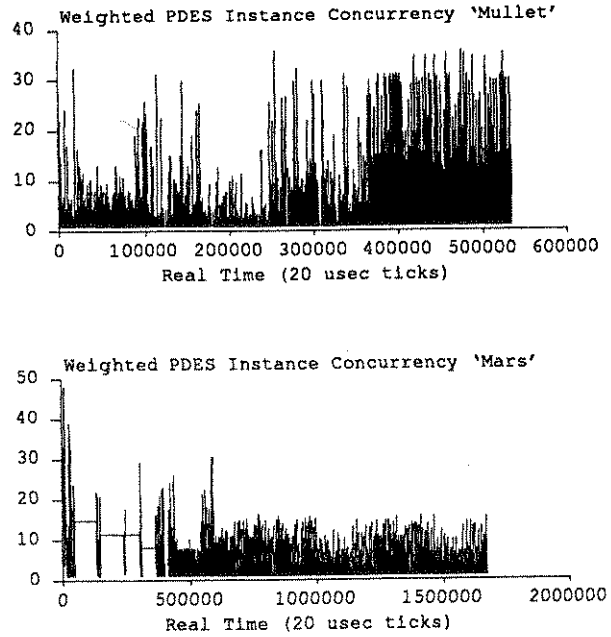


Figure 4: Weighted instance concurrencies for simple PDES (W_{in} vs T_{par})

profiles with this property, but from different analyses, can be compared most meaningfully.

The average concurrency is remarkably low, even given the naive nature of the simple PDES algorithm. The highly irregular nature of the first phase of Mullet 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 across all test simulations and all but eliminates this technique as a viable option for parallelising Funsim beyond the level of trivial and inconsistent speedups.

5.2 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 any conservative simulation algorithm).

This is done by exploiting a fundamental instance property called *lookahead*. For example, in an 'inverter' instance of a digital circuit simulator, it might be assumed that the output cannot change twice within

the propagation delay of the inverter.² 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 instances may now know the state of their inputs further into the future than would otherwise be possible, allowing for advanced execution.

To study the effect of varying lookahead and to limit the processing required to produce profiles, a *windowed time slot analysis* was devised (figure 5).

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 over any empty time slots and start a new window
  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
```

Figure 5: Windowed time slot analysis algorithm

The 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 (section 4) are given special con-

²Such a change would represent a glitch, worthy of special consideration in its own right.

sideration 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.

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 6 shows the resulting profiles for the Mullet and Mars simulations, while figure 7 shows the Mars profile in real time and again shows the distortion of the horizontal axis due to the slow instances at the beginning of the simulation (cf. figure 3).

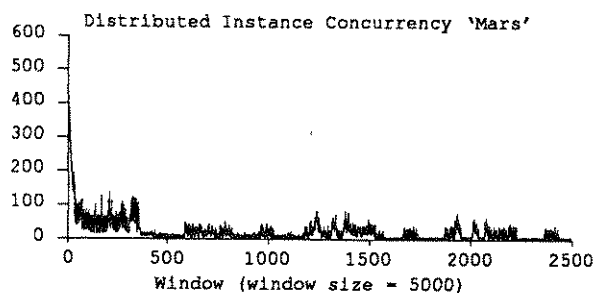
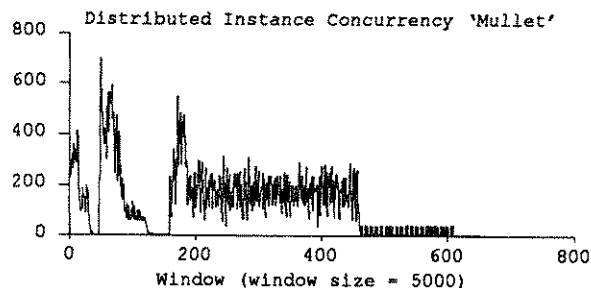


Figure 6: Distributed instance dispatches (large window)

In the Mullet profile, the effects of the unknown events are visible as discontinuities which actually occur at simulated times of 101nS and 351nS in agreement with the commands in the Mullet test vector

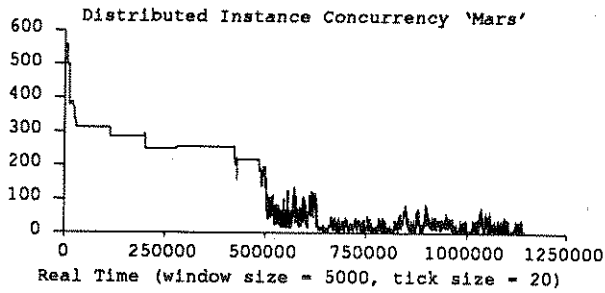


Figure 7: Distributed instance dispatches in Mars (large window, real time)

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.

Figure 8 shows the weighted large window profiles in real time (recall that the weighting expresses the average concurrency in each window). This is the performance one would achieve with a central event queue simulator with infinite lookahead, zero kernel overheads and perfect scheduling so that no time is lost due to a finite number of processing elements. Obviously, real implementation details will degrade these profiles, some of these effects are considered in sections 5.4 and 5.5, and in the conclusion.

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.

To establish a true upper bound on conservative PDES 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.

5.3 Timestamped Data-Flow Analysis

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

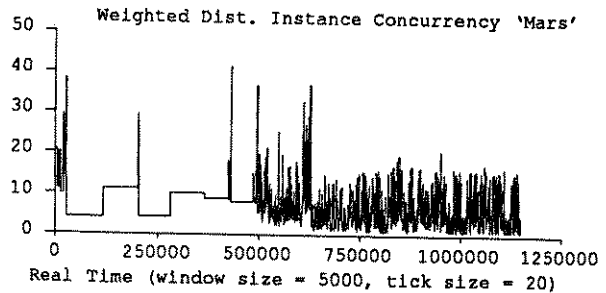
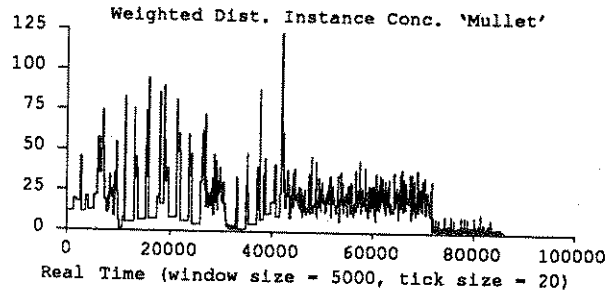


Figure 8: Weighted distributed instance concurrencies (large window, real time)

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 (figure 9).

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.³ 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 10 shows the final data-flow profiles for Mullet and Mars.

Note the similarity between the data-flow profiles

³ 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).

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
      descendant.earliest =
        max(descendant.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
```

Figure 9: Timestamped data-flow analysis algorithm

and the large window distributed time slot profiles (figures 6 and 8). 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.

5.4 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 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

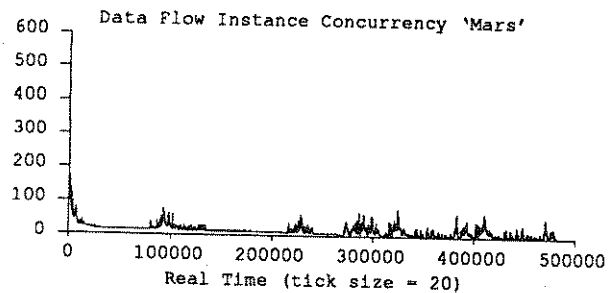
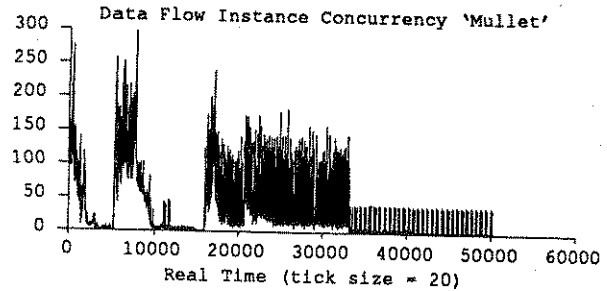


Figure 10: Data-flow instance concurrencies

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 described in section 5.2.

The effect of lookahead is determined by varying the window size in the distributed analysis. Figure 11(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, while dependent ones cannot), then execution efficiency may be reduced and the total time increased. This effect can be observed in figure 11(a) for phase 2 at a window size of 192.

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 11(b) (slots with no new events are ignored). The graphs in figure 11 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

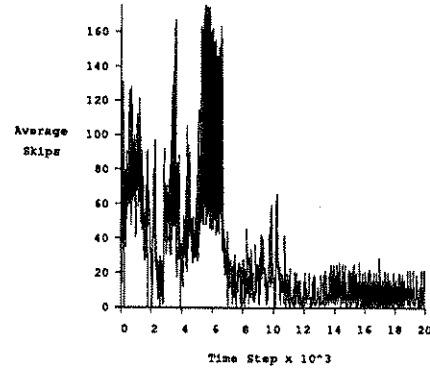
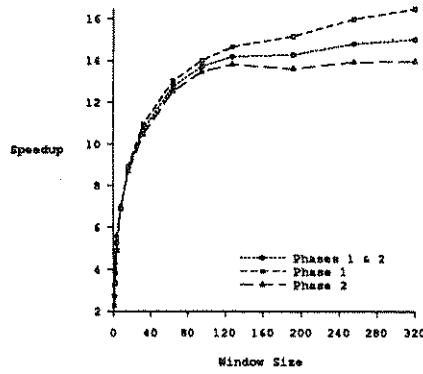


Figure 11: (a) Speedup over seq. DES (Mullet)

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

Trace	Sequential DES	Simple PDES	Distributed PDES [†]	Estimated	Data-Flow PDES
Mesh	5.91	2.71/2.2	1.49/4.0	1.73/3.4	0.11/54.8
Mullet entire	29.41	10.67/2.8	1.72/17.1	3.2/9.1	1.01/29.2
Mullet phase 1	13.35	5.80/2.3	0.66/20.3	1.1/11.7	0.32/42.2
Mullet phase 2	16.05	4.86/3.3	1.06/15.1	2.1/7.6	0.69/23.2
Mars	162.6	33.45/4.9	22.9/7.9	23.0/7.1	9.64/16.9

* figures represent time/speedup — all times in seconds, speedups relative to sequential DES times

[†] large window

Table 3: Predicted simulation run-times and speedups

speedup is lowest, while 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.

5.5 Run-Times and Speedups

Table 3 shows the run-times and speedups (over sequential DES) obtained from the different analyses. These figures assume no kernel overheads. The 'Estimated' column is based on distributed time slot performance with window sizes derived from the average lookahead profile of figure 11(b) for Mullet. For Mesh and Mars, the window sizes are estimated from special grouped time slot profiles reported elsewhere [12]. The grouped profiles plot instance dispatch groups vs time slot. A group includes all instances in the sequential dispatch profile up until the slot in which global simulated time exceeds that of the earliest event posted by any instance in that group. Thus, groups contain run-time independent instances regardless of where events are scheduled, the size of the groups representing how much look-ahead there is in the simulation. The grouped experiment results are not presented here since they do not correspond to any practical PDES implementation — the end of the current group

can not be determined at run time.

The lookahead estimates used were: Mesh 6; Mullet 40 for phase 1 and 10 for phase 2; and Mars 9. Time slot analyses with these window sizes give the run-times and speedups shown in the table. A similar lookahead analysis could be applied to the data-flow profiles; we would expect very similar results to those obtained for the 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 simple PDES scheme, and the distributed PDES scheme based on a central event queue, would be expected to have similar overheads to the existing sequential DES Funsim simulator, plus 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.

5.6 Fanout

Table 4 shows the maximum and average fanout from instance to instance for each simulation. An

Trace	Fanout (max)	Fanout (ave)
Mesh	8	1.06
Mullet entire	177	1.05
Mullet phase 1	177	1.12
Mullet phase 2	176	1.02
Mars	178	2.98

Table 4: Maximum and average dynamic fanouts

analysis of the individual simulations reveals that for Mullet and Mars, it is the main clock that has the maximum fanouts shown. Mesh is not clocked at all and simply has a maximum cell fanout of eight.

It is interesting to contrast the maximum to average fanouts of the large, clocked simulations. The large fanout variability indicates that these simulations are quite unbalanced, with large activity occurring with each clock and relatively little activity between clocks. In fact, the low average fanouts in these circuits go a long way in explaining why the average over-all concurrencies are so low.

6 Conclusions

The speedup results for Mesh, Mullet, Mars and other representative Funsim simulations (not shown here) are generally discouraging in terms of PDES applied to VLSI simulation. Chips with many thousands of instances are routinely found to have average concurrencies in the tens rather than thousands or even hundreds.

The simple PDES analysis indicates that the central event queue model with no lookahead is only capable of very small speedups, even given the likely efficiency of the algorithm. With infinite lookahead, the distributed time slot analysis gave speedups that were beginning to become significant (10–20), but these are optimal figures. With practical lookahead and realistic kernel overheads, we expect that only modest single figure speedups could ever be achieved. Comparing the optimal Mesh speedups of the event queue and message passing models (4.0 vs 54.8) clearly shows the potentially drastic effects of maintaining central synchronisation. This result (and other test cases, not discussed here) force us to conclude that there is no central event queue based simulation model that could generate large scale speedups for Funsim style simulators. Nevertheless, even a single figure potential speedup may turn out to be suitable for a tightly coupled small-scale MIMD machine such as the new multiprocessor workstations now becoming available. A speedup approaching four on a four processor ma-

chine would almost certainly be worth the effort of parallelising Funsim, if it could be achieved consistently.

The question of overheads of conservative PDES implementations can only really be resolved for individual architectures and algorithms by writing a complete simulator and testing it. A detailed analysis of a Chandy-Misra queueing network simulator was conducted by Reed *et al.* on a 20 processor shared memory Sequent Balance 21000 in [13]. They concluded that this algorithm can be grossly inefficient for many varied reasons and is basically not viable in the case of queueing network simulation. On the other hand, Soule and Gupta reported favourable results for Chandy-Misra applied to some VLSI test circuits on a simulated multiprocessor [15]. They concentrated on optimisation for deadlock avoidance and recovery, involving selective exploitation of lookahead, to reduce the inefficiencies of Chandy-Misra over DES to as low as 40% for some simulations. This figure is mildly encouraging for our data-flow analysis and indicates that a highly optimised Chandy-Misra implementation could achieve reasonably close to the optimal speedups of between 17 and 55 (see table 3). We stress however, that the simulations we are really trying to speed up here are Mullet (phase 2) and Mars, which have optimal concurrencies of 23.2 and 16.9 respectively — again, certainly not candidates for a massively parallel machine. The need for low overheads is clear, so one could envision that the best case speedups for these simulations would again be achieved on a tightly coupled, small-scale architecture. Whether to use a distributed algorithm like Chandy-Misra, or an event queue based algorithm depends on how many CPUs one is trying to exploit; any more than say four to eight CPUs would necessitate using the asynchronous approach.

There is a clear correlation between static and dynamic fanouts in Funsim simulation models. This suggests that optimisations based on exploitation of large fanout, where it exists, could work well. The optimum parallelism profiles suggest that even large speedups may be achievable in certain phases of these simulations. In particular, optimisations based on exploiting the clocked nature of most large VLSI circuits are likely to be more profitable than global attempts to ‘extract blood out of a stone’. Of course, there are many variations on the basic Chandy-Misra 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. Shared memory ‘tricks’ in lookahead exploitation and deadlock avoidance for avoiding null message traffic is one typical conserva-

tive PDES optimisation [4, 11], others can be found in [3, 8, 5].

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. The authors feel that an investigation into Time Warp's potential benefits when applied to Funsim is definitely warranted. If such a study was to be limited to a trace-driven analysis, like this one, then it would be difficult to investigate the effects of rollbacks, however, it would be possible to predict with some certainty the frequency with which rollbacks are likely to occur. This would be a significant measure of Time Warp's potential in this area.

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] O. Berry and D. Jefferson. Critical path analysis of distributed simulation. In *Proceedings of the 1985 SCS Conference on Distributed Simulation*, San Diego, January 1985.
- [3] 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.
- [4] 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.
- [5] 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.
- [6] E.H. Frank. Exploiting parallelism in a switch-level simulation machine. In *Proceedings of the 23rd Design Automation Conference*, pages 20–26, 1986.
- [7] R.M. Fujimoto. Lookahead in parallel discrete event simulation. In *Proceedings of the 1988 International Conference on Parallel Processing*, pages 34–41, August 1988.
- [8] R.M. Fujimoto. Parallel discrete event simulation. *Communications of the ACM*, 33(10):30–53, October 1990.
- [9] D. Jefferson and P. Reiher. Supercritical speedup. In *Proceedings of the 24th Annual Simulation Symposium*, pages 159–168, 1991.
- [10] D.R. Jefferson. Virtual time. *ACM Transactions on Programming Languages and Systems*, 7(3):404–425, July 1985.
- [11] 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.
- [12] M.W. Rawling. Potential parallelism in VLSI circuit simulation. Technical Report TR-DB-91-10, CSIRO Division of Information Technology, 1991.
- [13] D.A. Reed, A.D. Maloney, and B.D. McCredie. Parallel discrete event simulation: A shared memory approach. Technical Report 650, University of Illinois CSRD, 1987.
- [14] 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.
- [15] L. Soule and A. Gupta. Parallel distributed-time logic simulation. *IEEE Design and Test of Computers*, pages 32–48, December 1989.
- [16] K.F. Wong and M.A. Franklin. Parallel discrete event simulation. *Journal of Parallel and Distributed Computing*, 7(2):416–440, July 1989.
- [17] 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.