

Area Virtual Time

Johannes Schneiders



Doctor of Philosophy
Institute for Computing Systems Architecture
School of Informatics
University of Edinburgh

2004



Abstract

A novel synchronisation algorithm is presented for distributed discrete-event simulation (DDES), called the Area Virtual Time (AVT) algorithm. DDES algorithms can be characterised by two orthogonal features: the *synchronisation policy*, which is either conservative or optimistic; and the *time-keeping mechanism*, based on either Local Virtual Time (LVT) or Global Virtual Time (GVT). The AVT algorithm is based on a network of virtual time regions, which permits different parts of the simulation model to run either one of the time-keeping mechanisms. This is particularly suited to models which are less than homogeneous, in which case mapping the models entirely to either one of the time-keeping schemes would be inefficient. The AVT-algorithm was first simulated, which yielded promising performance results, which were confirmed by implementing it on a Beowulf cluster of PCs. The results demonstrated that the AVT-algorithm progresses the simulation times faster for a greater part of the parameter space than either the LVT or the GVT schemes, and is less sensitive to variations in some key model and communication parameters.

Acknowledgements

My thanks go to

D. K. Arvind, my advisor, for his guidance and comments,
my parents and my mother-in-law for their support,
Lennart Beringer, for his help in submitting the thesis,
and last and foremost my wife Thamarai, for sticking it out with me
during the too many years this work took to complete.

I also wish to thank for the loan of the group laptop, which enabled me to finish this work on a different continent from the one where it started, and for the EPSRC grant covering my tuition fees.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(Johannes Schneiders)

To Thamarai

Table of Contents

1	Introduction	8
2	Background	13
2.1	Discrete Simulation	14
2.2	Discrete Event Simulation	14
2.3	Parallelism in Simulation	14
2.4	Parallel and Distributed Simulation	15
2.5	Events	15
2.6	Virtual Time	16
2.7	Logical Processes	17
2.8	Classes of Simulation Methods	18
2.8.1	Conservative Protocols	19
2.8.2	Optimistic Protocols	20
2.8.3	Hybrid Protocols	21
2.9	Simulation Policy and Time-keeping Mechanism	21
2.9.1	Global Time-keeping	21
2.9.2	Local Time-keeping	22
2.9.3	Optimistic Synchronisation	22
2.9.4	Conservative Synchronisation	23
2.9.5	Hybrid Synchronisation	23
2.10	Null-message Multiplication	24
2.11	Event Fragmentation	24
2.12	Summary	27
3	Related Work	28
3.1	Conservative Algorithms	28
3.1.1	Deadlock Avoidance	29

3.1.2	Deadlock Detection	30
3.1.3	Carrier-null Message Protocol	32
3.2	Optimistic Algorithms	34
3.2.1	Limiting Optimism	36
3.2.2	Adapting the Degree of Optimism	37
3.2.3	Reducing the Cost of Optimism	38
3.2.4	Controlling Memory Consumption	39
3.3	Hybrid Algorithms	41
3.3.1	Unified Framework	41
3.3.2	Local Time Warp	42
3.3.3	Cluster Virtual Time	42
3.3.4	Local Adaptive Protocol	43
3.3.5	Composite ELSA	43
3.4	Summary	45
4	The AVT Synchronisation Algorithm	46
4.1	Area Virtual Time	47
4.2	Description of the AVT Algorithm	50
4.2.1	Definitions	50
4.2.2	The LVT-node	52
4.2.3	The GVT-node	53
4.2.4	The Hybrid-node	62
4.2.5	The AVT-keeper	65
5	AVT-Algorithm – Simulation	67
5.1	The Simulation Environment	67
5.2	Results	69
6	AVT-Algorithm – Implementation	75
6.1	AVTSIM Modelling Interface	77
6.1.1	Entities	77
6.1.2	Communication Between Entities	78
6.1.3	Models	82
6.1.4	Running the Simulation	83
6.2	Example Simulation	84

7	AVT-Algorithm – Evaluation Methodology	86
7.1	Parameters	86
7.2	Result Data	87
7.3	Models	88
7.4	Test Node	90
7.5	Distributed Platform	92
8	Results	93
8.1	Model <i>Echo</i>	93
8.1.1	Results for Parameter <i>Delay</i>	95
8.1.2	Results for Parameter <i>Event Processing Delay</i>	98
8.1.3	Results for Parameter <i>Output Unchanged Probability</i>	100
8.1.4	Summary of Results for Model <i>Echo</i>	102
8.2	Model <i>Test</i>	102
8.2.1	Model <i>Test</i> _{2,1,2,2}	105
8.2.2	Model <i>Test</i> _{4,1,4,1}	109
8.2.3	Model <i>Test</i> _{4,2,4,2}	114
8.2.4	Model <i>Test</i> _{4,3,4,4}	120
8.3	Summary	125
9	Conclusions	128
A	DS-RT 2001: Area Virtual Time	131
	Bibliography	140

Chapter 1

Introduction

Simulation provides one with the means to study the properties and dynamic behaviour of physical systems for which analytical or numerical models do not yet exist or are infeasible.

There are several reasons why one would like to simulate physical systems: they change either too quickly or too slowly for our perception; they are still in the planning stage and therefore unavailable for observation; or it may be too dangerous to exercise certain aspects of a system in the real world.

Simulation is widely used in many areas of science and engineering as a means to study and understand complex phenomena. However, simulating complex systems on a computer can be very time-consuming. Since physical systems usually consist of many components operating concurrently, one can exploit their inherent parallelism. The system is divided into smaller parts and the method of distributed computing is employed to speed up the simulation. The processes replicated onto many processors cooperate to solve the problem in parallel, each dealing with a part of the problem. They need to exchange data and synchronise their actions by using some kind of communication mechanism, such as using message passing.

The area of research which is concerned with issues in distributing a simulation over many computers is called either parallel or distributed simulation. In order to simulate a physical system one has to create an accurate description – a simulation model, that can be executed by a computer. The *physical processes* of the system under simulation are represented by *logical processes* (LP). Changes in state are exchanged between LPs in the form of messages containing time-stamped data which are termed as *events*.

In *distributed simulation* a collection of logical processes is distributed over a loosely-coupled multiprocessor system, where processors are connected by a local area network or a wide area network such as the Internet. Processes run asynchronously in parallel, usually employing message-passing as the means to transmit data and for synchronisation. In contrast, *parallel simulation* is one in which the computation runs on a tightly-coupled multiprocessor computer, possessing high-speed communication links or a shared memory, and the processes perform operations on different data in lock step, while communicating frequently.

In *discrete event simulation* the occurrence of an event is an instantaneous state-change at a point in time, whereas in *continuous simulation*, state changes occur continuously over time. *Time-driven* simulation advances time in steps of fixed size, whereas in *event-driven* simulation, time is advanced according to the time-stamps of the events being processed.

This work is situated in the area of Distributed Discrete Event Simulation (DDES).

DDES can run either under a conservative or an optimistic synchronisation policy. The policy determines the manner in which simulation time is progressed. *Conservative* DDES algorithms observe the strict partial ordering in the sequence of evaluation, i.e. a result at an LP is deemed safe to commit if, and only if, no message with a timestamp less than the local time will ever arrive at that LP (*local causality constraint*). This condition is relaxed in *optimistic* DDES algorithms, in which events can be processed at the LPs in advance of their arrival. Errors due to mis-speculation are detected should they occur, and the algorithm recovers by rolling back in time to a consistent state and then proceeding with the simulation.

The time-keeping mechanism in DDES is orthogonal to the synchronisation policy. Time is maintained in one of two ways: each LP maintains a local clock, called the *Local Virtual Time* (LVT), based on the time-stamps of incoming events; or, the algorithm maintains a *Global Virtual Time* (GVT), which is the minimum time-stamp of all the incoming events and those in transit. An increment in GVT is notified to all the LPs, which update their respective local clocks.

Optimistic DDES algorithms such as the Time Warp [JeffersonS85] use a GVT-based time-keeping mechanism, whereas conservative ones based on the Chandy-Misra-Bryant algorithms [ChandyM79], [Bryant77] use LVT-based time-keeping schemes. The Composite ELSA algorithm [ArvindS92] is an exception to this rule, since it inte-

grates both the conservative and optimistic synchronisation policies, but uses only an LVT-based time-keeping mechanism.

The contribution of this thesis is the introduction of the concept of Area Virtual Time (AVT) as an intermediate in the continuum between LVT and GVT, and a new hybrid synchronisation algorithm for DDES, called the Area Virtual Time (AVT) algorithm. The AVT-algorithm integrates the conservative and optimistic synchronisation policies, as in composite ELSA, and also the LVT and the GVT time-keeping mechanisms. Previous work has combined time-keeping mechanisms, but has not integrated them on a per-LP basis. The AVT algorithm is based on a network of virtual time regions, which is a happy medium between the Local Virtual Time (LVT) and the Global Virtual Time (GVT). The AVT algorithm permits different parts of the simulation model to run either under the LVT or the GVT time-keeping mechanism. This is particularly suited to models which are less than homogeneous. In those cases, mapping the models entirely to either one of the time-keeping schemes would not be efficient.

Figure 1.1 shows the classification of the simulation algorithms based on synchronisation policy and time-keeping mechanism.

We conduct the evaluation of the AVT-algorithm in two steps:

- Simulation of the AVT-algorithm to yield preliminary results.
- The verification of the simulation results together with extensions by means of a test implementation on an actual distributed computing platform.

For the first step, the behaviour of the AVT-algorithm was modelled and embedded into a sequential event-driven simulator, which models a distributed system consisting of processors connected by a network. The second step was conducted by implementing the AVT-algorithm on a distributed computer. The test implementation provides a general-purpose simulator targeted at the simulation of asynchronous systems. It was implemented on a Beowulf cluster of AMD Athlon processor-based Linux computers.

Instead of a number of case-studies to evaluate our approach, we created generic configurable example models, which capture the essence of the behaviour of LPs and the model topology. The AVT-algorithm has been evaluated using these parameterised models which contain cyclic and acyclic sub-models. The parameters determine the topology of the models, i.e. the number of LPs, number of communication connections between LPs, and size of the cyclic and acyclic area, and the behaviour of the models,

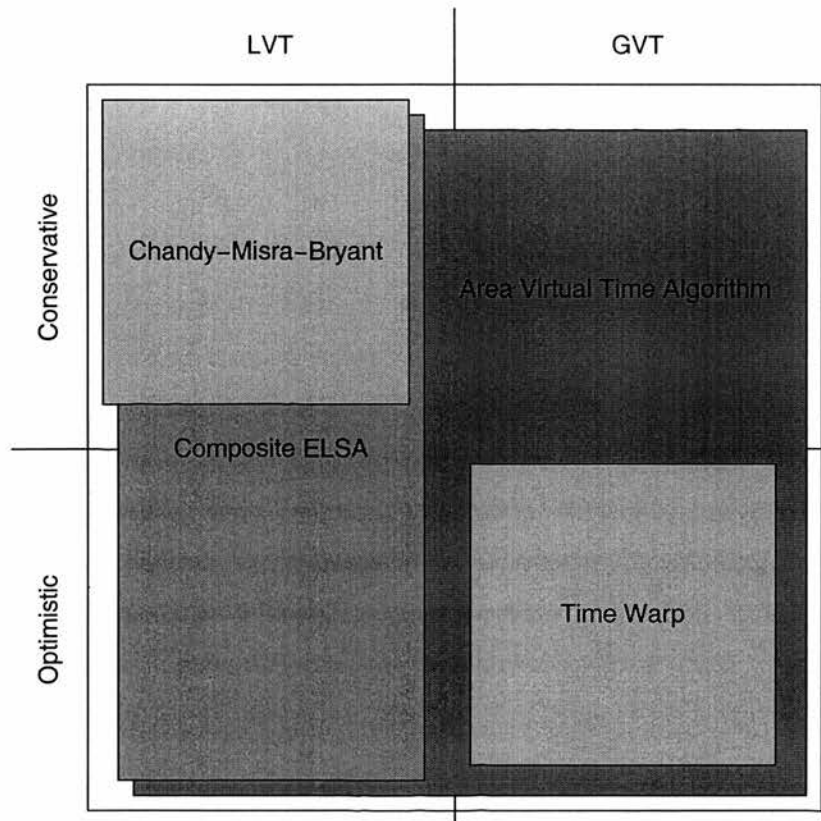


Figure 1.1: Simulation algorithms in relation to methods of synchronisation and time-keeping

i.e. computation times and communication patterns.

As an exhaustive search of the parameter space would be too time consuming, we performed a semi-exhaustive one, selecting representative model configurations and spreading a sample grid of parameter sets for communication times and patterns over the search space, which resulted in a large number of data points.

For each data point we measured the performance of the AVT-algorithm in terms of simulation progress relative to the run-time and compared this to the performance of both the LVT and the GVT time-keeping mechanisms.

Our results show in detail the influence of the parameter values on the performance of the AVT-algorithm, as well as the LVT and the GVT time-keeping mechanism. They demonstrate that the AVT-algorithm progresses the simulation times faster for a greater proportion (around 70%) of the parameter space than either the LVT or the GVT schemes, and that it is less sensitive to variations in some key model and communication parameters.

The thesis is organised as follows:

The next chapter, Chapter 2, introduces and explains important concepts in parallel and distributed simulation.

Chapter 3 gives an overview of related work in the field of distributed discrete event simulation.

A detailed description of the simulated and the implemented version of the AVT-algorithm follows in Chapter 4.

We continue in Chapter 5 with the simulation of the AVT algorithm and associated results.

Chapter 6 contains details of the test implementation, including the simulator modelling interface.

The models and parameters used in the evaluation of the AVT-algorithm are described in Chapter 7.

In Chapter 8 we present the results obtained using the test implementation and the models from the previous chapter.

Finally we interpret the results with conclusions in Chapter 9.

Chapter 2

Background

This chapter introduces and defines important concepts in distributed simulation and elucidates the rationale for developing the Area Virtual Time algorithm.

As in any distributed computing, distributed discrete event simulation decomposes the system spatially and executes those parts on different physical processors. Synchronisation protocols guarantee that events are processed in the correct order, i.e. they enforce the event-causality constraint: the future must not influence the past.

In order to simulate a physical system, one needs to create a sufficiently accurate description – a simulation model. Entities, often called physical processes, in the physical system are represented by *logical processes* (LP). Changes in state, termed as *events*, are propagated by event-messages, which are marked with their time of occurrence called the *time-stamp*. A simulation model can be represented as a directed graph, in which the nodes and the arcs correspond to LPs and communication channels, respectively.

In a distributed simulation, system state and events are physically distributed, which poses the problem of synchronisation. Various solutions to this problem, i.e. different distributed simulation algorithms, have been developed. Two classes of simulation algorithms have evolved: conservative and optimistic. Conservative algorithms preserve causality by processing only those events which cannot cause a causal violation, i.e. events which cannot be intercepted by other earlier events. The observation that such a system spends much time waiting for synchronisation led to the development of optimistic simulation algorithms. These algorithms allow processing of events which might result in a causal violation but store enough information that enables them to re-

cover should one occur, i.e. they “roll back” in time until a consistent state is reached. These systems increase processor utilisation, although not all of it contributes towards advancing the simulation time. Hybrid simulation algorithms, which integrate both conservative and optimistic simulation paradigms, have also been developed.

2.1 Discrete Simulation

Simulation models are usually specifications of physical systems or parts thereof. A simulation emulates the occurrence of events in time and the induced changes in the state of the original physical system.

State-changes in real-world systems often occur continuously over time, which is reflected in *continuous simulation*. However, changes of state in digital computers are discrete. Every continuous simulation model can be transformed into a discrete one by considering both the start and the end of a change. Thus *discrete simulation* may be used to simulate continuous real-world systems.

2.2 Discrete Event Simulation

Two ways of progressing time in discrete simulation have evolved. In *time-driven* simulation, time is advanced in steps of fixed size. In contrast, in *event-driven* simulation time is advanced according to the events being processed. Once the event with the earliest time-stamp has been processed, time can be advanced to the time-stamp of the next earliest event. Thus discrete event simulation (DES) advances time as much as possible when processing each event. This proves to be more efficient than discrete time-driven simulation, especially in cases of events being irregularly dispersed over time.

2.3 Parallelism in Simulation

Spatial Parallelism Since physical systems often consist of many components which could be independent of each other, there usually is a certain amount of inherent parallelism in such a system. This can be exploited by preserving the parallelism

while modelling the system. Events which occur in different components at the same time can be executed in parallel.

Temporal Parallelism It is possible to process events which occur at different times in parallel without violating causality if they do not depend on each other. This is obvious for events which occur at different locations. But it is also possible to process events in parallel which occur at the same component in sequence. This is worthwhile for events whose processing time is higher than their average inter-arrival time at the processor.

Other Kinds of Parallelism There are other kinds of parallelism in simulation which can be exploited: the same simulation with different sets of parameters or inputs, can be run on several computers at the same time; or a sequential simulator could be parallelised at a sub-routine level.

In this work we are only concerned with spatial and temporal parallelism.

2.4 Parallel and Distributed Simulation

In *parallel simulation* the computation runs on a tightly-coupled multiprocessor computer, possessing high-speed communication links or a shared memory, and the processes perform identical operations on different data, while using strict synchronisation among the parallel processes and communicating frequently. In *distributed simulation*, a collection of logical processes is distributed over a loosely-coupled multiprocessor system, where processors are connected by a local area network or a wide area network such as the Internet. Processes run asynchronously in parallel, usually employing message-passing to transmit data and as a means for synchronisation, in order to avoid causal violations. Distributed simulation is a form of distributed computing, with the essential difference being the explicit notion of time. All state changes occur at a certain *virtual time* which is valid throughout the distributed computation.

2.5 Events

An *event* is a change of state at a certain time and place. More precisely, if one imposes a space-time grid on a distributed physical system, then one can associate with every

event in the system, a well-defined point in the space-time coordinate system. An event is the occurrence of a state change at a point in the space-time coordinate system, as shown in Figure 2.1.

An event A causes another event B , if the occurrence of B is an immediate consequence of the occurrence of A and A has a smaller time-coordinate than B , i.e. they adhere to the causality constraint: the future cannot influence the past. In Figure 2.1, event 1 causes events 3 and 4, event 3 causes event 5, and so on.

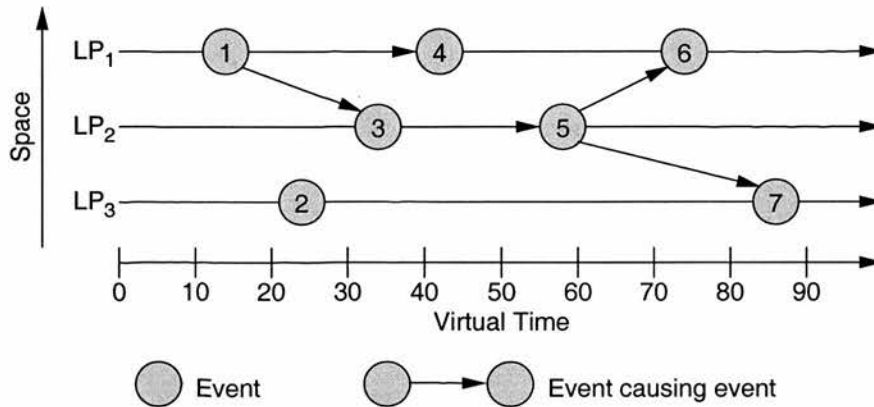


Figure 2.1: Events, Virtual Time, and Logical Processes

2.6 Virtual Time

When simulating the behaviour of a physical system, one uses *virtual time* to model temporal dependencies (causality) and the progress of time which is independent of the real-world time. One may wish to simulate a longer time period in a shorter real-world time period, e.g. when computing the daily weather forecast or a long term climate model it is not useful to have the forecast after the event. Likewise, to enable very short-timed processes to be visible, we can simulate time more slowly than the passing of real-world time, such as the explosion in the cylinder of a car engine. However, if our task is to simulate parts of a real-world system which should cooperate with already physically realised parts of this system, then the real-world time is a hard lower bound on virtual time. The virtual time of simulated events which are not visible to the world may differ from real-time, but the virtual time of events which are propagated from the simulation to the outside world must match real-world time precisely.

While real-world time is the same everywhere in the universe and progresses uniformly

(disregarding Einstein's Theory of Relativity for the purpose of this comparison), virtual time need not be advanced uniformly and not even synchronously in a distributed system, as long as no violation of causality can occur. In synchronous simulation, time is advanced in steps of fixed width and managed centrally so that on every processor the virtual time is the same. In asynchronous simulation every logical process stores and updates its own local virtual time (LVT) according to received and processed events independently from the other logical processes. In asynchronous approaches such as Time Warp [JeffersonS85], time is not advanced uniformly but clock values still need to be collected in one location to compute a global value called the *global virtual time* (GVT).

GVT is defined as the minimum of the virtual times of all the local clocks and the time-stamps of all the event-messages in transit.

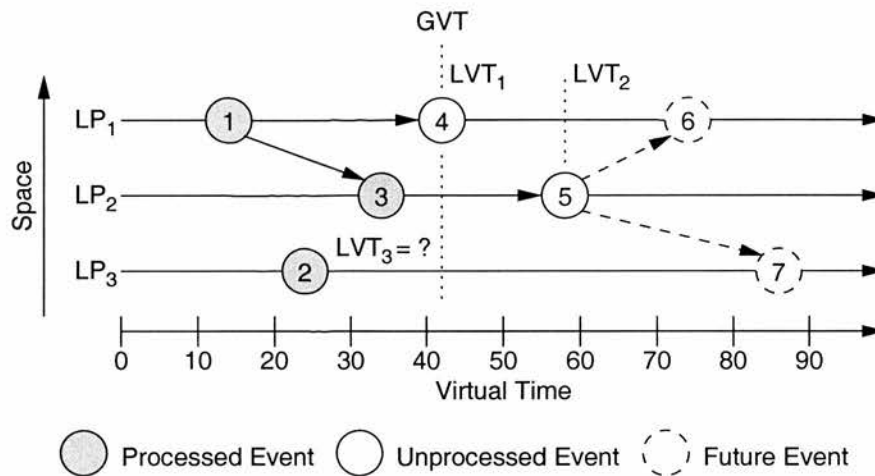


Figure 2.2: Local Virtual Time and Global Virtual Time

Figure 2.2 illustrates the relationship between the LVT and the GVT. As there are no event-messages in transit when the GVT is being computed, the GVT is therefore the minimum of the local clocks. LP_3 's local virtual time is unknown since there is no next event pending. In this special case LVT is assumed to be infinite. The resulting GVT is the minimum of all LVTs, i.e. LVT_1 in this example.

2.7 Logical Processes

The system under simulation is viewed as a collection of *physical processes*. These can be represented by *logical processes* (LP) in the simulation model. A set of LPs

cooperate, each of them simulating a part, in space and time, of the simulation model. The LPs communicate with each other by means of a mechanism which is used to synchronise activities, and to receive and propagate events. Each LP maintains a local simulation time clock, which indicates the time-stamp of the most recent event processed, some local state, and a list of time-stamped events that have not yet been processed. The incoming events are processed in time-stamp order. Local state may be updated, and events generated as a result are propagated to the appropriate LPs. The time-stamp of an event generated by an LP must be greater than or equal to the LP's simulation time clock when the event was processed.

2.8 Classes of Simulation Methods

There are two basic approaches to guarantee causality in simulation algorithms: the conservative, and the speculative or optimistic. Conservative simulation prevents violation of causality by processing and propagating only safe events. These are events which cannot ever cause a violation of causality because there are no unprocessed events on which they depend.

In optimistic algorithms the speculative processing and propagation of uncertain events is permitted and information is stored, which allows the incorrect simulation steps to be undone (rolled back) should causality violation occur. The simulation then resumes at the rolled back state using the corrected information. Output to the real world must not be committed before it is certain that it can never be rolled back.

The possible advantages of optimistic methods over conservative ones is that while a conservative process, waiting for an event, does not utilise all the available computing resources, an optimistic process may use these resources to speculate on future events and, if the speculation is successful, gain an advantage over the conservative one. However, if the speculation fails then incorrectly guessed events have been propagated and the subsequent rollback generates additional load and can slow the simulation down instead of speeding it up. Whether an optimistic or conservative algorithm yields better results depends on the simulation model and the nature of the input data. To address this problem, simulation algorithms which incorporate both approaches have been developed.

2.8.1 Conservative Protocols

Conservative simulation protocols date back to the work by Chandy and Misra [ChandyM79], and Bryant [Bryant81]. Causality is preserved in the following way: Event messages are marked with the time of their creation – the time-stamp. Logical processes (LP) may only process events with a time-stamp not greater than their local virtual time (LVT). LVT is the time for which the LP has been guaranteed not to receive event messages with a time-stamp less than LVT, i.e. messages from the local past. Once an event has been processed, LVT is advanced to the minimum time for which it is sure that no event message with a time-stamp less than the new LVT can arrive. To guarantee the correctness of this algorithm, all events must be processed and all messages must be sent and delivered in chronological order. The requirement to process events in time-stamp order is called the *local causality constraint*. If all LPs adhere to this constraint, then the execution of the simulation model on a parallel computer produces exactly the same result as on a sequential one. In addition, conservative protocols require a static communication topology. Otherwise, an LP cannot determine when it is safe to advance LVT or to process an event.

Deadlocks can occur if a set of processes wait for each other in a closed cycle. These deadlocks are avoided by sending null-messages to all the successors, whenever the processing of an event did not produce a new event-message. A null-message is not related to the simulation model. It is an artifact of the algorithm, and is sent for the purpose of synchronisation and consists only of a time-stamp. It informs the receiver that no event with time-stamp smaller than that of the null-message will be sent by the sender. If a null-message is received, then the local virtual time of the receiver can possibly be increased, thus breaking the deadlock. If an LP's LVT is t , and it can guarantee that it will not send a message with a time-stamp less than $t + l$, regardless of what messages it may receive, the LP has a *lookahead* of l , e.g. the sender can look ahead into the future at least for a period of time which is equal to the service time for the event. Thus it may set the time-stamp of the null-message to LVT, plus the lookahead. This scheme only works if at least one LP in each closed cycle can look ahead into the near future by some non-zero amount.

In practice, however, conservative protocols tend to create an overwhelming amount of null-messages, because logical processes can only look ahead into the very near future. In closed cycles this happens frequently because a logical process basically

waits for itself. So, instead of going through several cycles of null-messaging, each of which advances LVT only by a small amount, it could have sent just one null-message covering the entire safe time period. Some approaches for alleviating this problem are described in Chapter 3.

2.8.2 Optimistic Protocols

The first optimistic protocol – Time Warp – was described by Jefferson and Sowrizal [JeffersonS85]. The basic idea behind Time Warp, as with all other optimistic protocols, is to allow the violation of causality. Should a causal violation occur, i.e. an event arrives with a time-stamp in the local past (straggler), the effects of all events that have been processed, between the local clock value before the straggler arrived and the time-stamp of the straggler, have to be undone. Time Warp performs a rollback in time to the most recent saved state which is consistent with the time-stamp of the event and restarts the simulation from there on. The propagation of incorrect messages is undone by sending anti-messages. When an anti-message arrives at a process, the corresponding positive message in the input queue, which was received previously, is removed (annihilated). Should the positive message have been processed, then the receiving process has to perform a rollback as well. This can lead to rollback chains or even recursive rollback, if the rollback chain spans a cycle of logical processes. In contrast to Chandy-Misra-Bryant type protocols, event messages in Time Warp need not be received in order. It may therefore happen that an anti-message is received before the corresponding positive message. Anti-messages are then saved to annihilate the positive messages when they do arrive.

In order to determine when it would be safe to discard a saved state a global virtual time (GVT) is required. GVT is defined as the minimum of all local clock values and the virtual receive times of all messages in transit. It is guaranteed that no event with a time-stamp smaller than the global virtual time will ever be rolled back. Although optimistic protocols can exploit greater degrees of parallelism and avoid the performance pitfalls of Chandy-Misra-Bryant type protocols, namely blocking and safe-to-process determination, they introduce new ones: GVT calculation and rollback cascades.

Several approaches have been taken to reduce the cost introduced by rollback, some of which are described in Chapter 3.

2.8.3 Hybrid Protocols

Depending on the properties of the simulation model, either conservative or optimistic protocols may perform better. For heterogeneous simulation models it might even be the case that parts of the models are better run in a conservative mode, while other parts run faster using an optimistic strategy.

Hybrid protocols incorporate concepts from both classes of simulation algorithms, i.e. they try to combine the best of both worlds. Composite ELSA [ArvindS92] is one such protocol. It is essentially a Chandy-Misra-Bryant type protocol enhanced with the ability to calculate optimistically and with explicit lookahead information stored in event-messages. The optimism can be adjusted independently for calculating and propagating events. This is described in more detail in Chapter 3.

2.9 Simulation Policy and Time-keeping Mechanism

Optimistic and conservative synchronisation policies are typically used in conjunction with GVT, and LVT with null-messages, as the time-keeping mechanism, respectively. Composite ELSA is an exception to this rule, since it is optimistic but uses LVT as the time-keeping mechanism. In principle, the time-keeping mechanism (GVT/LVT) is independent of the synchronisation policy (conservative/optimistic) of a simulation, resulting in four possible classes of simulation algorithms depicted earlier in Chapter 1, Figure 1.1.

In the following sections, the pros and cons of each synchronisation policy and time-keeping mechanism are described.

2.9.1 Global Time-keeping

Simulation methods using global synchronisation via global virtual time (GVT) assume implicitly that all components in a simulation are connected. This may be an unnecessarily general assumption as models exist where not all components are connected to each other. They support models with dynamic structure but suffer from problems due to scalability as the time-keeping is centralised, and from overhead due to GVT-calculation and its propagation for the synchronisation of processes. Poor esti-

mates for GVT increase the memory overhead, since old state information is preserved for longer than necessary. Imbalanced computation, i.e. some LPs progressing far ahead of others, can lead to long rollbacks and large memory utilisation.

2.9.2 Local Time-keeping

Simulation algorithms using local virtual time (LVT) and direct temporal synchronisation between LPs require a mechanism for deadlock prevention or detection. A common approach for deadlock prevention is the use of null-messages. Null-message protocols suffer from a multiplication of communication for the following two reasons.

Firstly, null-messages need to be sent to all successors of an LP even if they do not require the result the LP just produced. This leads to a potentially large increase in messages solely for synchronisation purposes. In the worst case, i.e. a fully-connected topology, each LP is required to send a null-message to every other LP after processing an event.

Secondly, event fragmentation occurs in cycles of LPs. LVT-based algorithms (unless they use a technique such as carrier-null-messages, as described in Chapter 3) split up the intervals of time between the arrival of events, into small chunks the size of the smallest lookahead in the cycle, and move these chunks around the cycle. This leads to a great increase in the number of (potentially unnecessary) messages sent. Event fragmentation is described in more detail in Section 2.11.

Since LPs require knowledge about which other ones to expect synchronisation messages from, LVT-based schemes do not support applications with dynamic structure, i.e. number of LPs and communication peers for an LP may change, in a straightforward way. To simulate such models, all LPs which can potentially communicate need to be synchronised. If one does this by employing a method similar to null-messages, then the amount of additional connections and synchronisation along them can be overwhelming.

2.9.3 Optimistic Synchronisation

The optimistic approach is prone to memory overhead due to state saving, and processing overhead as the algorithm is more complex than a conservative one. With

increasing simulation size, the risk of cascading rollbacks increases which can cause the simulation to become unstable, i.e. LPs get far ahead in time relative to others only to have their results rolled back and this process repeats, and use an excessive amount of memory for saving state. The optimistic synchronisation policy may also incur the overhead of completing non-preemptable speculative event computations, i.e. ones that have been started just before the arrival of an event that invalidates them, but must run to completion because they cannot be interrupted, which in turn depends on the actual implementation of the simulator.

On the other hand, optimistic methods have the potential to exploit maximum parallelism and are not sensitive to lookahead.

2.9.4 Conservative Synchronisation

Conservative methods perform poorly when the application exhibits poor lookahead. Since no computation takes place if no committed input data is available, LPs may spend considerable time waiting for results from other LPs. Conservative methods generally require less memory than optimistic ones, since they do not need to save speculated states.

2.9.5 Hybrid Synchronisation

Hybrid algorithms such as Composite ELSA, which are based on Chandy-Misra-Bryant type (conservative, LVT) protocols, incorporate optimistic and conservative methods to address the issues related to the synchronisation policy, i.e. certain models or parts thereof may be better run in a conservative or optimistic fashion. However, these hybrid ones still suffer from the problems of algorithms employing an LVT-based time-keeping mechanism:

- Large number of synchronisation messages (null-messages)
- Event fragmentation in cycles

We will consider next each of these two problems in more detail, and compare the LVT, and the GVT time-keeping schemes.

2.10 Null-message Multiplication

In order to synchronise m predecessor LPs with n successor ones, an LVT-based time-keeping scheme requires $n \cdot m$ connections, and as many messages for every time-step (LHS in Figure 2.3). This number is independent of the amount of actual event-messages required between the LPs. Note that it may be necessary to send these time-keeping messages (null-messages) although the LPs in question do not exchange any event-messages. If, however, all these LPs do indeed exchange simulation data, then null-messages do not need to be sent and this time-keeping mechanism does not introduce any additional message overhead.

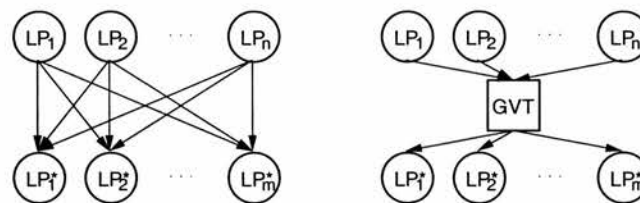


Figure 2.3: Time-keeping connections in local vs. global time-keeping

In contrast, a GVT based scheme only requires as many event-carrying connections and event-messages between LPs as are actually used for the exchange of data. For temporal synchronisation, only $n + m$ connections are required (RHS in Figure 2.3). The total number of connections (and messages for each time-step, assuming GVT is updated for every event processed) is $n + m + d$, where d is the number of data connections required, and $d \in \{0, 1, \dots, n \cdot m\}$. The trade-off in terms of the number of connections required for data exchange and synchronisation between local and global time-keeping is depicted in Figure 2.4.

2.11 Event Fragmentation

LPs connected in a cycle are prone to event-fragmentation when an LVT time-keeping scheme is employed. The amount of fragmentation, depends on the relationship between the inter-arrival time between events and the delay (or lookahead) for executing an event in an LP. Consider the simulation model consisting of two LPs connected in a cycle as shown in the left side of Figure 2.5.

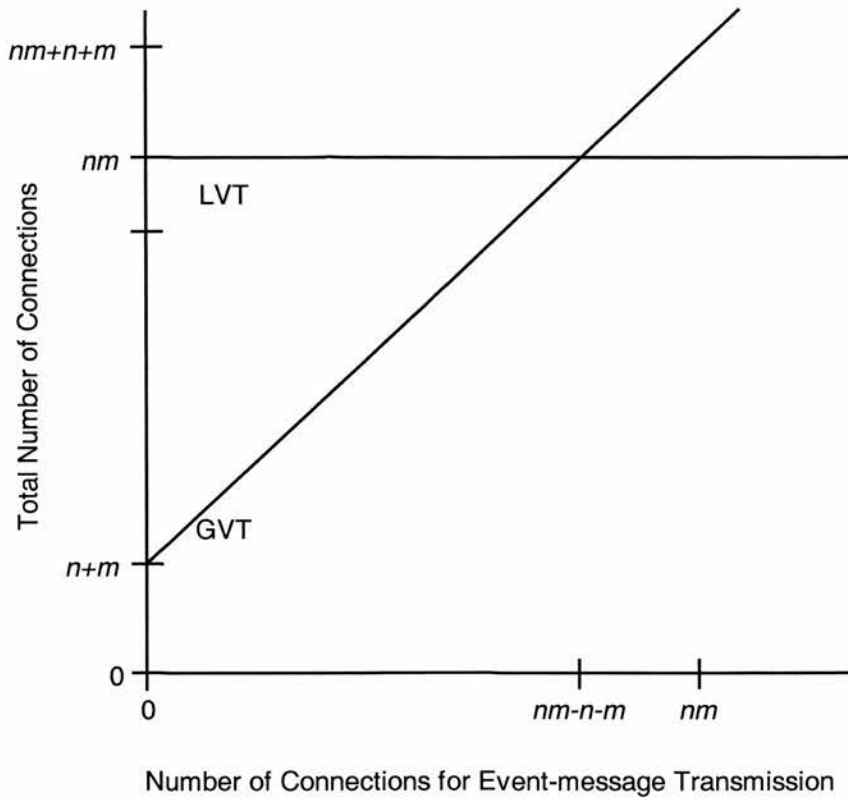


Figure 2.4: Trade-off between local and global time-keeping

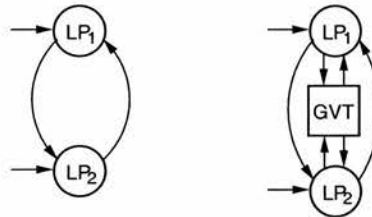


Figure 2.5: Cyclic model with local (left) and global time-keeping (right)

If the virtual time in both the LP s were 100, and the next unprocessed event in both LP s had a time-stamp of 200, then LVT time-keeping would result in null-messages with time-stamps: 101, 102, 103, ..., 200, being sent by both LP s. Only after all these null-messages have been sent and the simulation time has advanced to 200, can the pending events with time-stamp 200 be processed (Top of Figure 2.6).

In cycles of LP s run using an LVT time-keeping algorithm, event fragmentation causes the following number of messages to be sent to advance simulation time by an amount

equal to the time interval between the arrival of two subsequent events:

$$\frac{I}{\delta} \cdot m$$

where I = interval between two successive events, δ = delay of LP, m = number of connections between LPs in the cycle.

A GVT-based scheme requires $2n$ messages per interval to be sent for computing and distributing GVT, in addition to the m messages required to send the events (m = number connections between LPs in the cycle) which results in a total number of messages of

$$2n + m$$

This potential advantage of a GVT-based method only exists if the event sent to LP_1 does not change the LP's output, i.e. the event does not cause the creation of a new event and the optimism proves to be successful. Otherwise the GVT based methods suffer from the same amount of event fragmentation as LVT-based ones, and the messages sent for GVT computation are wasted. The number of messages sent for

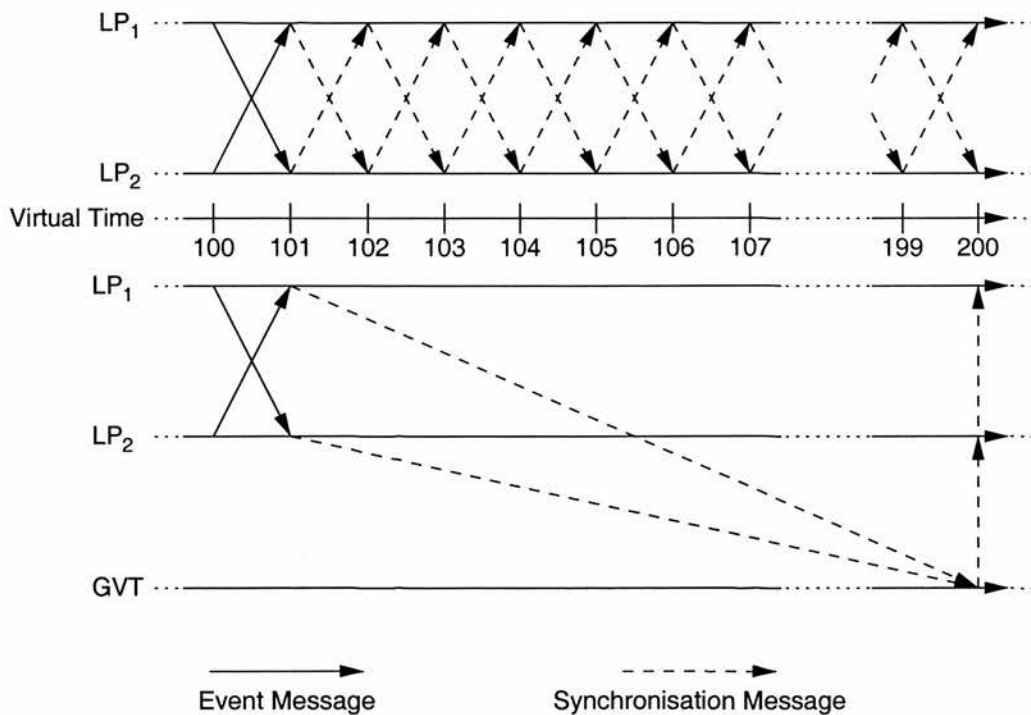


Figure 2.6: Messages required for local (top) and global time-keeping (bottom)

advancing virtual time by one inter-arrival-time is then

$$\frac{I}{\delta} \cdot m + 2n.$$

Also, the optimistic computation of events could cause additional overhead.

2.12 Summary

In summary, we make the following observations regarding the influence of the synchronisation policy and time-keeping mechanism on the performance of DDES.

Simulation algorithms employing a conservative synchronisation policy, generally have lower run-time overheads, but their performance is dependent on the *lookahead*, i.e. the time into the future that a value is known to hold, if they use a LVT time-keeping mechanism. Optimistic algorithms which use either LVT or GVT time-keeping mechanisms, can potentially infuse more concurrency at run-time by relaxing the strict ordering of evaluation, but at a price: the overheads of completion of non-preemptable speculative event computations, saving and restoring of states, as well as cancelling erroneous event-messages can attenuate this gain.

The influence of the time-keeping mechanism on the performance of DDES is as follows: a GVT-based time-keeping mechanism coupled with an optimistic synchronisation policy (assuming good predictability) is better suited for cyclic graphs. On the other hand, LVT time-keeping mechanisms are unable to exploit predictability due to event fragmentation. The amount of fragmentation depends on the relationship between the inter-arrival time for events and their execution time in an LP. In acyclic graphs, event fragmentation does not hamper LVT time-keeping mechanisms and they are better suited due to their lower communication overheads.

Based on these observations, a hybrid algorithm was created, which integrates both time-keeping methods, in order to combine the advantages of both, without incurring too many of the disadvantages of either. This new synchronisation algorithm for DDES is based on the concept of Area Virtual Time (AVT) as an intermediate in the continuum between LVT and GVT. It is called the AVT-algorithm and is described in detail in Chapter 4. The next chapter, Chapter 3, gives an overview of related work in DDES concerned with the problems described in this chapter.

Chapter 3

Related Work

Conservative synchronisation algorithms for distributed simulation date back to the ground-breaking work of Chandy, Misra [ChandyM79], and Bryant [Bryant77] in the late 1970s. In the mid-1980s, Jefferson introduced Time Warp [JeffersonS85], an optimistic distributed simulation algorithm. These and associated works are described in some detail next.

3.1 Conservative Algorithms

Conservative simulation protocols preserve the local causality constraint in the following way.

Event messages are marked with a time-stamp denoting the time of their creation. Logical processes (LP) may only process events with a time-stamp equal to their local virtual time (LVT). LVT is the time for which the LP has been guaranteed not to receive event messages with a time-stamp less than the LVT, i.e. messages from the local past.

An LP possesses a list of events which have been received, but not yet processed. The event with the smallest time-stamp is selected for processing. If the time-stamp of this event is equal to the LVT, then it is processed by the LP, or else the LP blocks. Once an event has been processed, it is removed from the event list and the LVT is advanced to the minimum time for which it is certain that no event-message with a time-stamp less than the new LVT can ever arrive. Events generated as a result of processing are propagated to other LPs.

In order to guarantee the correctness of this algorithm, all events must be processed and all messages must be sent and delivered in chronological order. This implies that the timestamp of the last event-message received on a connection from a predecessor LP is a lower bound on the time-stamp of any subsequent message on that connection. In order to determine when it is safe to process an event, conservative protocols require a static communication topology, i.e. it is fixed as to which LPs send messages to which others.

While this algorithm guarantees that the local causality constraint is not violated, it is susceptible to deadlock. A set of processes may wait for each other in a closed cycle because there are no event-messages with time-stamps which are less than or equal to the LVT in the simulator for any LP in the cycle.

3.1.1 Deadlock Avoidance

In the original work of Chandy and Misra [ChandyM79], and Bryant [Bryant77], deadlocks were prevented through the use of null-messages.

A null-message informs the receiver that no event-message with time-stamp smaller than that of the null-message will be sent by the sender. It consists only of a time-stamp and is not related to the simulation model, but sent solely for the purpose of synchronisation. LPs send null-messages to each successor LP whenever the processing of an event did not produce a new event-message to that LP.

If a null-message providing additional information is received, then the local virtual time of the receiver can possibly be increased, which may lead to some events now being safe to process and thus breaking the deadlock.

A null-message is processed by an LP in a way similar to an event-message. The differences are that the local state is not changed and no new event-messages are generated or sent. However, processing of a null-message can advance the LVT, because LVT is the minimum of all time-stamps received at all inputs. If the LVT increases due to the receipt of the null-message, then the LP in turn sends out further null-messages to all the successor LPs.

The time-stamp of these null-messages can be determined by using the LVT and the information from the simulation model. The LVT is the lower bound on the time-stamp of any future incoming messages. Information from the simulation model provides the

minimum time to service an event, the *lookahead*. The LP may set the time-stamp of the null-messages to LVT plus the lookahead, since it will never produce an event with a time-stamp less than that value. The receivers of the null-messages are enabled to re-compute their LVT and send null-messages based on this information to their neighbours, and so on.

This scheme only works if at least one LP in each closed cycle can look ahead into the near future by some amount greater than zero. It has been shown that the null-message algorithm prevents deadlock if the above condition is met [ChandyM79].

In practice, however, this protocol tends to create an excessive amount of null-messages as described in Sections 2.10 and 2.11.

3.1.2 Deadlock Detection

Alternatives to the use of null-messages are algorithms that allow deadlocks to occur, but are able to detect and to break them. For deadlock detection, any general distributed termination or deadlock detection algorithm can be used. Once deadlock is detected, it can be broken by employing the following methods:

- Processing the event with the lowest time-stamp in the simulation.
- Computing a lower bound on the safe virtual time (SVT) and using it to determine the set of events that are safe to process. The safe time can be computed by each LP as a lower bound on the time-stamp of event-messages that it might later receive from predecessor LPs as the minimum of
 - the lowest event time-stamp plus the lookahead in an active LP,
 - the LVT plus lookahead in a blocked LP,
 - and the time-stamps of all messages in transit.

A method to detect and correct deadlock using the first approach is described in [Misra86]. A prerequisite for this deadlock detection scheme, is that all connections are first-in, first-out. It makes use of a marker message which circulates through the simulation on a precomputed cyclic path encompassing all connections between LPs (to create such a path connections may be added). LPs are coloured white to denote that they have been visited by the marker since having last sent or received a message, or black otherwise. The marker message comprises of a counter of white LPs visited

since the last black one, the minimum time-stamp of event-messages received at the white LPs, and the identification of the white LP containing the minimum time-stamp. All LPs are coloured black initially. When an LP either receives or sends an event-message it sets its colour to black. When an LP receives the marker, the action taken depends on its colour:

Black Once the LP becomes idle, it colours itself white, resets the counter in the marker, sets the marker's minimum of time-stamps and LP identification to its own, and then transmits it along the precomputed path.

White The marker's counter is incremented by one, the minimum of time-stamps is updated, and, if it changed, the identification is set to that of the LP. An LP can identify deadlock if the counter in the marker reaches the number of connections. If no deadlock is detected, then the marker is propagated along its path.

In order to break the deadlock, the LP identified in the marker as containing the minimum of the time-stamps of received event-messages, can be activated to process its first event and thus to progress the simulation.

A conservative algorithm that implements the second method, i.e. computing SVT, is described in [ChandyM81]. It runs in alternate phases:

Parallel phase: The simulation runs until it deadlocks. This is detected by a controller employing a distributed deadlock detection algorithm described in [MisraC82]. Other distributed deadlock detection schemes not employing a central controller could be used as well, e.g. as described in [ChandyMH83].

Phase interface: The controller initiates the SVT computation in all LPs, allowing some LPs to advance the LVT and recover from the deadlock. Each LP_i computes the time-stamp, (U_{ij}) , of the next event-message output along any edge, (i, j) , assuming no further input is received. The minimum of all U_{ij} is already a usable lower bound on the time-stamp of the next event-message in the system. However, this is improved on by taking into account the dependency along paths formed by the connections between LPs and the lookahead at each LP.

In order to find the best possible lower bound, (W_{ij}) , on the time-stamp of the next message to be transmitted along any communication connection (i, j) , the LPs cooperate in computing the W_{ij} in a distributed manner. The computation at each LP consists of n cycles ($n =$ number of LPs), where in the k^{th} cycle LP_i computes $W_{ij}^{(k)}$ and sends it to LP_j , for every outgoing connection (i, j) , and

receives $W_{hi}^{(k)}$ along every incoming connection (h, i) , which is used to compute $W_{ij}^{(k+1)}$ in the $(k + 1)$ th cycle.

Initially $W_{ij}^{(1)}$ is set to U_{ij} . $W_{ij}^{(k+1)}$ is computed as the time-stamp of an event-message if one is pending to be sent to LP_j , or as

$$W_{ij}^{(k+1)} = \max(t_{ij}, \min(U_{ij}, \min_h \{W_{hi}^{(k)}\})),$$

where t_{ij} is the time-stamp of the previous message, otherwise.

After n iterations the computation is complete and $W_{ij} = W_{ij}^{(n)}$. LP_i can be certain that it will not receive any event-message with a time-stamp less than W_{hi} on input connection (h, i) , and that it will not send an event-message with time-stamp less than W_{ij} on output connection (i, j) . LP_i updates its local clock, sends a signal to the controller, and then enters the next parallel phase.

The algorithm has the property, that in every parallel phase at least one event will be processed which causes at least one event-message to be generated and propagated before the next deadlock; however, it is hoped that many LPs can be activated in each parallel phase.

3.1.3 Carrier-null Message Protocol

The carrier null-message protocol addresses the inefficiency of using an LVT time-keeping mechanism for models with cycles. In closed cycles it happens frequently that a logical process is basically waiting for itself. Instead of going through several cycles of null-messaging, each of which advances LVT only by a small amount, it could send just one null-message covering the entire safe time period. The carrier-null message protocol [WoodT94] implements this idea by adding route and lookahead information to null-messages.

The route information keeps a record of the path a null-message took through the LPs, and the lookahead information contains a lower bound on the earliest time-stamp of any event-message which can subsequently be received by any of the LPs visited.

The cyclic areas in the simulation model (termed strongly connected components in [WoodT94]) are precomputed as sets consisting of LPs which are cyclically dependent with the property that LPs from different sets are not. (A previous carrier-null scheme [CaiT90], detects cycles dynamically when a carrier-null returns to an LP which it had

visited before. This scheme is unable to reduce the number of null-messages when encountering some nested cycles [WoodT94].) The inputs to an LP_i are divided into cyclic ones, i.e. the inputs from other LP_j within the same cyclic area, and non-cyclic ones, i.e. those from all other LPs. Each cyclic area can be safely simulated up to the minimum of the time-stamps in event-messages received on its non-cyclic inputs.

In an extension to an ordinary null-message, which consists only of a time-stamp t , a carrier-null message is of the form (t, ct, r, ncc) , where ct (creation time) is the creating LP's LVT; r (route) is a list of the LPs the message has visited; and ncc (non-cyclic ceiling) is the minimum nc-ceiling of the LPs it has visited, where the nc-ceiling of each LP is the minimum time-stamp of all its non-cyclic inputs.

In order to process carrier-nulls, each LP has a list of those LPs in its cyclic area and it also stores the time-stamp of the latest event-message sent to any other LP in its cyclic area. As the topology of the simulation model is determined before run-time, the cyclic outputs of the LPs are marked to denote that they should transmit carrier-null-messages instead of ordinary null-messages. The normal creation and transmission of null-messages is extended for cyclic outputs in the following way:

1. If an LP_i has received no carrier-null since the last activation, then carrier-nulls, $(t = LVT, ct = LVT, r = \{i\}, ncc = \text{minimum time-stamp of } LP_i\text{'s non-cyclic inputs})$, are created and sent.
2. If an LP_i receives a carrier-null (t, ct, r, ncc) , then if ct is less than the time-stamp of the latest event-message sent by the LP, then the carrier-null is discarded and new ones are created and sent as in case 1; else updated carrier-nulls are sent as $(t' = t, ct' = ct, r' = r \cup \{i\}, ncc' = \min\{ncc, \text{minimum time-stamp of } LP_i\text{'s non-cyclic inputs}\})$.
3. A carrier hit occurs, if an LP receives a carrier-null whose route encompasses exactly all those LPs in its cyclic area. The ncc is then a lower bound on the time-stamp of any event-message that can be received subsequently at any cyclic input of the LP. The carrier-null is discarded, new ones are created and sent as in case 1, and the LVT is updated to: $LVT = \min\{ncc, \text{minimum time-stamp of the LP's non-cyclic inputs}\}$.

The advantage of this approach is that when a carrier hit occurs, the LP can potentially

increase the LVT by a large amount since no event-message with a time-stamp less than or equal to the carrier-null's *ncc* can arrive at a cyclic input of the LP, thus greatly reducing the number of null-messages. However, this comes at a cost, as the length of the carrier routes (and with it the size of the carrier-null messages) increases directly with the number of LPs in a cyclic area.

3.2 Optimistic Algorithms

Time Warp was the first optimistic protocol. It introduced fundamental concepts widely used in the class of optimistic synchronisation algorithms. The basic idea behind Time Warp is to allow the violation of causality, but with the ability to detect and then recover from such a violation. If a causal violation occurs, i.e. an event-message arrives with a time-stamp in the local past (straggler), the effects of all events that have been processed between the local clock value before the straggler arrived and the time-stamp of the straggler have to be undone. Time Warp performs a rollback in time, restoring the most recent saved state consistent with the time-stamp of the event, and restarts simulation from there on. The transmission of messages is undone by sending anti-messages. When one arrives at a process, a previously received corresponding positive message found in the input queue is removed (annihilated) and, if the positive message has already been processed, the receiving process has to perform a rollback, possibly producing additional anti-messages. In contrast to Chandy-Misra-Bryant protocols, event messages in Time Warp need not be received in order. It is possible that an anti-message is received before the corresponding positive message. Anti-messages are then saved to annihilate the positive message when that is received. To determine when it is safe to discard a saved state and to commit I/O operations, one needs to compute a lower bound on the time-stamp of future messages that may still be received. This lower bound, called the *global virtual time* (GVT), is defined as the minimum of all local clock values and the time-stamps of all event-messages in transit. It is guaranteed that no event with time-stamp smaller than GVT will ever be rolled back.

GVT computations are essentially the same as the SVT computations used in conservative algorithms to break deadlock. This is because rollbacks are caused by causal violations which the use of SVT computation prevents.

The actual frequency of GVT computation depends on a trade-off: high frequency

results in lower memory consumption, due to more frequent fossil collection, but it increases the processor time and network bandwidth needed, which can slow simulation progress.

A GVT computation algorithm employing a central GVT-manager is described in [Samadi85]. The algorithm triggers GVT computation through a central GVT manager which sends a GVT-start message to all LPs to initiate GVT computation. To keep track of event-messages in transit, every event-message is acknowledged. An LP reacts to a request for GVT-computation by sending the minimum of all time-stamps of unacknowledged messages in the LP's output queue and the LP's local GVT estimate, to the GVT-manager. The GVT-manager performs a min-reduction, and, once all LPs have replied, broadcasts the result to all the LPs.

Another algorithm, [Bellenot90], reduces the communication complexity by using two binary trees, which needs $O(\log(N))$ time, where N is the number of LPs, and sends less than $4N$ messages per GVT estimation. The binary trees are connected by their leaves, which together encompass all the LPs, and are embedded in the simulation graph as the communication structure for GVT computation. The first root LP sends GVT-start to its neighbours, which propagate it until it reaches the second root LP. When the second root LP receives GVT-start from both neighbour LPs, it replies by sending its GVT estimate. These LPs in turn propagate the minimum of their GVT-estimate, and the minimum of the GVT-estimates received, until this process reaches the first root LP. The resulting GVT-estimate at the first root is then propagated throughout the GVT communication structure.

Distributed GVT estimation algorithms which are independent of any particular communication topology or a global controller have also been developed. In [Mattern93] a distributed snapshot algorithm is used to approximate GVT which does not require channels to be FIFO or messages to be acknowledged. Two colours are used to indicate whether an LP has taken its local snapshot and whether a message was sent before or after this, allowing the algorithm to detect if a message would make the snapshot inconsistent, and to catch messages which are in transit at the receiving LP. To determine when the snapshot is complete, i.e. when all in-transit messages have been caught, a distributed termination detection scheme is employed. GVT approximation is realised by making two cuts and to ensure that no messages cross both cuts, i.e. by taking further cuts until the first and the last cut conform to the condition. Then the minimum of the time-stamps of all messages which cross the last cut can be determined from the

messages which were sent between the two cuts.

Although Time Warp avoids the performance pitfalls of conservative protocols, namely blocking and safe-to-process determination, it introduces new ones: GVT calculation as well as rollback cascades and excessive memory utilisation due to overly optimistic execution, i.e. some LPs advancing too far ahead of others.

Several approaches have been taken to reduce the cost introduced by rollback. Some limit the degree of optimism to address these problems, and some even adapt the degree of optimism in response to the behaviour of the simulation. Others specifically target the memory utilisation problem either by saving state less frequently, incrementally, or not at all, by instead employing reverse computation to implement rollback.

3.2.1 Limiting Optimism

3.2.1.1 Moving Time Window

The Moving Time Window (MTW) protocol [SokolBW88] restricts the amount of optimism by defining a time window as a temporal interval $[GVT, GVT + w]$, where w is the user-specified global window size. Only events having a time-stamp within this interval are eligible for processing. When GVT is computed, the time window is moved forward, and later events may move into the window. The rationale for MTW is that events in the near future are less likely to be rolled back than events further into the future. Adjusting the parameter, w , is a trade-off between the better exploitation of parallelism and greater synchronisation cost, i.e. rollbacks, for large and small windows, respectively. When an LP runs out of eligible events, it may initiate window movement. If the number of LPs with eligible events drops below a *polling threshold*, then the initiating LP polls all other LPs to report the time of their earliest event in order to determine GVT. If all the LPs have progressed their virtual time beyond the old GVT, then the window is moved forward to the new GVT.

3.2.1.2 Local Optimism

The approach in [DickensR90], although allowing optimistic simulation progress, delays the sending of event-messages until it is guaranteed that the send will not be later rolled back, i.e., until GVT advances to the event's time-stamp. Therefore, a rollback

can only be local to an LP, which prevents cascaded rollbacks and also eliminates the need for anti-messages.

3.2.1.3 Breathing Time Buckets

The Breathing Time Bucket protocol [Steinman91] is an optimistic protocol that has a conservative message transmission policy. As in [DickensR90] this restricts potential rollback to be local to an LP. All events are sorted into time buckets such that each bucket contains only events which are causally independent. The size of the bucket is chosen to contain the maximum number of causally-independent events, i.e. events that can be executed concurrently. The local event horizon is the minimum time-stamp of any event that is generated as a result of processing the events within the time bucket. The global event horizon is defined to be the minimum of all local event horizons. Events are executed optimistically but only event-messages with a time-stamp smaller than the global event horizon are propagated, since they will not affect the current time bucket. To determine when the last event in a bucket has been processed, and the event-messages generated can be distributed, the LPs must signal that their LVT has progressed past their local event-horizon. When all LPs have done this, they can be requested to send their event-messages and a new bucket is computed. It is hoped that a sufficiently large number of events is processed in each bucket in order for Breathing Time Bucket to run efficiently.

3.2.2 Adapting the Degree of Optimism

3.2.2.1 Probabilistic Distributed Simulation Protocol

The probabilistic distributed simulation protocol [FerschaC94] adapts how far into the future optimistic execution is permitted, based on observation of event inter-arrival times in the past. If the time-stamp of an event lies in the interval $[s, u]$, it allows the simulation to progress up to the expected time-stamp of the next event $t(O)$, $s \leq t(O) \leq u$. $t(O)$ is an estimate based on the inter-arrival times, $O = (d_1, d_2, \dots, d_n)$, observed during a time window by an LP. Further optimistic progress is controlled by the confidence in the estimate, i.e. the probability that an LP is allowed to progress further, depends on the confidence. The arrival pattern observed in O is used to adapt

the LPs behaviour dynamically to the best tradeoff between conservative and optimistic synchronisation.

3.2.2.2 Elastic Time Algorithm

Elastic Time [SrinivasanR98] is a near-perfect state information protocol. These are characterised by the use of near-perfect state information to compute the error potential and the use of this to control optimism adaptively. The term *elastic time* is derived from the interpretation of the error potential as an elastic force that sometimes either restricts or releases the progress of an LP in virtual time. The Elastic Time Algorithm uses temporal information (logical clock, unreceived message time, and next event time) to compute the error potential. This is then used to control optimism by introducing delays which are proportional to the error potential between event-computations. The delays are asynchronous and may be different for each LP; and, the optimism is adapted after each event-computation.

3.2.3 Reducing the Cost of Optimism

3.2.3.1 Lazy Cancellation

In contrast to the behaviour of the original Time Warp which sends anti-messages immediately upon receiving an event-message with a time-stamp in the local past, lazy cancellation delays sending anti-messages until it is certain that the recomputation after the rollback yields an event different from the previously-propagated one (sometimes wrong computations produce correct results) [Gafni88]. This scheme avoids unnecessary cancellation of correct event-messages, but incurs the cost of additional memory overhead for the deferred anti-messages and of delaying the annihilation of actually wrong event-messages which may be used in further incorrect computations at other LPs.

3.2.3.2 Lazy Re-evaluation

Lazy re-evaluation delays discarding saved states during a rollback until it is proven that they are incorrect [West88]. If the recomputation after the rollback reaches a state that matches a saved one and the input events are the same as before in that state,

then the LP can progress to the time before the rollback occurred. This mechanism prevents the recomputation of correct states, but causes additional memory and book-keeping overhead for copies of the input queue in every state, which is needed to verify that the saved and the current input queue are equal and the LP can progress without re-evaluation.

3.2.4 Controlling Memory Consumption

3.2.4.1 Message Sendback

In order to recover from memory overflow, due to optimistic execution, Time Warp employs the message sendback mechanism [Jefferson85]. If an LP runs out of memory to store an incoming event-message, space is recovered by returning one or more unprocessed event-messages, possibly including the one that just arrived, back to their respective sender(s). These are caused to roll back to the respective state they were in before they sent the returned event-message. Event-messages with the latest time-stamps are chosen to be returned over ones with earlier ones, as the former are more likely to contain incorrect information, and any rollback caused by their return is likely to be shorter in virtual time.

3.2.4.2 Cancelback

The cancelback scheme [Jefferson90] is the first optimal memory management protocol. A memory management scheme is considered to be storage optimal if it has the same order of storage requirement as the sequential DES. The cancelback scheme allows memory to be freed in any LP which need not be the one that ran out of memory. Whether the overflow was caused by an incoming event-message, the need to save a new state, or the creation of a new event-message, the memory for the object with the latest time-stamp *globally* is reclaimed, irrespective of either its type or which LP stores it. Whereas cancelback allows Time Warp to run with a limited memory size of the same order as the sequential DES, in distributed systems it introduces overhead to collect the information required to determine which object to free globally. However, this cost is reduced in shared memory systems.

3.2.4.3 Artificial Rollback

Artificial rollback is a memory management scheme [LinP91], which invokes the rollback procedure to reclaim memory. This is permitted as rollback does not affect the operational correctness of Time Warp. Artificial rollback is similar to cancelback in that the cancellation of an incoming event-message is equivalent to an artificial rollback of the event-message's sender. The cancellation of a saved state or an output event-message in an LP is equivalent to an artificial rollback of that LP. The implementation of artificial rollback is simpler than cancelback because the rollback mechanism is already available. An artificial rollback is performed when the amount of free memory is too small to satisfy the requirement in an LP. In the shared memory implementation in [LinP91], the issues of which LP to roll back and how far are governed by a processor scheduling that guarantees a certain amount of free memory and rolls back events with the latest time-stamp. However, in a distributed memory setting, to select the LP to roll back may be more complicated. A lower bound for the earliest time to roll back to is given by GVT.

3.2.4.4 Infrequent State Saving

State saving can be performed infrequently, i.e. periodically at every i^{th} event-computation rather than at each one, to reduce the state saving overhead [LinL89]. Two situations can arise when an LP is rolled back: the state to which the LP is rolled back has either been saved, which results in a normal state restoration, or it has not. To reconstruct the state in the latter case, the LP starts with the most recent saved state prior to the rollback time and then re-executes until the required state is restored. The cost of this re-execution is lower than that of normal event-computation since only the LP's local state is modified and no event-messages are generated or sent. Periodic state saving reduces the memory requirement for state saving at the expense of increasing the state restoration time.

3.2.4.5 Incremental State Saving

Incremental State Saving [BauerSK91] saves only the *change* in order to return to the previous state for an event-computation. When a rollback occurs, the state to which the LP is rolled back, is reconstructed by applying in reverse sequence the saved state

increments to the current state before the rollback. The advantage of this technique is the low memory usage and reduced state saving overhead if the differences to be saved in each event-computation are small. However, large rollback distances will result in increased computation time to restore the state from the increments.

3.2.4.6 Reverse Computation

An alternative to state saving and restoring, is the use of reverse computation techniques to implement rollback [CarothersPF99]. An event computation is rolled back by executing the inverse computation, e.g., to undo incrementing a state variable, the variable is decremented. This technique avoids state saving and the memory utilisation overhead associated with rollback. However, it requires code to be available to undo every possible computation performed by the simulation. In [CarothersPF99] a reverse compiler is described that automatically generates inverse computations. Destructive operations, such as assignment or modulo computation, can be reversed by restoring a previously-saved value. In this case reverse computation degenerates to state-saving. However, even if the individual steps of a computation are not efficiently reversible, the computation may be reversible at a higher level, e.g. a swap operation implemented by three assignments, which can be exploited to increase efficiency. The computation time required to roll back is equal to the computation time for progressing, i.e. reverse computation trades off speed for memory utilisation.

3.3 Hybrid Algorithms

Algorithms which integrate or combine conservative and optimistic synchronisation policies and/or LVT and GVT time-keeping mechanisms are termed *hybrid algorithms*. Some examples of these and their relationship to the AVT-algorithm are described next.

3.3.1 Unified Framework

The unified framework for conservative and optimistic DES, called ADAPT [JhaB94], is based on the idea of arbitrary combinations of a global control mechanism (GCM), which is based either on null-messages or GVT (or even both), and a local control mechanism (LCM) which can be either conservative or optimistic. Each LP can be

configured individually to use either LCM and may switch dynamically between them, and the GCM is used over the entire system. Unlike the AVT-algorithm, there is no notion of multiple areas of virtual time, i.e. areas with different GCMs.

3.3.2 Local Time Warp

Local Time Warp [RajaeiAT93] hierarchically combines a conservative time window (CTW) algorithm with Time Warp. The simulation model is divided into clusters of LPs. At the LP level, Time Warp is used locally in each cluster in order to exploit parallelism, and at the cluster level, CTW is used to control cascade rollbacks. Each cluster has its own cluster virtual time (CVT), i.e. a “local GVT”, and one designated input and output gateway process in addition to the LPs of the cluster. The LPs of a cluster progress the simulation in the same way as in Time Warp where GVT is replaced by CVT. All event-messages to LPs outside a cluster are sent via its output gateway and the input gateway of the receiving LP’s cluster. Output gateways do not send any event-messages until CVT becomes greater than or equal to their time-stamps, i.e. until they are safe from rollback. The gateway processes of the clusters are controlled by a modified CTW algorithm which ensures that no event-message ever arrives at an input gateway with a time-stamp less than the time of that gateway. Input gateways deliver only those event-messages to LPs whose time-stamp is within the conservative time window.

Unlike the Local Time Warp algorithm, the AVT-based approach does not have designated input/output nodes per cluster, and an LVT-based protocol is used between AVT region, which can operate either conservatively or optimistically.

3.3.3 Cluster Virtual Time

Unlike the Local Time Warp algorithm, Clustered Time Warp (CTW) [AvrilT95] uses Time Warp to synchronise between clusters of LPs, where each cluster is allocated to a processor and runs a sequential simulation algorithm. CTW is a special case because it takes advantage of the fact that the LPs in a cluster share the same address space.

In contrast, the AVT-algorithm makes no assumptions about how LPs are allocated to processors and uses an LVT-based protocol *between* AVT region.

3.3.4 Local Adaptive Protocol

The Local Adaptive Protocol [HamnesT94] allows the degree of optimism to be adjusted on a per-LP and per-channel basis and uses an LVT time-keeping mechanism. In order to optimise performance, a real-time blocking window (RTBW) is computed for each input connection of an LP as a function of the last and next message's time-stamp, the inter-arrival time, the average rate of increase of virtual time, and multiplied by an adaptation factor. This factor is periodically adjusted to maximise the rate progress in virtual time. If the factor is zero, then RTBW is zero and unlimited optimism is used, while if the factor is infinite, the local adaptive protocol synchronises conservatively.

3.3.5 Composite ELSA

Composite ELSA (Edinburgh Logic Simulator) integrates the concepts of conservative and optimistic control [ArvindS92]. It is an LVT-based, conservative protocol enhanced with the ability to calculate optimistically and with explicit lookahead information stored in event-messages. The optimism can be adjusted independently for calculating (*Degree of Optimism*) and propagating (*Degree of Risk*) events.

The Composite ELSA protocol is related to Chandy-Misra-Bryant protocols in that Composite ELSA does require a static communication topology and has no centralised control like a GVT mechanism. Instead LVT is calculated locally in each LP and simulation time is advanced based on received event-messages only.

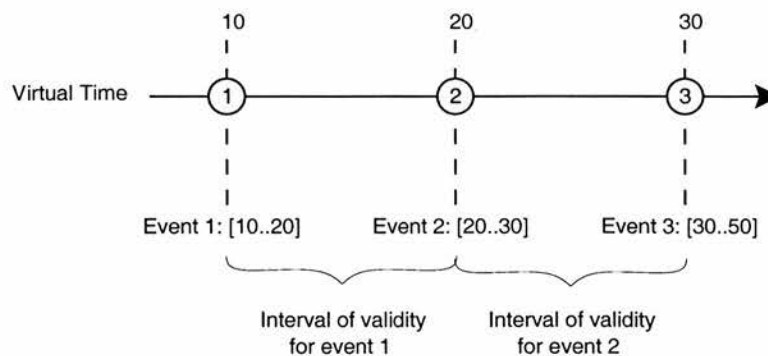


Figure 3.1: Chains of validity intervals in Composite ELSA

While deadlocks in other LVT-based protocols are avoided by sending null-messages, Composite ELSA uses a different mechanism for providing and exploiting lookahead.

Every event-message is not only time-stamped with its creation time but also with its interval of validity which provides lookahead information. Event-messages are tuples consisting of four fields, of the form $(v, t_{start}, t_{end}, \Delta_c)$: where v is the value, t_{start} and t_{end} are start- and end-times for which it is valid, and the *Degree of Confidence*, $(\Delta_c \in \{certain, guess\})$, which denotes whether the value is certain or guessed.

It is guaranteed that the next event will occur exactly at the end of that interval of validity (Figure 3.1). Because of these chains of event validity intervals, LPs can always determine LVT exactly. If the validity interval of a value ends, a new adjacent event has to be sent even if the value did not change. This corresponds to sending a null-message in other LVT-based protocols. By requiring that each LP in a simulation has a non-zero time increment, Composite ELSA adheres to the deadlock avoidance property of Chandy-Misra-Bryant protocols.

Since only those “null-messages” are sent which are absolutely necessary, due to the lookahead information encoded in the event-messages, Composite ELSA avoids much of the overhead of null-messaging protocols. However in cycles, still a great number of event-messages must be sent around the cycle to advance time, if the lookahead in a cycle is small (event fragmentation).

Optimism is handled by allowing LPs to compute values ahead of LVT under the restriction that it must be possible to undo erroneous computations. For this purpose Composite ELSA stores not yet committed values similar to Time Warp or other optimistic protocols. Unlike Time Warp, which uses GVT to commit calculations eventually, Composite ELSA uses LVT.

In Composite ELSA, the amount of optimism for every LP can be adjusted individually. Every LP in Composite ELSA possesses a parameter called the *Degree of Optimism* ($0 \leq \Delta_o < \infty$), which defines how far in advance of a safe state a node may evaluate; and the *Degree of Risk* ($0 \leq \Delta_r \leq \Delta_o$) defines how far in advance of a safe state possibly erroneous results may be propagated. An LP, with $\Delta_o = \Delta_r = 0$, is conservative; while an LP, with $\Delta_o > 0$ and $\Delta_r = 0$, is locally optimistic by confining rollbacks to the local process; and an LP, with $\Delta_o \geq \Delta_r > 0$, is globally optimistic (also called risky), and can cause rollbacks across a number of nodes.

This supports the execution of simulation models which are inhomogeneous in that some parts of the model require a conservative and others require an optimistic synchronisation policy, which would lead to substantial performance loss under either

purely conservative or optimistic protocols.

3.4 Summary

Figure 3.2 gives an overview of the algorithms described in this chapter, sorted into categories according to synchronisation policy and time-keeping mechanism.

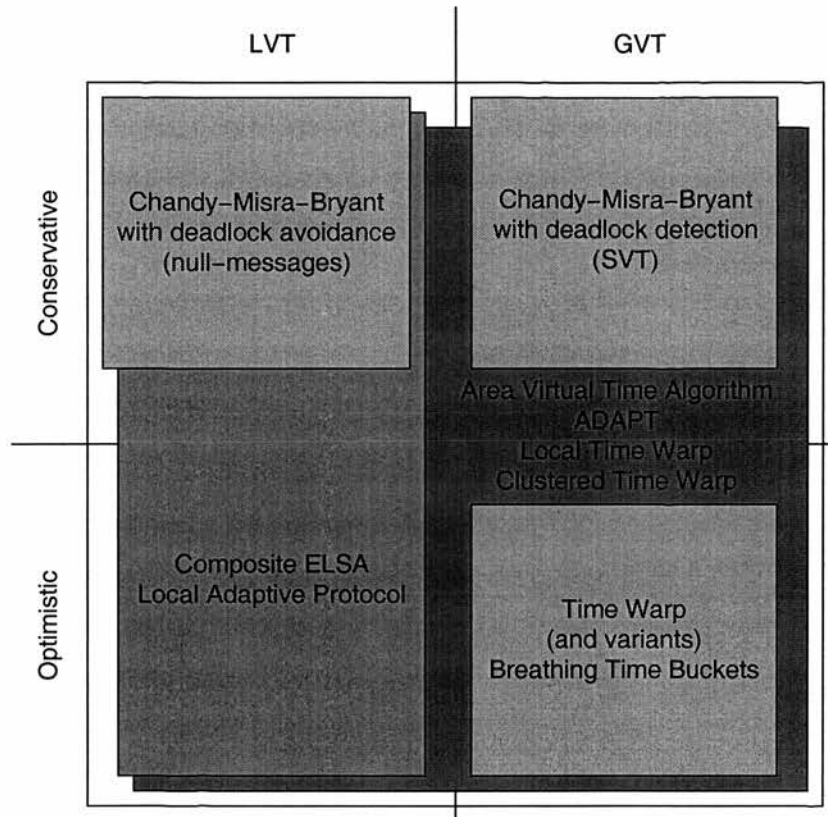


Figure 3.2: Simulation algorithms in relation to methods of synchronisation and time-keeping

Chapter 4

The AVT Synchronisation Algorithm

The following observations are made regarding the influence of the synchronisation policy on the performance of Distributed Discrete Event Simulation (DDES). Conservative algorithms, irrespective of the time-keeping mechanisms used, generally have lower run-time overheads, but their performance is dependent on the *lookahead*, i.e., the time into the future that a value is known to hold, which is a property of the simulation model. Optimistic algorithms, which use either Local Virtual Time (LVT) or Global Virtual Time (GVT) time-keeping mechanisms, can potentially infuse more concurrency at run-time by relaxing the strict ordering of evaluation, but at a price: the overheads due to completion of non-preemptable tasks and the saving of states can attenuate this gain.

The influence of the time-keeping mechanism on the performance of DDES is as follows: In cyclic graphs the LVT time-keeping mechanism is unable to exploit predictability due to event fragmentation (Section 2.11). The amount of fragmentation depends on the relationship between the inter-arrival time for the events and their execution time in an LP. In acyclic graphs, event fragmentation does not hamper LVT time-keeping mechanisms and they are better suited due to their lower communication overheads. On the other hand, the GVT time-keeping mechanism coupled with an optimistic synchronisation policy (assuming good predictability) is better suited for cyclic graphs, as it is less prone to event fragmentation.

We aim to integrate both time-keeping methods with the view of combining the advantages of both, without incurring much of the disadvantages of either. By extending an LVT-based algorithm with a GVT for only a part of the simulation model, we hope

to take advantage of the better lookahead properties of GVT-based algorithms, while avoiding most of the associated overhead.

The model will be divided into a patchwork of areas each employing either LVT or GVT, alongside each other. The areas in which GVT is computed will be restricted to those where the benefits will be the greatest. This “local GVT” is named the *area virtual time* (AVT).

We propose a novel algorithm for DDES called the Area Virtual Time Algorithm which supports both conservative and optimistic synchronisation policies, and is based on a combination of LVT and GVT time-keeping mechanisms. This is a happy medium between the extremes of either each LP maintaining its LVT, or calculating the GVT over all the LPs in the simulation model.

Our hypothesis is that such a hybrid protocol will execute non-homogeneous simulation models more efficiently, while still being capable of running homogeneous models just as efficiently as either the LVT or the GVT time-keeping mechanism.

4.1 Area Virtual Time

In the AVT algorithm, a set of LPs is mapped to a Virtual Time Area (VT-area), which gives rise to a network of VT-areas. The choice of mapping a particular set of LPs to a VT-area is determined by the topology of the simulation graph, i.e. by areas in which the synchronisation overhead using LVT is greater than the overhead induced by AVT (see Sections 2.10 and 2.11, Page 24).

Area virtual time is defined in a manner similar to GVT [Jefferson85]:

$$AVT_i = \min_{j=1,\dots,N;k=0,\dots,M} \{LOVT_j, t_{e_k}\},$$

where VTA_i is the virtual time area i , N is the number of LPs in VTA_i , LP_j is inside VTA_i , $LOVT_j$ (Local Optimistic Virtual Time) is the time up to when LP_j has speculated, M is the number of event-messages in-transit in VTA_i , and t_{e_k} is the time-stamp of the k^{th} event-message in-transit between LPs in VTA_i .

LPs will update their AVT estimate each time their local $LOVT$ is updated, i.e. they will send their $LOVT$ to the AVT-keeper, i.e. the process computing the AVT. The AVT-keeper in turn will notify all LPs in the VT-area, whenever the AVT can be progressed. The AVT-algorithm can progress on LVT alone, if lookahead is available.

The overhead for GVT estimation in this case is reduced to the AVT-messages sent by the LPs to the AVT-keeper. Only if the AVT improves on an LP's LVT, does the AVT-keeper propagate it to the LP.

The simulation model is interpreted as a directed graph, in which the nodes and the arcs correspond to LPs and communication channels, respectively. Each cyclic area of the simulation graph is assigned to a Virtual Time Area (VT-area), in which a GVT time-keeping scheme is used. The AVT algorithm can be applied to LPs which can be configured to use any combination of the modes of synchronisation and time-keeping mentioned in Section 2.9.

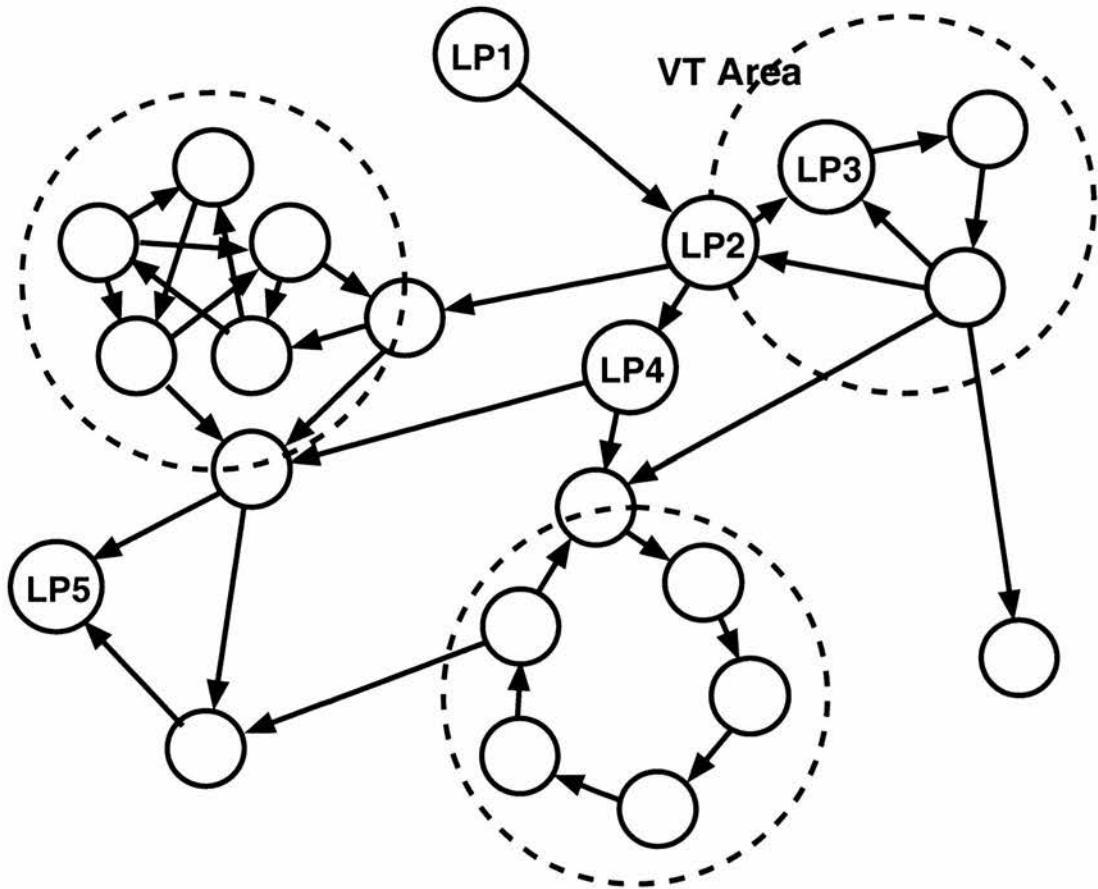


Figure 4.1: Simulation graph

The LPs that do not belong to a cyclic area are termed as *LVT nodes*, each one of which uses LVT as its time-keeping mechanism and can either use a conservative synchronisation policy (corresponding to a Chandy-Misra-Bryant node), or an optimistic one (corresponding to an optimistic ELSA node). Example: LP4 in Figure 4.1. An LVT node becomes active when all its inputs contain events that allow virtual time to be progressed.

Two special cases of LVT nodes are the *input nodes* and the *output nodes*. Input nodes (e.g. LP1 in Figure 4.1) are LPs that generate input from outside of the simulation model, e.g. from an input file, and have only outputs. Conversely, output nodes (e.g. LP5 in Figure 4.1) are used to communicate results from the simulation to the outside world, e.g. a file or the screen, and as such possess only inputs.

In the current instantiation of the algorithm, an LVT node is conservative, although this is not a fundamental restriction: any LVT node could range from conservative, to limited or unlimited locally or globally optimistic - similar to nodes in Composite ELSA, as described in Section 3.3.5.

Any LP that belongs to a VT-area and receives input exclusively from within the area is called a *GVT node* (e.g. LP3 in Figure 4.1). GVT nodes use an optimistic synchronisation policy and a GVT time-keeping mechanism. A GVT node becomes active and computes a new event if at least one of its inputs contains an event that allows virtual time to be progressed. In the current instantiation of the algorithm, output to LPs outside the VT-area is only sent after it has been committed, i.e. to LPs within the VT-area, a GVT node employs globally optimistic (or risky) strategy, while for LPs outside the VT-area it uses a conservative strategy, i.e. it is only locally optimistic.

LPs which receive inputs from both outside and inside the VT-area, are called *hybrid nodes* (e.g. LP2, in Figure 4.1). They interface the LVT time-keeping mechanism outside a VT-area to the GVT mechanism inside. Hybrid nodes employ a GVT time-keeping mechanism and a semi-optimistic synchronisation scheme. The optimism is effectively limited by the look-ahead in the event-messages from outwith the VT-area. Hybrid nodes become active and generate new events, when all inputs from *outside* the VT-area contain events that allow virtual time to be progressed. They may then compute optimistically, but uncommitted results are propagated only within the VT-area (cf. GVT node). Outside the VT-area only committed events are ever transmitted; for LPs outside the VT-area, hybrid nodes are locally optimistic, whereas to those inside the VT-area, hybrid nodes use a globally optimistic (or risky) send strategy.

Hybrid and GVT nodes send their LOVT to the AVT-keeper after every event computation. The AVT-keeper computes the minimum of all the respective LOVTs in the VT-area, and, if the new minimum is greater than the old, then the updated AVT is broadcast to all members of the VT-area whose LCVT (Local Committed Virtual Time – the time up to when an LP has progressed based on committed input) is less than

the AVT. The member LPs then send all committed pending output events (or a null-message) to LPs outside the VT-area.

The VT-areas never emit an uncommitted event-message, i.e. when viewed from beyond, a VT-area behaves as if it were a conservative LVT node, albeit a conglomerate one. To restrict propagation of uncommitted events to the respective VT-area is a design choice, and not a requirement: GVT and hybrid nodes in the VT-areas could send optimistic results to LPs outside of the VT-area. However, a globally optimistic strategy would allow the effect of relatively local rollbacks, confined to a VT-area, to propagate to the remainder of the simulation.

The format of the event-messages are similar to those used by Composite ELSA, i.e., a time interval defines the period for which a value is guaranteed to hold. The AVT algorithm's event-messages have an additional time-stamp to denote the time until when the value is presumed to hold optimistically.

Two slightly different instantiations of the AVT algorithm, are described in this chapter: The first one was used to obtain the simulated performance results in Chapter 5, which assumes the virtual time delay to be fixed for each computation of an event in an LP, and that the current virtual time is not an input parameter to the computation of a new event. The second one, which is geared towards actual implementation, drops these requirements, and allows dynamically changing delays and virtual time to be an input parameter for event computation. Changeable delays result in a reduction of lookahead and the time for which an event holds must be computed differently. In the description of the AVT algorithm in the next section both versions have been included.

4.2 Description of the AVT Algorithm

4.2.1 Definitions

LCVT: Local Committed Virtual Time – the time up to when an LP has progressed based on committed input.

LOVT: Local Optimistic Virtual Time – the time up to when an LP has speculated based on its input

Event-Message is of the form: $(t_{start}, t_{end}, t_{opt}, data)$, where:

t_{start} is the start-time of an event

t_{end} is the time until when the event is committed to hold

t_{opt} is the time until when an event is presumed to hold based on the input

$data$ is the value of the event.

Event-messages can be either: *committed*, i.e., $t_{start} < t_{end}$, and $t_{end} = t_{opt}$; *uncommitted*, i.e., $t_{start} = t_{end}$, and $t_{end} < t_{opt}$; or *partially committed*, i.e., $t_{start} < t_{end} < t_{opt}$.

AVT: Area Virtual Time is defined as the minimum LOVT of the members of the VT-area and the minimum (t_{start}) of all the messages in transit in the VT-area.

AVT-Message is used to communicate between the LPs and the AVT-keeper, and is of the form $(LCVT, AVT)$ and defined as:

LCVT – the sending LP's LCVT, or empty when sent by the AVT-keeper

AVT – the sending LP's LOVT, or the updated AVT sent by the AVT-keeper

Delay (δ) – the virtual execution time of an LP.

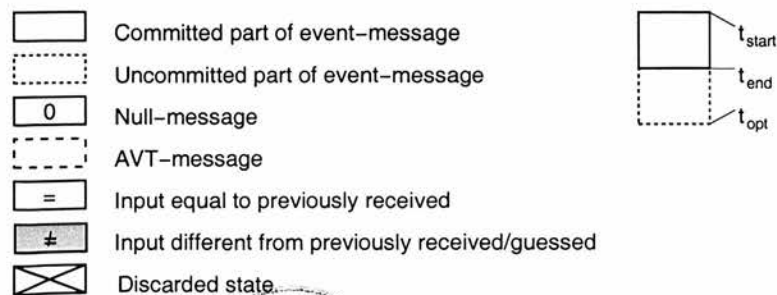
$\min\{t_{end}^{in}\}$ – denotes the minimum t_{end} of all inputs to an LP

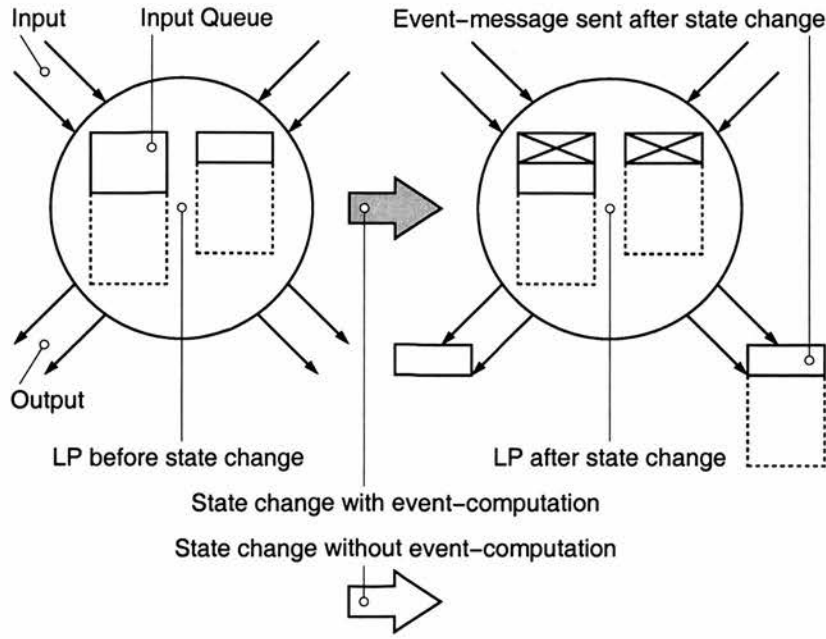
$\min^{LVT}\{t_{end}^{in}\}$ – denotes the minimum t_{end} of all inputs to an LP from predecessors outside the VT-area.

$\min^{AVT}\{t_{end}^{in}\}$ – denotes the minimum t_{end} of all inputs to an LP from predecessors inside the VT-area.

Predecessor (Successor) of an LP, A , is any other LP from which A receives its input (to which A sends its output).

Graphic Symbols: The height of a box corresponds to the interval between t_{start} and t_{end} (or t_{opt}).





4.2.2 The LVT-node

The LVT nodes synchronise conservatively. The node is ready to progress its virtual time once all the end-times of the input events are greater than its LCVT, i.e., $\min\{t_{end}^{in}\} > LCVT$.

4.2.2.1 LVT-node Computing

Once an event has been computed, an LVT-node sends the following event-message [change event to event-message throughout] to its successors (Figure 4.2):

Simulated version:

$$(t_{start} = LCVT + \delta, t_{end} = \min\{t_{end}^{in}\} + \delta, t_{opt} = t_{end}, f(input)) \quad (4.1)$$

Implemented version:

$$\begin{aligned} (t_{start} = LCVT + \delta, t_{end} = \max\{t_{start} + 1, \min\{t_{end}^{in}\} + 1\}, \\ t_{opt} = t_{end}, data = f(input)) \end{aligned} \quad (4.2)$$

Its LCVT is advanced to the maximum of the values: $\min\{t_{end}^{in}\}$ and t_{start} , of the event-message which was sent:

$$LCVT := \max\{t_{start}, \min\{t_{end}^{in}\}\} \quad (4.3)$$

Any state prior to the LCVT is discarded.

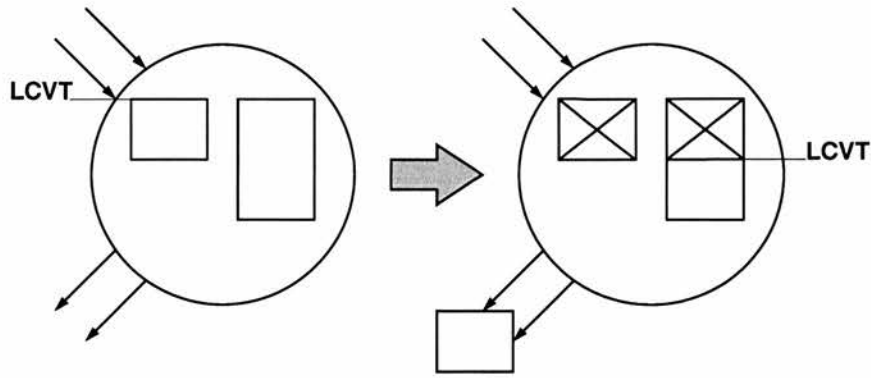


Figure 4.2: LVT-node in conservative mode

4.2.2.2 LVT-node Processing Null-messages

If the input to an LVT-node consists of events which either have the same value as the previously received one or are null-messages, then an LVT-node does not compute a new output event and sends the following null-message to all successors (Figure 4.3):

$$(t_{start} = LCVT + 1, t_{end} = \min\{t_{end}^{in}\} + 1, t_{opt} = t_{end}, null) \quad (4.4)$$

Its LCVT is advanced in the following manner:

$$LCVT := \min\{t_{end}^{in}\} \quad (4.5)$$

Any state prior to the LCVT is discarded.

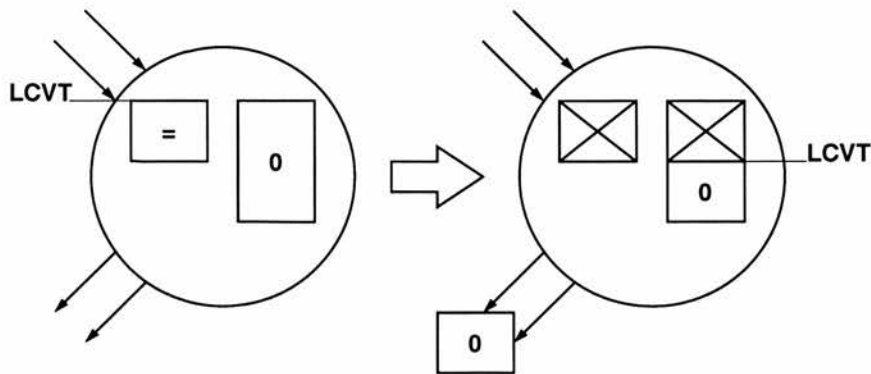


Figure 4.3: LVT-node in conservative mode

4.2.3 The GVT-node

A multitude of cases exist that determine a GVT-node's behaviour. These are listed and the resulting behaviour is described in detail in this section.

A GVT-node is ready to progress if at least one of its input-queues contains an unprocessed event, i.e. $\min\{t_{opt}^{in}\} > LOVT$. If committed inputs exist in all of the input queues, i.e. $\min\{t_{end}^{in}\} > LCVT$, a GVT node computes conservatively, whereas optimistically, otherwise.

4.2.3.1 GVT-node Computing on Committed Input

When progressing conservatively (Figure 4.4), the following event-message is sent to the node's successors within the VT-area:

Simulated Version:

$$(t_{start} = LCVT + \delta, t_{end} = \min\{t_{end}^{in}\} + \delta, t_{opt} = \min\{t_{opt}^{in}\} + \delta, f(input)) \quad (4.6)$$

Implemented Version:

$$\begin{aligned} (t_{start} = LCVT + \delta, t_{end} = \max\{t_{start} + 1, \min\{t_{end}^{in}\} + 1\}, \\ t_{opt} = \max\{t_{end}, \min\{t_{opt}^{in}\} + 1\}, f(input)) \end{aligned} \quad (4.7)$$

And, the following event-message is sent to successors outside the VT-area.

Simulated Version:

$$(t_{start} = LCVT + \delta, t_{end} = \min\{t_{end}^{in}\} + \delta, t_{opt} = t_{end}, f(input)) \quad (4.8)$$

Implemented Version:

$$(t_{start} = LCVT + \delta, t_{end} = \max\{t_{start} + 1, \min\{t_{end}^{in}\} + 1\}, t_{opt} = t_{end}, f(input)) \quad (4.9)$$

LCVT is advanced as follows:

$$LCVT := \max\{t_{start}, \min\{t_{end}^{in}\}\} \quad (4.10)$$

and, any state prior to the LCVT is discarded. Finally, an AVT-message, $(LCVT, LOVT)$, is sent to the AVT-keeper.

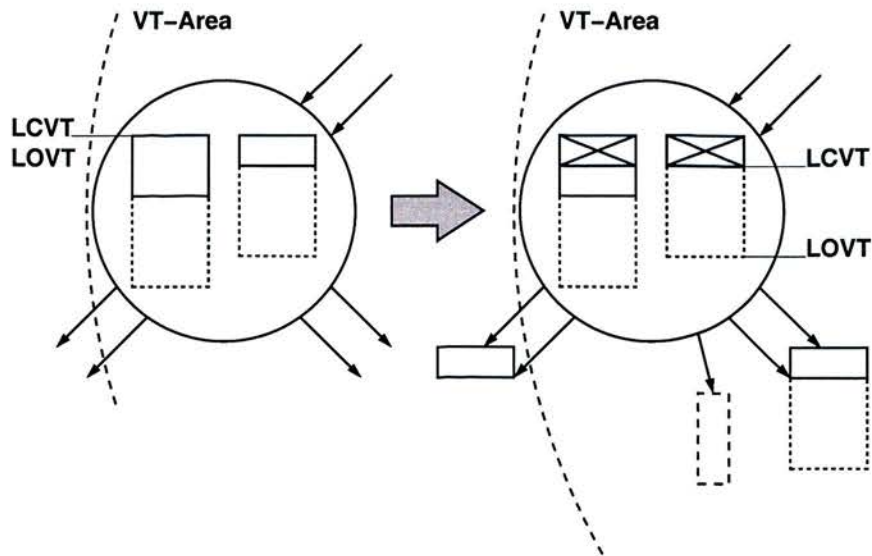


Figure 4.4: Computing conservatively in a GVT-node

4.2.3.2 GVT-node Computing on Uncommitted Input

When progressing optimistically (Figure 4.5), the following event-message is sent to the node's successors within the VT-area:

Simulated Version:

$$(t_{start} = LCVT + \delta, t_{end} = \min\{t_{end}^{in}\} + \delta, t_{opt} = \min\{t_{opt}^{in}\} + \delta, f(input)) \quad (4.11)$$

Implemented Version:

$$\begin{aligned} (t_{start} = LCVT + \delta, t_{end} = \max\{t_{start} + 1, \min\{t_{end}^{in}\} + 1\}, \\ t_{opt} = \max\{t_{end}, \min\{t_{opt}^{in}\} + 1\}, f(input)) \end{aligned} \quad (4.12)$$

No event-message is sent to successors outside the VT-area, and the LOVT is progressed as follows:

$$LOVT := \max\{t_{start}, \min\{t_{opt}^{in}\}\}, \quad (4.13)$$

and, an AVT-message, $(LCVT, LOVT)$, is sent to the AVT-keeper.

4.2.3.3 Rollback and Conservative Re-evaluation in a GVT-node

A rollback is caused should the input be different from the one received previously. The LOVT is reset to the value, t_{start} , of the "offending" event. If committed input is

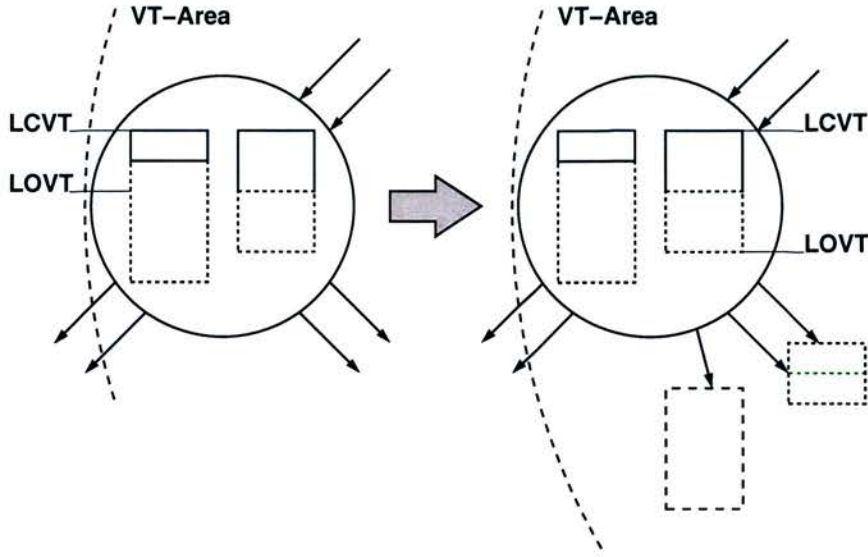


Figure 4.5: Computing optimistically in a GVT-node

available at the time of the rollback, then the GVT-node progresses conservatively as described in this section; it progresses optimistically otherwise (see Section 4.2.3.4). The output event for LOVT is recomputed, and the result compared to the previous one which was based on speculated value.

The following two cases arise:

1. The recomputed result remains unchanged (Figure 4.6), then no event-message is sent to successors inside the VT-area. The following event-message is sent to the successors outside the VT-area, if $\min\{t_{end}^{in}\} > LCVT$:

Simulated Version:

$$(t_{start} = LCVT + \delta, t_{end} = \min\{t_{end}^{in}\} + \delta, t_{opt} = t_{end}, f(input)) \quad (4.14)$$

Implemented Version:

$$\begin{aligned} (t_{start} = LCVT + \delta, t_{end} = \max\{t_{start} + 1, \min\{t_{end}^{in}\} + 1\}, \\ t_{opt} = t_{end}, f(input)) \end{aligned} \quad (4.15)$$

LCVT and LOVT are progressed in the following manner:

$$\begin{aligned} LCVT &:= \max\{LCVT + \delta, \min\{t_{end}^{in}\}\}, \\ LOVT &:= \max\{t_{start}, \min\{t_{opt}^{in}\}\}, \end{aligned} \quad (4.16)$$

and, an AVT-message, $(LCVT, LOVT)$, is sent to the AVT-keeper and all the states before the LCVT are discarded.

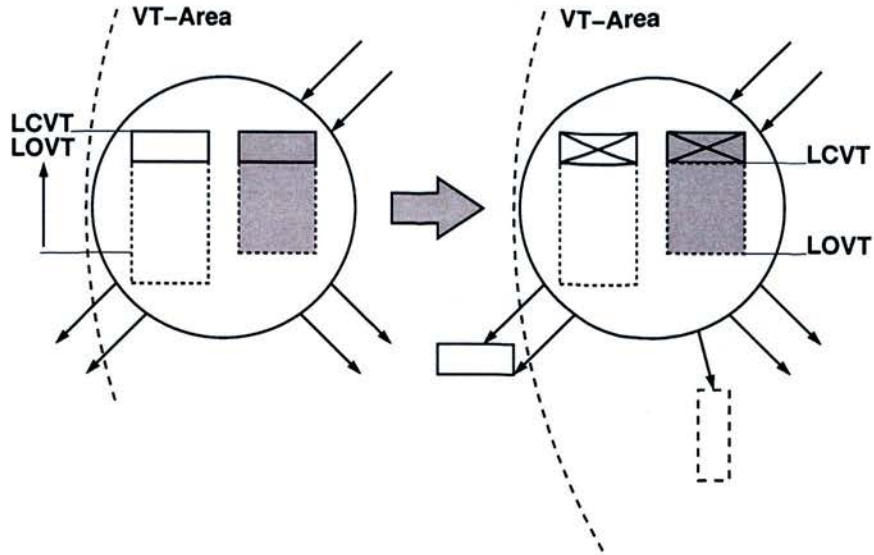


Figure 4.6: Rollback without change in output and conservative progress in a GVT-node

2. Should the result differ from the one propagated (Figure 4.7), then the correct result is sent to the GVT-node's successors within the VT-area, in the following event-message:

Simulated Version:

$$(t_{start} = LCVT + \delta, t_{end} = \min\{t_{end}^{in}\} + \delta, t_{opt} = \min\{t_{opt}^{in}\} + \delta, f(input)) \quad (4.17)$$

Implemented Version:

$$\begin{aligned} (t_{start} = LCVT + \delta, t_{end} = \max\{t_{start}, \min\{t_{end}^{in}\}\} + 1, \\ t_{opt} = \max\{t_{end}, \min\{t_{opt}^{in}\} + 1\}, f(input)) \end{aligned} \quad (4.18)$$

The following event-message is sent to the successors outside the VT-area:

Simulated Version:

$$(t_{start} = LCVT + \delta, t_{end} = \min\{t_{end}^{in}\} + \delta, t_{opt} = t_{end}, f(input)) \quad (4.19)$$

Implemented Version:

$$\begin{aligned} (t_{start} = LCVT + \delta, t_{end} = \max\{t_{start}, \min\{t_{end}^{in}\}\} + 1, \\ t_{opt} = t_{end}, f(input)) \end{aligned} \quad (4.20)$$

The LCVT and LOVT are advanced as follows:

$$\begin{aligned} LCVT &:= \max\{LCVT + \delta, \min\{t_{end}^{in}\}\}, \\ LOVT &:= \max\{t_{start}, \min\{t_{opt}^{in}\}\} \end{aligned} \quad (4.21)$$

Any state before LCVT is now discarded and an AVT-message, $(LCVT, LOVT)$, is sent to the AVT-keeper.

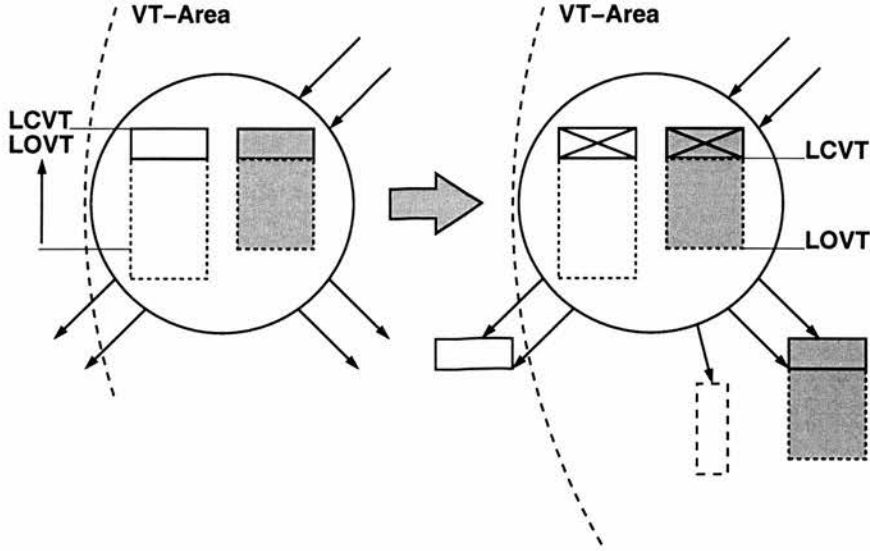


Figure 4.7: Rollback with change in output and conservative progress in a GVT-node

4.2.3.4 Rollback and Optimistic Re-evaluation in a GVT-node

When rollback occurs, the LOVT is reset to t_{start} of the “offending” event. The output event for LOVT is recomputed, and the result compared to the previously one based on speculated value.

The following two cases arise:

1. If the recomputed result is unchanged (Figure 4.8), then no event-message is sent to successors, and the LOVT is progressed as follows:

$$LOVT := \max\{t_{start}, \min\{t_{opt}^{in}\}\} \quad (4.22)$$

and, an AVT-message, $(LCVT, LOVT)$, is sent to the AVT-keeper.

2. If the result is different from the one propagated (Figure 4.9), then the correct one is sent to the GVT-node’s successors in the VT-area, in the following event-message:

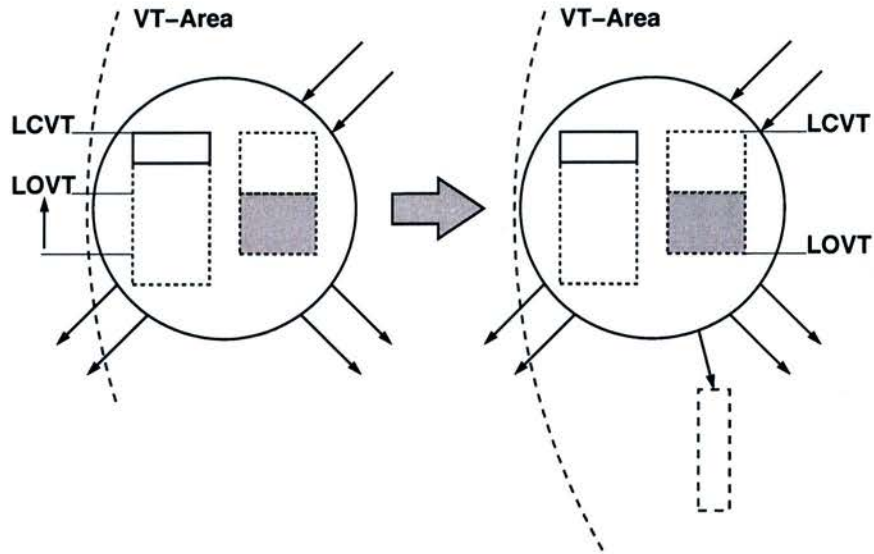


Figure 4.8: Rollback without change in output and optimistic progress in a GVT-node

Simulated Version:

$$(t_{start} = LOVT + \delta, t_{end} = t_{start}, t_{opt} = \min\{t_{opt}^{in}\} + \delta, f(input)) \quad (4.23)$$

Implemented Version:

$$(t_{start} = LOVT + \delta, t_{end} = t_{start}, t_{opt} = \max\{t_{end}, \min\{t_{opt}^{in}\}\} + 1, f(input)) \quad (4.24)$$

The LOVT is incremented as shown below:

$$LOVT := \max\{t_{start}, \min\{t_{opt}^{in}\}\} \quad (4.25)$$

and, an AVT-message, $(LCVT, LOVT)$, is sent to the AVT-keeper.

4.2.3.5 Roll-forward in a GVT-node

Roll-forward was only introduced in the implemented version as an optimisation of the AVT algorithm and the following cases were considered.

4.2.3.6 Roll-forward in a GVT-node with Committed Input

When rolling forward conservatively, the GVT-node does not compute a new result and sends a null-message to its successors outside the VT-area, if $\min\{t_{end}^{in}\} \geq t_{end}^{out}$ (Figure 4.10):

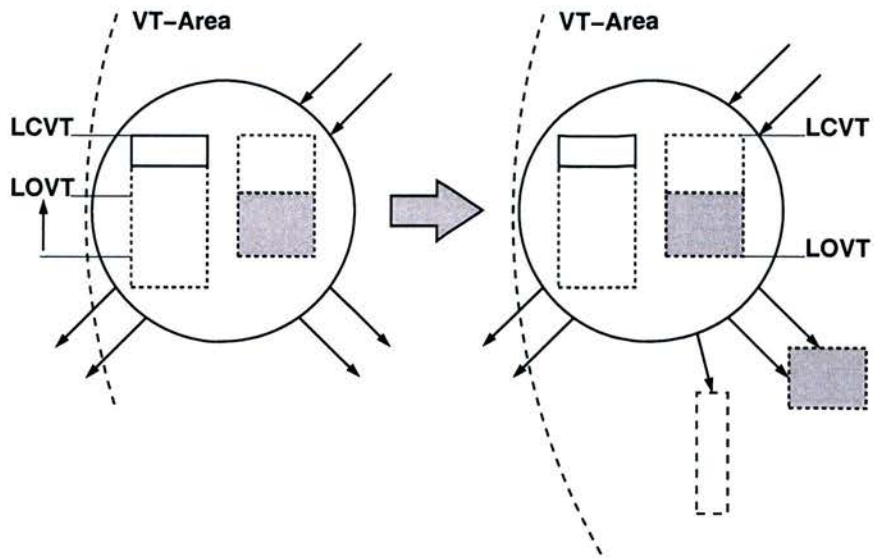


Figure 4.9: Rollback with change in output and optimistic progress in a GVT-node

Implemented Version:

$$(t_{start} = \max\{LCVT, t_{end}^{out}\}, t_{end} = \max\{t_{start}, \min\{t_{end}^{in}\}\} + 1, t_{opt} = t_{end}, null) \quad (4.26)$$

Irrespective of whether the null-message is sent, LCVT is advanced as shown:

$$LCVT := \max\{t_{start}, \min\{t_{end}^{in}\}\} \quad (4.27)$$

and any state prior to the LCVT is discarded.

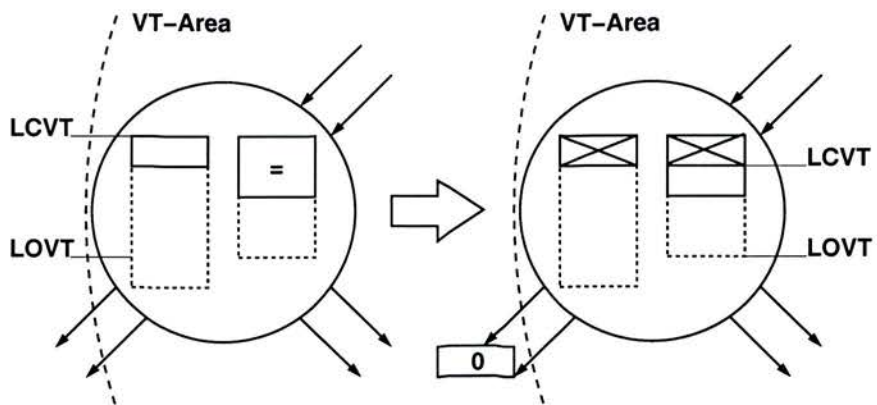


Figure 4.10: Rolling forward conservatively in a GVT-node

4.2.3.7 Roll-forward in a GVT-node without Committed Input

When rolling forward optimistically, neither a new result is computed nor is an event-message sent to successors. LOVT is progressed thus:

$$LOVT := \min\{t_{opt}^{in}\} \quad (4.28)$$

and, an AVT-message, $(LCVT, LOVT)$, is sent to the AVT-keeper.

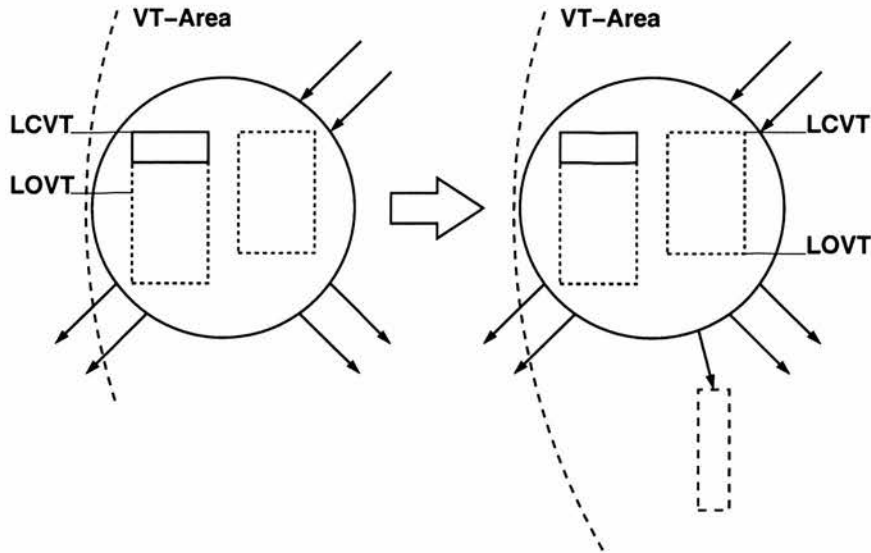


Figure 4.11: Rolling forward optimistically in a GVT-node

4.2.3.8 Progressing Virtual Time Based on AVT in a GVT-node

A GVT-node progresses its LCVT based on the AVT when the following conditions hold: the GVT-node has received an AVT-message, $(0, AVT)$, from the AVT-keeper and the updated AVT is greater than the minimum end-time of all the committed inputs, i.e., $AVT > \min(t_{end}^{in})$.

Simulated Version (Figure 4.12): The following event-message is sent to all its successors outside the VT-area:

$$(t_{start} = LCVT + \delta, t_{end} = AVT + \delta, t_{opt} = t_{end}, f(input)) \quad (4.29)$$

Implemented Version (Figure 4.13): All output events, based on input events whose

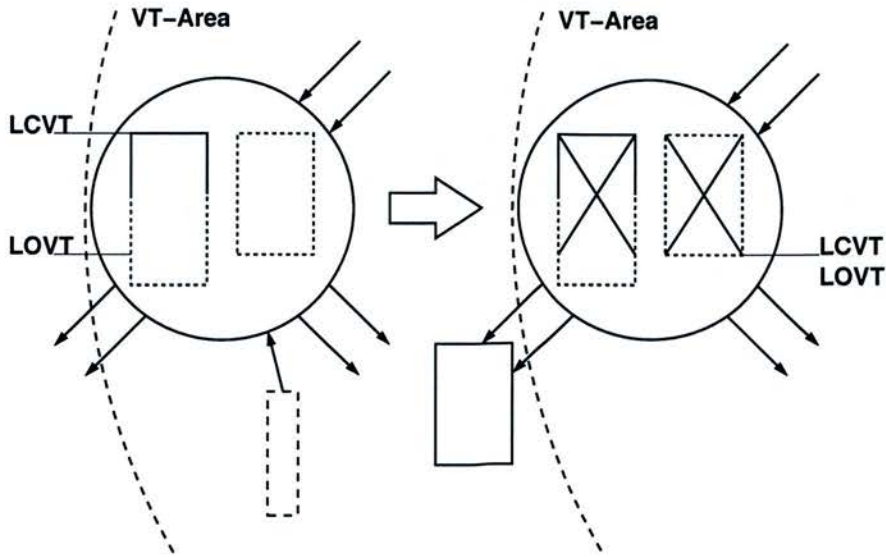


Figure 4.12: GVT-node progressing based on AVT (simulated)

$t_{start} < AVT$, are sent to its successors outside the VT-area:

$$\begin{aligned}
 & (t_{1_{start}} = LCVT + \delta, t_{1_{end}} = t_{2_{start}}, t_{1_{opt}} = t_{1_{end}}, f(input)) \\
 & (t_{2_{start}} = t_{1_{end}}, t_{2_{end}} = t_{3_{start}}, t_{2_{opt}} = t_{2_{end}}, f(input)) \\
 & \quad \vdots \\
 & (t_{N_{start}} = t_{N-1_{end}}, t_{N_{end}} = \max\{t_{N_{start}}, AVT\} + 1, t_{N_{opt}} = t_{N_{end}}, f(input))
 \end{aligned} \tag{4.30}$$

LCVT is advanced as follows:

$$LCVT := \max\{t_{N_{start}}, AVT\} \tag{4.31}$$

and all the states before the LCVT are discarded.

4.2.4 The Hybrid-node

Hybrid nodes appear as GVT/optimistic nodes to those inside the VT-area, and as LVT/conservative nodes to those outside. When the committed inputs from outwith the VT-area are available, the hybrid node guesses the values for the other inputs from within the VT-area, evaluates the function based on all the inputs (both guessed and committed), and fires optimistically, but only to its neighbours within the VT-area.

The behaviour of hybrid-node is similar to that of the GVT-node, and only the cases which differ are described. The main difference is that a hybrid node will never speculate on events it is waiting to receive from outside the VT-area.

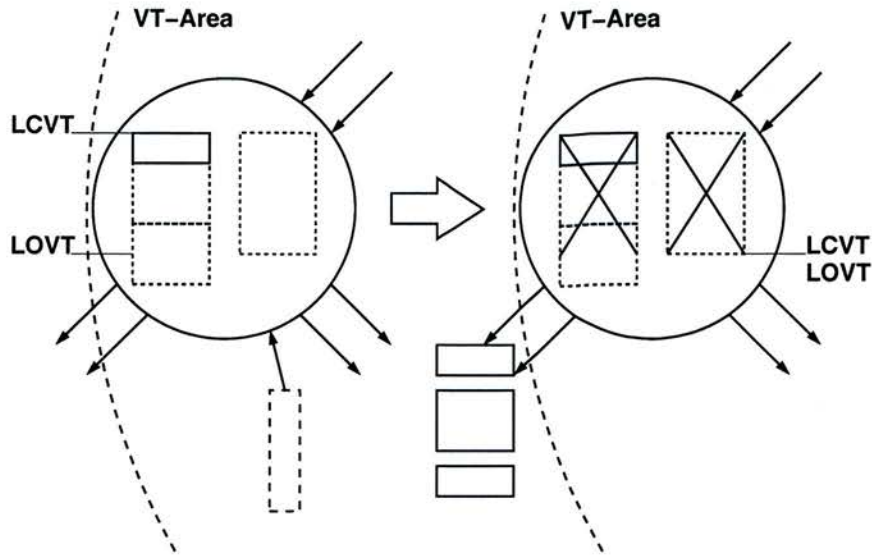


Figure 4.13: GVT-node progressing based on AVT (implemented)

A hybrid node is ready to speculate if all the input-queues to the LVT part of the hybrid-node contain events, and the minimum end-time of the LVT-inputs is greater than the minimum optimistic-end-time of the AVT-inputs, i.e.

$$\min^{LVT} \{t_{end}^{in}\} > \min^{AVT} \{t_{end}^{in}\} \wedge \min^{LVT} \{t_{end}^{in}\} > LOVT \quad (4.32)$$

and, the LP assumes that the previous values for the AVT-inputs hold and fires optimistically.

4.2.4.1 Hybrid-node Speculating Based on Committed Input

If all inputs to the hybrid node contain unprocessed, committed events, then the following event-message is generated, stored in the LPs output-queue, and sent to the LP's successors in the VT-area (Figure 4.14):

Simulated Version:

$$(t_{start} = LCVT + \delta, t_{end} = \min\{t_{end}^{in}\} + \delta, t_{opt} = \min^{LVT} \{t_{end}^{in}\} + \delta, f(input)) \quad (4.33)$$

Implemented Version:

$$\begin{aligned} t_{start} &= LCVT + \delta, t_{end} = \max\{t_{start}, \min\{t_{end}^{in}\}\} + 1, \\ t_{opt} &= \max\{t_{end}, \min\{t_{opt}^{in}\} + 1\}, f(input) \end{aligned} \quad (4.34)$$

The LOVT is advanced as follows:

$$LOVT = \max\{t_{start}, \min\{t_{opt}^{in}\}\} \quad (4.35)$$

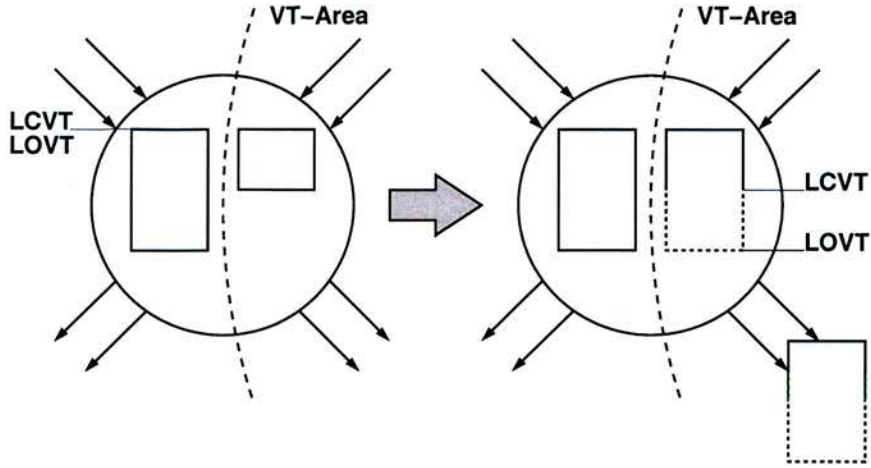


Figure 4.14: Speculation on committed input in a hybrid-node

4.2.4.2 Hybrid-node Speculating Based on Uncommitted Input

If the input-queues to the hybrid-node do not all contain unprocessed, committed events, then the following event-message is generated and sent to the LP's successors in the VT-area (Figure 4.14):

Simulated Version:

$$(t_{start} = LOVT + \delta, t_{end} = t_{start}, t_{opt} = \min\{t_{opt}^{in}\} + \delta, f(input)) \quad (4.36)$$

Implemented Version:

$$\begin{aligned} (t_{start} = \max\{LOVT + \delta, \min\{t_{end}^{in}\}\}, t_{end} = t_{start}, \\ t_{opt} = \max\{t_{end}, \min\{t_{opt}^{in}\} + 1\}, f(input)) \end{aligned} \quad (4.37)$$

The LOVT is advanced in the following manner:

$$LOVT = \max\{t_{start}, \min\{t_{opt}^{in}\}\} \quad (4.38)$$

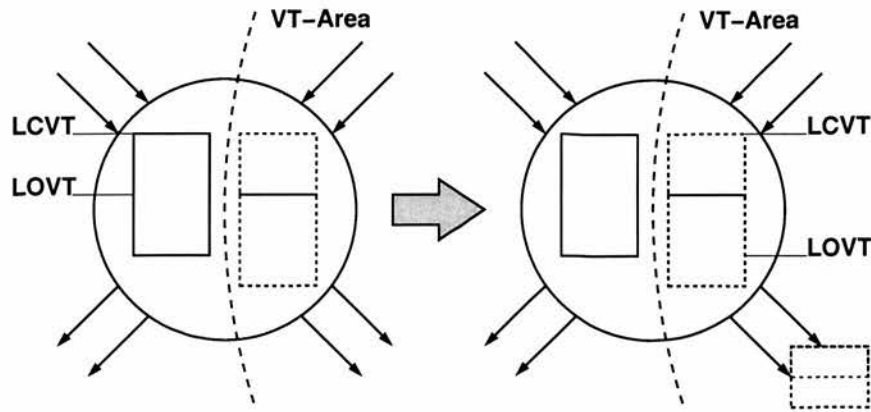


Figure 4.15: Speculation on uncommitted input in a hybrid-node

4.2.5 The AVT-keeper

The AVT-keeper computes the AVT as the minimum of all the estimates received by the GVT- and hybrid-nodes in the VT-area, and the minimum, committed time-stamp of any event-message in transit within the VT-area. To keep track of event-messages in transit, special information is attached to event-messages and AVT-messages (Section 4.2.5.1 below).

If AVT can be advanced, then the AVT-keeper sends the updated AVT to all the nodes in the VT-area. On receipt of the AVT update, the nodes commit all events up to this point in time and send pending events to neighbours outwith the VT-area as described in Section 4.2.3.8. The nodes then bring forward their LCVT to the AVT, and discard all state information before the AVT.

4.2.5.1 Send and Receive Notices

Send and receive notices are attached to event- and AVT-messages to keep track of messages in transit. They are of the form, $(type, ID, t_{start}, count)$, where:

$type$ is an element of $\{Sent, Received\}$, denoting whether an event-message has been sent and is in transit, or whether it has been received at its destination, respectively.

ID is the unique identifier of the sender of the event-message.

t_{start} is the start-time of the event

$count$ is the number of events with ID and t_{start} which are sent or received.

4.2.5.2 Operation

Every event-message, $(t_{start}, t_{end}, t_{opt}, data)$, sent from an LP A to an LP B inside a VT-area is annotated with a send notice, $(Sent, A, t_{start}, 1)$. When the event-message is sent, the AVT-message sent at that time is annotated with a send notice $(Sent, A, t_{start}, N)$, where N is the number of LPs the event-message has been sent to.

When an LP receives an event-message annotated with a send notice, $(Sent, A, t_{start}, 1)$, it stores the send notice and attaches to the next AVT-message the receive notice $(Received, A, t_{start}, 1)$. More than one receive notice can be attached to an AVT-message.

When the AVT-keeper receives an AVT-message,

$$((LCVT, LOVT)(Sent, A, t_{start}, N)),$$

(where N is the number of event-messages, as described previously) from an LP within its assigned VT-area, the received LCVT, LOVT, as well as the send notice are stored and the AVT is re-computed as the minimum of the received LOVTs of all the LPs in the VT-area and t_{start} of all send notices stored.

When the AVT-keeper receives an AVT-message,

$$((LCVT, LOVT)(Received, A, t_{start}, 1) \dots),$$

then the received LCVT and LOVT are stored. The *count* entry of every send notice, $(Sent, A, t_{start}, N)$, for which a corresponding receive notice had been attached, is decremented by one. Any send notice whose *count* reaches 0 is removed. AVT is re-computed as the minimum of the received LOVTs of all the LPs in the VT-area, and t_{start} of all remaining send notices which are stored. If the new AVT is greater than the previous one, then the AVT-keeper notifies all the LPs in the VT-area whose LCVT is less than the AVT.

The AVT-algorithm was first simulated in order to understand better its behaviour and performance in practice, and the results are presented next.

Chapter 5

AVT-Algorithm – Simulation

The simulation results described in this chapter were presented at the conference Distributed Simulation and Real-time Applications (DS-RT 2001) held in August 2001 in Cincinnati, Ohio [ArvindS2001]. The full text of the paper published in the proceedings of the conference is included in Appendix A.

5.1 The Simulation Environment

The behaviour of the AVT-algorithm was modelled in C++ and simulated in a sequential event-driven simulator (also written in C++). The simulator also models a distributed system consisting of processors connected by a network. All significant operations in the distributed system, such as computing results and sending event-messages, are assigned costs.

The network model assumes a fixed cost and a minimum fixed delay for each message. The *bandwidth* of the network determines the number of messages that can be transmitted concurrently over the network at any time. Messages are delayed until the network bandwidth becomes available. Thus the network model takes congestion into account but not the relative sizes of messages, or the effects of thrashing.

We distinguish between two notions of virtual time: the virtual time according to the AVT-algorithm is called *virtual time*, and the virtual time according to the simulated distributed system on which the AVT-algorithm executes is referred to as the *simulated real time*.

The simulation was run under many different parameter sets, including:

Number of Processors The number of virtual processors used to run the simulation.

Communication Bandwidth This determines the number of messages the network can transmit at any time. Bandwidth can be set to unlimited, in which case every message is delayed by a fixed amount of *simulated real time*.

Message Delay The amount of *simulated real time* required to send an event-message or an AVT-message from one processor to another.

Event Processing Delay The amount of *simulated real time* needed to compute a result in an LP.

AVT Processing Delay The amount of *simulated real time* needed to process an AVT computation in the AVT-manager.

Amount of *simulated real time* The number of units of *simulated real time* for which the distributed system is simulated for each parameter set.

Mode One parameter from the set {LVT, AVT, GVT} is selected. In LVT-mode, every LP behaves as an LVT-node. In GVT-mode, all LPs except input-nodes behave as a GVT-node and are members of one global VT-area. In AVT-mode, all LPs in the acyclic area of the simulation model behave as LVT-nodes, and the LPs in the cyclic area are members of a VT-area encompassing the cyclic area and behave as GVT- or hybrid-nodes, respectively.

Input Interval The amount of virtual time between the arrival of any two input events to the model.

Delay The virtual time between receiving an event in an LP and producing an output event in this LP.

Output Change Probability A value between 0 and 1 which gives the probability that the output of an LP changes if one of the inputs changes. If the probability for an output change after one input change is P , then the resulting probability for a change in output given that N inputs have changed, is $1 - (1 - P)^N$.

5.2 Results

The benchmark model under simulation contains cyclic (2 LPs) and acyclic parts (3 LPs), and one LP generating input to the model (Figure 5.1).

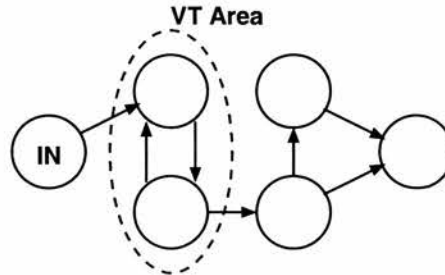


Figure 5.1: Benchmark model graph

The model was run under all the different combinations of the following parameters:

Number of Processors {1, 2, 5}. In the two-processor case, the cyclic part was mapped to one processor and the acyclic part to the other. The input node was always allocated to processor 1.

Communication Bandwidth 1

Message Delay {1, 3, 10, 32, 100} units of *simulated real time*

Event Processing Delay 100 units of *simulated real time*

AVT Processing Delay 0 units of *simulated real time*

Amount of Simulated Real Time 10000 units

Mode {LVT, AVT, GVT}

Input Interval 100 units of virtual time.

Delay {1, 3, 10, 32, 100} units of virtual time.

Output Change Probability {0%, 25%, 50%, 75%, 100%}

The performance was measured in terms of progress of virtual time per unit of *simulated real time*. The graph in Figure 5.2 compares the progress in virtual time for the three time-keeping schemes (y-axis), while changing the simulation parameters along four dimensions: *Delay*, *Message Delay*, *Number of Processors*, and *Output Change Probability* (Change Prob. in the figure).

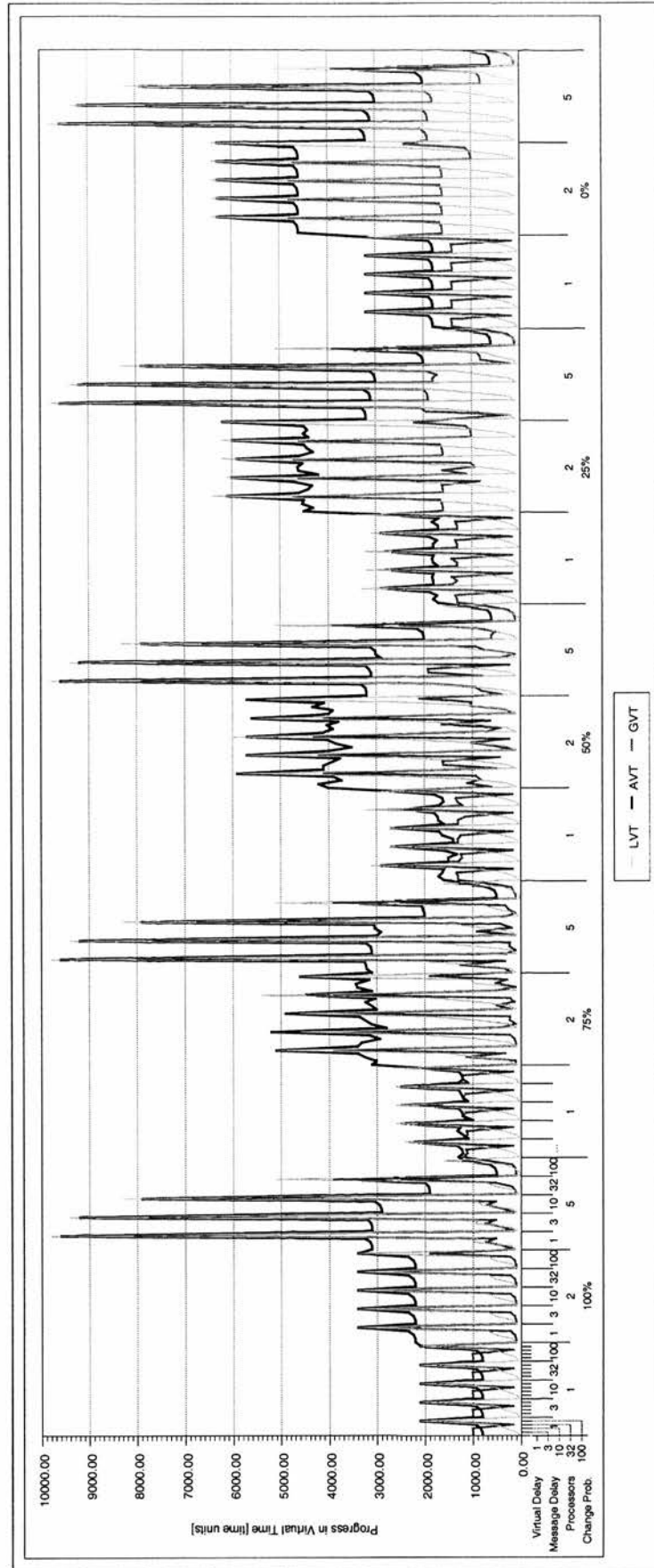


Figure 5.2: Progress of Virtual Time for 10000 units of Simulated Real Time

Whereas the graph in Figure 5.2, at first glance, displays an overwhelming amount of information, it also allows one to compare patterns over the entire data set. Consider the lefthand column of graphs (*Change Prob.* = 100%) which on comparison of the three sets for 1, 2, 5 processors shows the speedup attained by using additional processors. In the third set of this column (5 processors) one can see that, as the *Message Delay* increases, the performance degrades; however, this effect does not occur in the middle set (2 processors) as the communication is mostly local to each processor. Finally, each graph allows the comparison of LVT, AVT, and GVT performances for a range of *Virtual Delay* values. Comparing sets and columns with matching parameters across the whole graph, results in an overview of the behaviour of the AVT algorithm. Details of Figure 5.2 are shown in the graphs in Figures 5.3–5.6.

As the value of the *Delay* is incremented, the progress of the virtual time improves, and in the extreme, to a spike, when the *Delay* is the same as the *Input Interval*. In this case the existing look-ahead reaches up to the arrival of the next event, resulting in no event fragmentation.

We also observe that in the case of mapping to two processors, the performance of the AVT-algorithm is more stable over changes in the parameters, compared to the GVT time-keeping scheme, which is hit by network delays. We note that the LVT time-keeping mechanism only performs well in the cases of good look-ahead, i.e. large *Delay*. In the mapping to five processors, it is evident that the AVT algorithm degrades more gracefully with increases in network delays.

The AVT-algorithm maintains this superiority over the other two time-keeping mechanisms for different *Output Change Probability* values.

Figure 5.3 details the influence of the *Delay* on the three different time-keeping schemes. While LVT performance rises in proportion to the *Delay*, AVT and GVT performances are independent, except for large values of *Delay*. AVT outperforms GVT and also LVT, except for *Delay* = 100.

Figure 5.4 shows that as the *Message Delay* increases, the performance decreases, however, the AVT-algorithm's performance remains higher compared to the other two time-keeping mechanisms, as it reduces the number of messages required for synchronisation.

The speedup of the three time-keeping mechanisms is depicted in Figure 5.5. As the *Output Change Probability* is zero, only the LP connected to the input (IN) performs

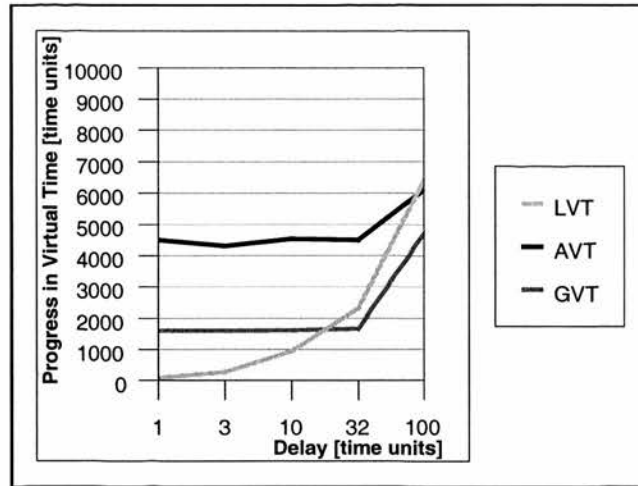


Figure 5.3: Detail of Figure 5.2: Progress for $Delay = 1, 3, 10, 32, 100$; $Message Delay = 1$; $Number of Processors = 2$; $Output Change Probability = 75\%$

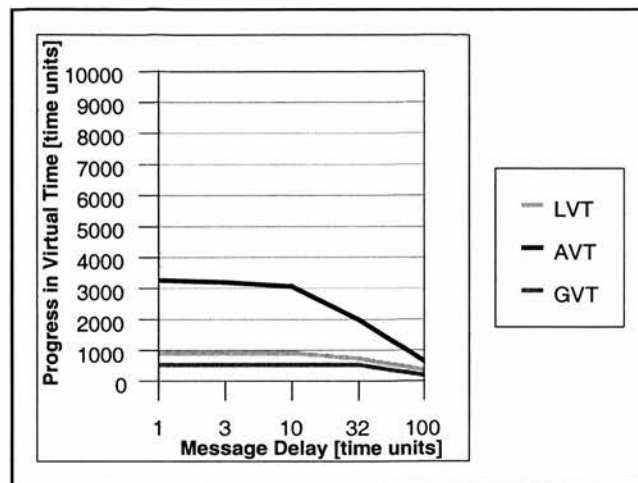


Figure 5.4: Detail of Figure 5.2: Progress for $Delay = 32$; $Message Delay = 1, 3, 10, 32, 100$; $Number of Processors = 5$; $Output Change Probability = 0\%$

event-computations. Due to the null-message overhead, LVT performance increases only slightly when more processors are used. The GVT time-keeping mechanism cannot attain any speedup as it needlessly uses global time in the acyclic part of the simulation model. The AVT-algorithm sends fewer synchronisation messages which results in the best performance of the three time-keeping mechanisms.

Figure 5.6 shows that as the *Output Change Probability* decreases, the performance of all three time-keeping mechanisms increases. However, due to the AVT-algorithm's

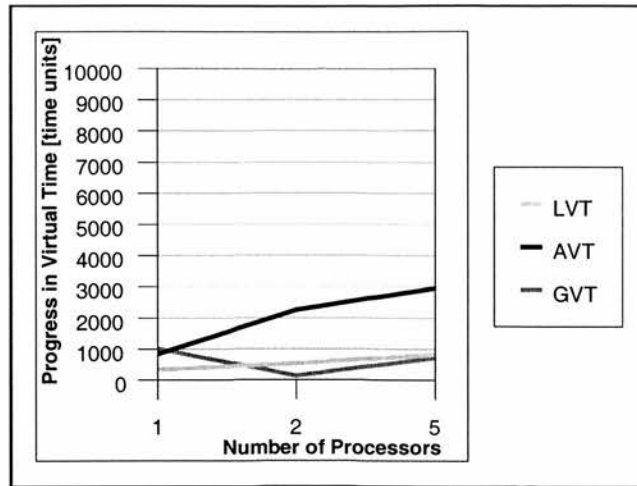


Figure 5.5: Detail of Figure 5.2: Progress for *Delay* = 10; *Message Delay* = 10; *Number of Processors* = 1, 2, 5; *Output Change Probability* = 0%

lower messaging overhead, it outperforms the LVT and GVT time-keeping mechanisms.

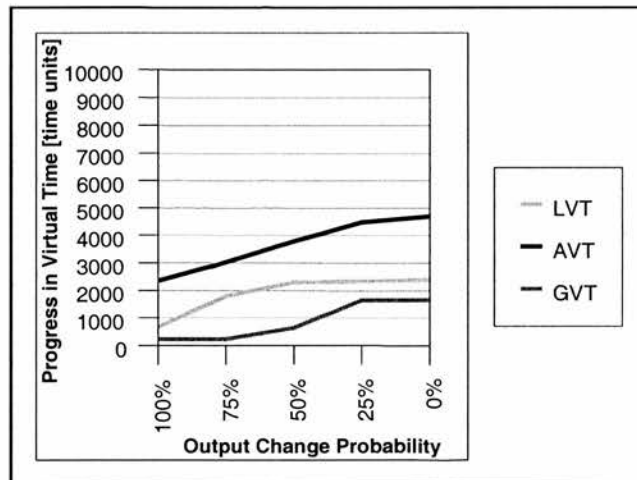


Figure 5.6: Detail of Figure 5.2: Progress for *Delay* = 32; *Message Delay* = 32; *Number of Processors* = 2; *Output Change Probability* = 0%, 25%, 50%, 75%, 100%

The results demonstrate that the AVT-algorithm performs better or just as well as the other two mechanisms for a simulation model containing a mixture of cyclic and acyclic parts.

However, there is a caveat: the simulation of the AVT-algorithm does not take into

account the computational overheads of computing AVT and state-saving and rollback. The rationale for this omission is the assumption that computation is cheap and fast, and communication slow and expensive by comparison. The evaluation of the AVT-algorithm employing an actual implementation, addresses this topic (Chapter 7).

The results show the efficacy of the AVT-algorithm and the next chapter describes its actual implementation on a real distributed system.

Chapter 6

AVT-Algorithm – Implementation

The AVT-algorithm has been implemented in C++ as a simulation-specific library of parameterised classes. The simulation library with the AVT-algorithm at its heart was named AVTSIM for Area Virtual Time Simulator. Distributed computing for AVTSIM was supported through the use of the MPI (Message Passing Interface) communication library [<http://www.lam-mpi.org/>]. This combination of computing language and communication library was chosen for portability reasons: at the time of the inception of this work this was the most widely available platform. Only a small subset of the MPI message transmission functionality was used: the asynchronous MPI primitives `MPI_Iprobe` and `MPI_Send`, and synchronous `MPI_Recv`.

AVTSIM can be compiled using the GNU C++ compiler `g++` on Linux, Solaris, and Microsoft Windows, the Sun C-compiler on Solaris (without library support), and Microsoft Visual C++ on Windows (without MPI-support).

Simulation models are written in C++ with the support of the AVTSIM library. An object-oriented language was chosen as it provides the most natural way of describing real-world entities, as state is encapsulated in objects and method invocation is similar to message passing.

Simulation models in AVTSIM are described in terms of LPs, inputs, and outputs, implemented by the classes `Entity`, `InPort`, and `OutPort`, respectively. Entities may possess an arbitrary number of inputs and outputs and while fan-out from outputs is permitted, fan-in to inputs is not. Entities react when they perceive a *change* in one or more input values, whereas an event-message carrying the same value seen before is not considered to be an event. When an LP reacts at virtual time t it may generate

events, that may be presented after different delays for different outputs. Or, an LP may not generate any output for the activation at virtual time t .

All simulated time in AVTSIM is measured in virtual time units (VTU) which are not a fixed representation of any real world time unit, e.g. seconds. The meaning of VTUs depends on the application and is determined by the programmer of the simulation model. Virtual time is represented in AVTSIM by default by a 64-bit integer. However, the module defining the data type for virtual time can be exchanged for any other data type or class that supports the set of operations and relations required by AVTSIM.

The implementation of the AVT-algorithm serves two purposes:

- the evaluation of the AVT-algorithm under real-world conditions while retaining the ability to simulate itself
- providing a general-purpose distributed simulator targeted at the simulation of asynchronous systems.

For the intended use as a general-purpose simulator, two simplifying assumptions made when simulating the AVT-algorithm (Chapter 5), were dropped. The delay of the event generation in an LP is no longer assumed to be fixed, i.e. an LP may present output after variable, or input dependent delays for each activation, and may present output after different delays for different outputs during one activation. In the simulated version the results computed in an LP were only dependent on the inputs to the LP. Whereas, in the implementation, the value of result-events may depend on the virtual time at which they were computed, i.e. the virtual time of the activation is considered an input parameter for the event computation. This means that a re-computation of an output must be scheduled whenever the new activation time differs from the activation time of the previous speculative evaluation, even if the inputs do not show any differences between those two points in virtual time.

AVTSIM has been tested by executing the test models (Section 7.3) under many different parameter sets. The output traces of the simulation-runs in LVT-mode on a uniprocessor were saved, and compared to about 5 million values from output traces of subsequent multiprocessor executions in AVT- and in GVT-mode.

A more detailed description of AVTSIM, models, entities, inputs, and outputs is provided next, along with an example of a complete small simulation at the end of this chapter.

6.1 AVTSIM Modelling Interface

To use the simulator one has to include a C++ header file containing all the required definitions of AVTSIM. This ensures that global objects of the simulator are instantiated and all required classes and functions of the simulator are available to the user's code.

6.1.1 Entities

All LPs (called entities) must be derived from the base class `Entity` and must implement the virtual function `evaluate ()`. Entities can have an arbitrary number of inputs and outputs which are implemented by the parameterised template classes `InPort` and `OutPort`, respectively (see Section 6.1.2).

The entity's function `evaluate ()` is called each time at least one input value to an LP (an object of class `Entity`) changes. The entity is then considered busy until the time-stamp, t_{start} , of the last output within the current call of `evaluate ()` sent, i.e. the entity's virtual time after `evaluate ()` is at least t_{start} . All value-changes that occur at inputs during the time the entity is busy are either:

- perceived at the virtual time at the end of `evaluate ()`,
- or, lost and only the last value-change during the busy-period is recognised, if more than one value-change occurs on an input while the entity is busy.

Whether any of the two conditions above should be considered to be an error, can be set individually for every input (see Summary of the `InPort` Class Interface in Section 6.1.2).

An entity which only possesses output-ports is called an input-entity, since it generates external input to the simulation, e.g. from a file. An input-entity's `evaluate ()` method is never called speculatively.

An entity which only possesses input-ports is called an output-entity. Its `evaluate ()` method is called whenever any one of its input-ports change value (just like for "normal" entities). However, in contrast to normal entities, the `evaluate ()` method of output-entities is never called speculatively.

Summary of the Entity Class Interface

Entity (const char* name) is used to construct an entity. The optional parameter `name` determines the name of the entity printed in error and warning messages, and debugging output. If the optional parameter is not set, then the name of the entity is the string representation of the entity's unique ID.

int id () Returns the unique ID of the entity.

const char * name () Returns a pointer to the name of the entity.

virtual void evaluate () This method must be overridden by sub-classes of entity. The `evaluate ()` method of an entity is called whenever there is a change in the values of the entity's input ports. The `evaluate ()` method must terminate, otherwise control is not passed back to the simulator's scheduler and no other entity in the simulator can ever become active.

int now () Returns the current virtual time of the entity.

void wait (int time) The entity waits for `time` virtual time units. During this time it is considered busy and is unable to react to changes in its inputs.

6.1.2 Communication Between Entities

The inputs and outputs to an entity are implemented as the parameterised template classes `InPort<Type>` and `OutPort<Type>`, with the transmitted data type `Type` as the parameter. This allows for type-checking of connections between inputs and outputs by the C++ compiler. Values of `Type` are transmitted between different physical processors, therefore pointers, or structures containing pointers, must not be used as the communication data-type, as pointers valid on one machine may not be pointing to valid data on a remote one.

Fan-in to inputs is not allowed, but outputs may have an arbitrary number of fan-out connections to inputs of the same communication data type.

In AVTSIM, an LP (an entity) will not react until presented with an input which is different from that previously perceived, i.e. an event is a *value-change*, and not the arrival of an event-message. To reflect this we introduce a pair of communication

primitives modelled, called *present* and *perceive*, which is modelled on the pair *send* and *receive*.

From within `evaluate ()`, the input values for the current virtual time can be obtained by calling the input's `perceive ()` function. The user's implementation of `evaluate ()` will then perform its computation and propagate results on the entity's outputs using `present (...)`. Only one output-value may be presented on each output for every evaluation. Sometimes it may be necessary to mark two identical values presented at an output at consecutive times as separate events. For this purpose a variant of *present* called `trigger (...)` is implemented.

Inputs and outputs are connected using the `connect (...)` primitive.

Summary of the InPort Class Interface

InPort <Type> (parent, initValue, mode, name) is the constructor for an InPort where:

Type is the transmitted data type.

Entity * parent is a pointer to the parent-entity, i.e. the entity the input belongs to.

const Type & initValue is an optional parameter giving the default value perceived at the input at the beginning of the simulation, before any event-message has been received. If the init-value is not set, AVTSIM sets it to 0, or whatever 0 represents in the communication data type.

const InPortMode mode is an optional parameter with a value of either one of `DisallowEventsWhenBusy`, `AllowSingleEventWhileBusy`, or `AllowMultipleEventsWhileBusy`, describing how AVTSIM should handle value-changes occurring at this input during the time an entity is busy.

DisallowEventsWhenBusy Do not allow any value-change to occur while the entity is busy. Print an error message and terminate the simulation if this condition is violated.

AllowSingleEventWhileBusy Allow one value change to occur while the entity is busy. This permits reaction to the value change after the busy period, i.e. the value-change is not lost, but the reaction is

delayed. An error message is printed and the simulation is terminated, if the above condition is violated.

AllowMultipleEventsWhileBusy Allow any number of value changes during the time an entity is busy. This may lead to some value-changes never to be perceived and to be lost.

The default is `DisallowEventsWhenBusy`.

const char * name is an optional parameter giving the name of the input printed in error and warning messages, and debugging output. The default is the string representation of the input's unique ID.

const PortId portId () Returns the unique ID of the input.

const char * name () Returns a pointer to the name of the input.

void connect (OutPort <Type> * outPort) Connects an input with an output of the same transmitted data type `Type`. If the input is already connected to another output, AVTSIM prints an error message and aborts the simulation.

bool triggered () Returns true, if the value perceived at the input has been produced using `trigger (...)`, instead of `present (...)`.

Type & perceive () Returns the current value perceived at the input.

Type & perceivePrev () Returns the value perceived at the input at the virtual time of the previous call of `evaluate()`.

void preserve () Sometimes it may be necessary to delay reacting to one particular input value while reacting to changes in other input values. To prevent the simulator from assuming that an input value-change has been processed when `evaluate()` finishes, `preserve()` can be used. It marks the value-change as unperceived in order to force AVTSIM to call `evaluate()` again.

void consume () On occasion an entity may not want to produce any output during a call of `evaluate()`, e.g. if the entity is waiting for another value-change at a different input before it can progress. In such cases `consume()` can be used, to tell AVTSIM not to activate this entity again on the value-change just perceived.

Summary of the OutPort Class Interface

OutPort (Entity * parent, const char * name) Constructs an output. `parent` is a pointer to the parent-entity. The optional parameter `name` determines the name of the entity printed in error and warning messages, and debugging output. The default is the string representation of the output's unique ID.

void connect (InPort <Type> * otherPort) Connects an output with an input of the same transmitted data type `Type`. If the input is already connected to another output, AVTSIM prints an error message and aborts the simulation.

void presentAfterDelay (value, delay) presents a value at the output after a delay.

const Type & value is the value to be presented, and

const VirtualTime delay is the delay in virtual time units between the virtual time `evaluate()` has been called and the time when the output-value is available.

void presentAfterDelayValidFor (value, delay, valid)

provides the same functionality as `presentAfterDelay(...)`. Additionally the parameter

const VirtualTime valid guarantees that the value will not change for `valid` VTUs after it has been presented.

void presentAtTime (value, time) presents a value at the output at a specified virtual time.

const VirtualTime time is the virtual time when the value is presented.

Note that `time` must be greater than the virtual time for which `evaluate()` has been called with the exception of input- and output-entities.

void presentAtTimeValidFor (value, time, valid) is the same as `presentAtTime(...)` except for the additional parameter

const VirtualTime valid which denotes that the value is guaranteed not to change for `valid` VTUs after it has been presented.

void trigger...(...) For each of the `present...(...)` functions above, a

`trigger...(...)` function is available which performs the same operation except that every output generated is marked as a new event regardless of whether it constitutes a value-change or not (cf. Section 6.1.2).

const Type & currentValue () returns the current output-value, i.e. it returns the value presented previously before any of the `present...(...)` or `trigger...(...)` functions have been called; otherwise, the value which has been just presented in a `present...(...)` or `trigger...(...)` function.

6.1.3 Models

A model is a collection of entities together with a few initialisation routines. Models must be derived from the base class `Model`. The initialisation part of the model is responsible for assigning entities to processors, for creating VT-areas and allocating the VT-area managers (implemented by class `VTArea`) to processors, and for assigning entities to VT-areas. The base class `Model` already provides all necessary initialisation for LVT and GVT-mode simulation.

By default, entities are allocated to processors in the following manner: Let E be the number of entities, P the number of processors, I the index of the current processor, initialised to 1.

1. Allocate $N = E \text{ DIV } P$ many entities on processor I .
2. Increment I by 1; decrement E by N , P by 1.
3. If $P > 0$ then 1.; else END.

LVT-mode simulations do not require any special initialisation. By default, GVT-mode simulations are initialised by creating a VT-area manager, allocating it on the first processor, and assigning all entities to the VT-area except for input- and output-entities.

For AVT-mode simulations the programmer must redefine `Model`'s class member function `init()`, since automatic determination and creation of VT-areas is not supported in the implementation.

The `model` class supports this by supplying the following member functions:

allocateEntityOnProcessor (entity, processorId) instructs AVT-SIM to allocate `entity` on the processor with index `processorId`, where

Entity * entity is a pointer to the entity and

int processorId the zero-based index of the processor.

allocateVTAreaOnProcessor (vtArea, processorId) allocates the VT-area manager vtArea on processor processorId, where

VTArea * vtArea is a pointer to an object of class VTArea and

int processorId is as for `allocateVTAreaOnProcessor(...)`

addEntityToVTArea (entity, vtArea) makes entity a member of vtArea, where

Entity * entity is a pointer to an instance of Entity and

VTArea * vtArea is a pointer to a VT-area manager object.

6.1.4 Running the Simulation

AVTSIM provides a global object called simulator of class Simulator to provide an interface to all the high level functions of AVTSIM, such as setting the simulation mode, logging, and running a simulation.

Summary of the Interface of the Class Simulator

simulationMode (SimulationMode simulationMode) One of LVTMode, AVTMode, GVTMode, setting the mode of the simulation.

distributedMode (bool on) If the parameter on is set to true, the simulation is distributed to several processors. If on is false AVTSIM simulates a distributed simulation using its built in processor and network model.

numberOfProcessors (int numberOfProcessors) Sets the number of processors to use for the simulation. Note that this number must not be greater than the number of physically available processors, or else AVTSIM will print an error message and abort the simulation.

verboseMode (bool on) Switches verbose output of the simulator on (on = true) or off. Each entity will print the virtual time, the state of its in-ports,

the resulting state of the out-ports and values and triggers transmitted, and the resulting final state for each step of the simulation.

run () Runs the simulator indefinitely. The simulator will terminate if, and only if, the simulation model will terminate, i.e. virtual time reaches the maximum virtual time.

run (Model* model) As in run (), however model is used instead of the default model (the collection of all entities defined).

runForRealTime (int time) Runs the simulator for time μ s and then ends the simulation.

runForRealTime (int time, Model * model) As in runForRealTime (time), but uses the model instead of the default.

6.2 Example Simulation

A small example, a latch built from two XOR gates, of a complete AVTSIM simulation is presented next.

```
// Simple Simulation Example: Latch

#include "Simulator.h"

// Logic XOR Gate with two inputs - class definition
class XORGate : public Entity
{
public:
    InPort <bool> in1;
    InPort <bool> in2;
    OutPort <bool> out;

    // Constructor/destructor
    XORGate () // Need to initialise
        : in1 (this), in2 (this), out (this) // in/out-ports with the
        { } // entity's address

    ~XORGate () { }

    // Redefinition of evaluate ()
    // Outputs result after 10 virtual time units
    void evaluate ()
```

```
{
    out.presentAfterDelay (in1.perceive () <> in2.perceive (), 10);
}
};

int main (int argc, char *argv[])
{
    Input    input1;           // Two input-entities (class
    Input    input2;           // definition not shown)
    XORGate  xorGate1 ("XOR1"); // Two XOR gates (instantiations of
    XORGate  xorGate2 ("XOR2"); // class XORGate above)
    Output   output1;          // Two output-entities (class
    Output   output2;          // definition not shown)

    input1.out.connect (& xorGate1.in1); // Connect inputs and
    input2.out.connect (& xorGate2.in2); // outputs to build a
    xorGate1.in2.connect (& xorGate2.out); // latch
    xorGate2.in1.connect (& xorGate1.out);
    xorGate1.out.connect (& output1.in);
    xorGate2.out.connect (& output2.in);

    simulator.distributedMode (true); // Run distributed
    simulator.numberOfProcessors (2); // on two processors
    simulator.runForRealTime (1000000); // for 1 second
}
```

Chapter 7

AVT-Algorithm – Evaluation

Methodology

Generic parameterised models were developed, alongside simulation models derived from literature based on artificial workloads, to investigate the performance of the AVT-algorithm. They were run under different combinations of the parameters to substantially cover the parameter space. This approach was favoured over case studies as it would exercise the AVT-algorithm in an unbiased manner.

The models were implemented and linked into a test application called AVTSIMTest, which reads configuration files containing parameter sets, executes the AVT algorithm for them, and writes the result sets to a file.

7.1 Parameters

The AVT-algorithm was exercised by simulation models over a range of parameters that included:

Model The name of the model to use with the other parameters. The name must refer to one of the models linked to AVTSIMTest.

Number of Processors The number of processors to be used to run the simulation. This number may be less than the number of available processors of the distributed machine.

Execution Time The real-world time given in μs for which the simulation should be run.

Simulation Mode selected from the set {LVT, AVT, GVT}.

Degree of Optimism The amount of virtual time an LP is permitted to speculate into the future. Degree of Optimism $\in \{1, \dots, \infty\}$.

Event Processing Delay The amount of real time given in μs required to compute an output event based on input events.

Output Unchanged Probability A value between 0 and 1 (0% and 100%) which gives the probability of the output of an LP remaining unchanged given that one of the inputs has changed. This is a metric which reflects the stability of an LP's output.

Input Interval The amount of virtual time between the arrival of two input events to the model.

Delay The virtual time between receiving an event in an LP and producing an output event.

Parameters are read from configuration files that contain one parameter set per line. The files containing all combinations of parameters over the ranges, were generated using a tool called MakeConfiguration which was developed for this purpose.

7.2 Result Data

The following result data were collected by AVTSIM for each set of parameters tested and written to the result file. The execution and the simulation time results were used for the performance evaluation, whereas the other collected values served as sanity checks.

Execution Time Time elapsed for running parameter set (in seconds).

Number of Events Computed Total number of output events generated based on input events, i.e. the number of times `evaluate()` was called.

Number of Rollbacks Total count of rollbacks performed due to unsuccessful speculation.

Processor Idle Time Total processor idle time (in μs) of all utilised processors. A breakdown of idle time for each processor is also available.

Number of Messages Sent Total number of messages (event- and AVT-messages) sent locally to each processor and over the network.

Number of Remote Messages Sent Number of messages sent over the network.

Message Transmission Time Sum of the transmission times of all messages (in μs). Transmission time is measured from the instant the message is accepted by AVT-SIM's network layer until it is received by counterpart on the remote processor.

Simulation Time Average virtual time of all LPs at the end of the simulation. A breakdown of virtual time at the end of simulation for each LP is also available.

7.3 Models

The evaluation of the AVT-algorithm concentrated on models that contained both, cyclic and acyclic areas, since these were the target application for the AVT-algorithm. The simulation models for the evaluation are composed of parameterisable test nodes (Section 7.4), which simulated workload and computing of output events based on input events.

We chose one model, "Echo", from the literature ([SrinivasanR98]) because of its documented behaviour of poor performance under optimistic synchronisation policies and which was expected to exhibit poor performance under a conservative synchronisation policy, due to the lack of lookahead. It consists of an input-LP, two LPs connected in a cycle, and a third LP receiving input from both the former LPs (Figure 7.1).

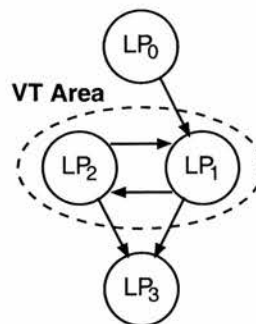


Figure 7.1: Model *Echo* topology

Model *Test* consists of an acyclic and a cyclic area, with the size of each area being given by a parameter. The number of input connections to every LP (TestNode) in each of the areas can be set by a parameter. The four parameters of Model *Test* are:

Number of Nodes in Cyclic Part: an integer $N_C > 0$.

Number of Connections per Node in Cyclic Part: an integer $C_C \in \{1, \dots, N_C - 1\}$.

Number of Nodes in Acyclic Part: an integer $N_A > 0$.

Number of Connections per Node in Acyclic Part: an integer $C_A \in \{1, \dots, N_C\}$.

The parameterised model *Test* consists of one input-node, N_C nodes in the cyclic part, N_A nodes in the acyclic area, and one output-node. The nodes are entered into an array, starting with the input-node, followed by the nodes in the cyclic area, then the acyclic area, and finally the output node. The array indices of the LPs are as follows:

Input node	0
Nodes in cyclic area	$1 \dots N_C$
Nodes in acyclic area	$N_C + 1 \dots N_C + N_A$
Output node	$N_C + N_A + 1$

For all LP_i , $i \in \{1 \dots N_C\}$, the LPs in the cyclic area, LP_i 's output is connected to $LP_{(i+j) \bmod N_C}$'s input, I_j , for all $j \in \{1 \dots C_C\}$, i.e. if the LPs have only one input then they are connected to form a circle; if the LPs have $N_C - 1$ inputs, then the graph representing the cyclic area is completely connected. The first node of the cyclic section has an additional input which is connected to the input-node.

All the inputs I_j , $j \in \{1 \dots C_A - 1\}$ of LPs, LP_i , (where $i \in \{N_C + 1 \dots N_C + N_A\}$) of the acyclic area, are connected to the output of LP_{i-j} . All inputs, I_{C_A} , of LPs, LP_i , of the acyclic area are connected to LP_{i-N_C} 's output, i.e. the LPs in the acyclic area possessing N inputs are connected to the $N - 1$ previous LPs in the array. The remaining input is used to connect the first, second, \dots , N_C^{th} LP in the acyclic area to the first, second, \dots , N_C^{th} LP of the cyclic area, respectively (see examples in Figure 7.2).

The model *Test* is implemented by the class `ModelTest`, a subclass of `Model`. A new instance of `ModelTest` is created by the constructor:

```
ModelTest (int numCyclicNodes,
           int numCyclicConnections,
           int numAcyclicNodes,
           int numAcyclicConnections)
```

An equivalent model to ModelEcho is created by the call `ModelTest (2, 1, 1, 2)`. Some additional examples are given in Figure 7.2.

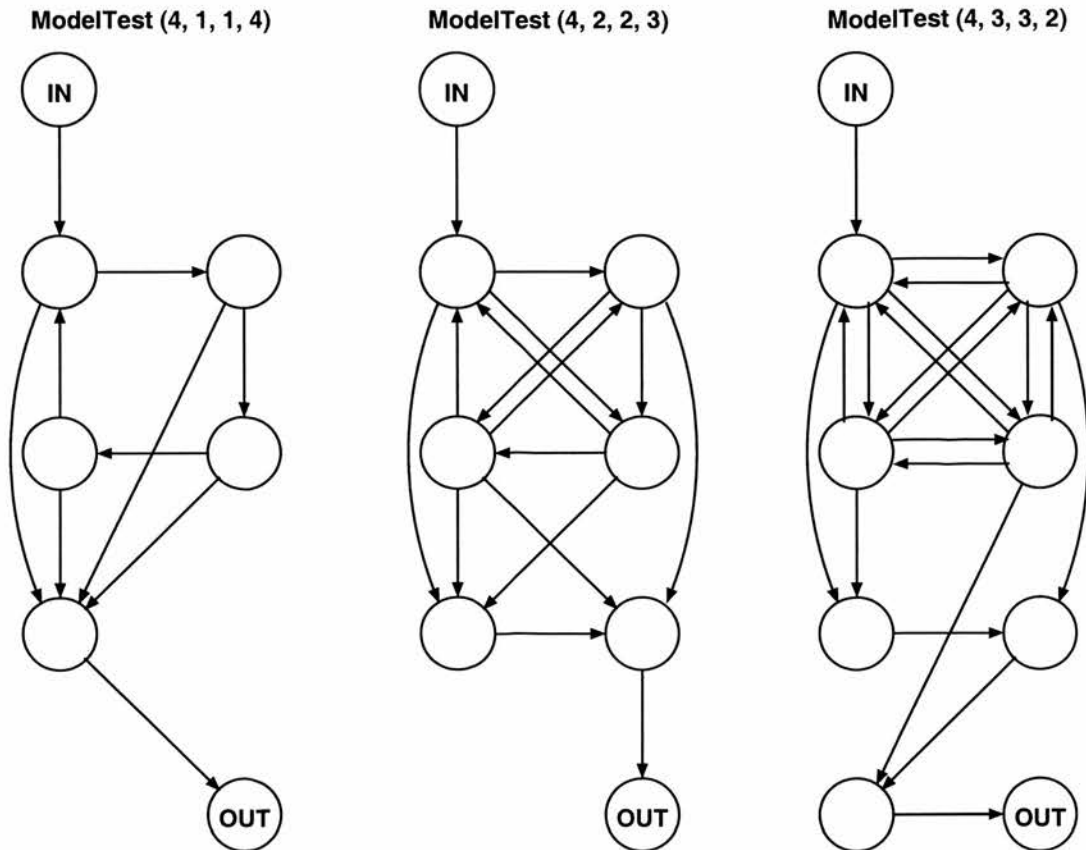


Figure 7.2: Model Test topology examples

7.4 Test Node

The `TestNode` is used to generate an artificial workload for testing AVTSIM. The `TestNode` class is derived from AVTSIM's `Entity` class. It has a configurable number of inputs and outputs of type integer and can be given an alphanumeric name. For instance, the call `TestNode (2, 1, "Example")` creates a `TestNode` with two inputs, one output, and with the name "Example". The simulation parameters can set the time taken by `TestNode` to generate an event and the probability for each input value that has changed to result in a change of output value. Thus `TestNode` is capable of simulating the essential behaviour of any instance of `Entity`.

The behaviour of a `TestNode` is described by its re-definition of `Entity`'s `evaluate()` function:

TestNodes with no inputs (first parameter = 0) behave as input-nodes, i.e. they generate a value based on the current virtual time, every N virtual time units, where N is specified in the simulation parameter *Input Interval*.

TestNodes without outputs (second parameter = 0) can be used to print any events they receive to standard output or to write them to a file. The default behaviour is to consume the event and do nothing.

TestNodes with both inputs and outputs, simulate workload and generate a new event whenever `evaluate()` is called from AVTSIM. The TestNode waits for $N\mu s$ in an empty loop, where N is set in the simulation parameter *Event Processing Delay*. For each input value that is different from the previously received value of the same input, the generated output value is different from the previous output value with the probability of $P = 1 - Q$, where Q is the simulation parameter *Output Unchanged Probability*, which describes the stability of the output value with regard to a change in an input value.

If any of the changes of input values result in a change of output value, then `evaluate()` generates a value M which is different from the previous value after the (virtual) delay given in the simulation parameter *Delay*. The value of M is the current virtual time plus the value of *Delay*. If none of the changes of input values result in a change of output value, then `evaluate()` propagates the previous output value after the virtual time given by *Delay*.

1. $T_{max} = time() + Event\ Processing\ Delay$
2. FOR each input I_i DO
 - IF $I_i.previousValue() \neq I_i.value()$ AND
 - $random(0,1) < 1 - Output\ Unchanged\ Probability$
 - THEN $newValue := true$
3. IF $newValue$
 - THEN $O.presentAfterDelay(virtualTime() + Delay, Delay)$
 - ELSE $O.presentAfterDelay(O.value(), Delay)$
4. WHILE $time() < T_{max}$ DO; END

7.5 Distributed Platform

The simulation algorithms were implemented on a distributed computation platform – a 16-node Beowulf cluster of 1 GHz AMD Athlon processors running the Linux operating system, with 40 GB hard disks, and connected by a 100MBit/s local area network. The message transmission time ($60\mu\text{s}$) was determined, using the UNIX ping command, as the time to send a 64-byte TCP/IP packet from one processor to another. AVTSIM was compiled with the GNU gcc (version egcs-2.91.66) C++ compiler and uses LAM-MPI (version LAM 6.5.2/MPI 2 C++) as the communication system.

Chapter 8

Results

Performance results are presented for the AVT-algorithm executing on a Beowulf cluster of PCs. Two simulation models are considered. Model *Echo* is the smallest model containing cyclic and acyclic parts, and was used to explore one simulation parameter at a time at a fine grain, against only a few samples for the others. The parameterised model *Test* was chosen to investigate the influence of the parameters *Delay* and *Output Unchanged Probability* on the performance of the AVT-algorithm.

8.1 Model *Echo*

The model *Echo* catches the essence of our requirement, being the smallest model with cyclic and acyclic areas, and is shown in Figure 8.1 with the VT-area for execution in AVT-mode. In the GVT-mode the VT-area encompasses all LPs except LP_0 and in LVT-mode no VT-area is allocated.

The parameters which were chosen to change frequently were ones which were most likely to influence the simulation performance. The *Delay* parameter is equivalent of lookahead, and it is therefore important to investigate its influence on the performance of the LVT time-keeping mechanism. The parameter *Output Unchanged Probability* determines the success of optimistic synchronisation. And, the relationship between *Event Processing Delay* and the message transfer time of the underlying architecture is important to understand hardware-dependent performance.

In order to study the influence of these three parameters on the simulation performance,

a sweep of about 100 values was performed for each parameter and the results for the three different simulation modes were compared. Initially, all combinations of the following parameters were considered to be interesting:

Number of Processors 1, 3. When executing the simulation on three processors, LP_0 and LP_1 (Figure 8.1) were allocated to processor 1, and LP_2 and LP_3 to processors 2 and 3 respectively.

Execution Time 1s

Simulation Mode LVT, AVT, GVT

Event Processing Delay 60, 120, 180, ..., 6000 μ s. We chose the *Event Processing Delay* to be a multiple of the time the hardware needed to transmit a message (60 μ s). Choosing a value lower than the message transmission time would make no sense as it would take less time to compute a value locally than to compute it remotely and to receive the result.

Output Unchanged Probability 0, 0.01, 0.02, ..., 1. We test the influence of the *Output Unchanged Probability* for 101 values on a fine-grained scale between 0 and 1.

Input Interval 100VTU (Virtual Time Unit). The event-inter-arrival time for external events is kept constant, since the behaviour of the simulation is determined by the *relationship* of the *Delay* to the *Input Interval*.

Example: the pattern of computation and event-messages is the same for the combination of *Delay* = 50 and *Input Interval* = 100, compared to *Delay* = 500 and *Input Interval* = 1000. The difference in the latter case is that all time-stamps are multiplied by a factor of 10, compared to the former.

Delay 1, 2, ..., 100VTU. For the *Delay* = 100, samples between 1 and the *Input Interval* were taken. Values of *Delay* greater than the *Input Interval* were not a useful choice, because an LP would be busy starting from time t for *Delay* VTU and would be ready to process a new event at time $t + \textit{Delay}$. A new event arriving at time $t + \textit{Input Interval}$ would not be seen by the LP, which would only recognise the next event at time $t + 2 \cdot \textit{Input Interval}$.

Enumerating the entire parameter space and running the simulation for every combination of the parameters, would result in a total of 6060000 data points ($2 \cdot 3 \cdot 100 \cdot 101 \cdot 100 = 6060000$). The required execution time, assuming each parameter set were run

for one second, would be $6060000s \approx 70$ days. This would be too time-consuming, so the investigation focused on one parameter at a time, i.e. one parameter was changed frequently, while only few (3) samples were taken of the others. Example: To investigate the influence of the *Delay* on the simulator's performance, the simulation run for model *Echo* was repeated with $Delay \in \{1, 2, \dots, 100VTU\}$ for all combinations of $Mode \in \{LVT, AVT, GVT\}$, $Number\ of\ Processors \in \{1, 3\}$, *Output Unchanged Probability* $\in \{0, 0.5, 1\}$, and *Event Processing Delay* $\in \{60, 600, 6000\mu s\}$. This technique resulted in 16254 data points ($2 \cdot 3 \cdot (3 \cdot 3 \cdot 100 + 3 \cdot 3 \cdot 101 + 3 \cdot 3 \cdot 100) = 16254$), with a run-time of $16254s \approx 4.5h$ for *Execution Time* = 1s, which is tractable.

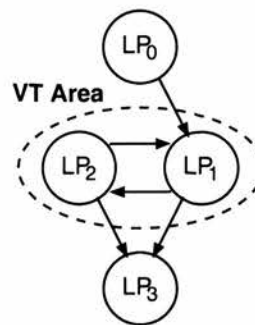


Figure 8.1: Model *Echo* with VT-area for AVT-mode

The metric chosen for comparing the performance under the different parameter sets is *relative progress*. This is defined as the progress in virtual time relative to the execution-time, which is the virtual time at end of the simulation, divided by the processor time to perform the simulation, i.e. greater *relative progress* implies better performance.

The results are presented as a matrix of graphs, with the columns and rows representing the less frequently varying parameters. Represented along the two axes are the frequently-changing parameter (x-axis) and the *relative progress* (y-axis).

8.1.1 Results for Parameter *Delay*

The influence of the *Delay* on performance was investigated under all combinations of the parameters listed below. Each parameter set was run for one second of real-time, and the progress in virtual time was measured and plotted as shown in Figure 8.2.

Number of Processors 1, 3

Execution Time 1s

Simulation Mode LVT, AVT, GVT

Event Processing Delay 60, 600, 6000 μ s

Output Unchanged Probability 0, 0.5, 1

Input Interval 100VTU

Delay 1, 2, ..., 100VTU

These graphs show the effects of the parameter *Delay* on the AVT-Algorithm's performance for different *Output Unchanged Probabilities* (left to right) and *Event Processing Delays* (top to bottom).

For *Output Unchanged Probability* = 0 (left-hand column), the performance of all the modes increases linearly with the *Delay*. The LVT-mode, which performs neither AVT/GVT computation, nor sends AVT-messages, outperforms the AVT- and GVT-modes because optimism always fails due to the instability of the LP's output.

For *Output Unchanged Probability* = 0.5 (middle column), the AVT- and GVT-mode performance is less influenced by the *Delay* than in the previous case. Successful optimism in conjunction with progress based on AVT/GVT leads to good performance even for low *Delay*. LVT-mode performance is reduced in comparison to the previous column. This is because good predictability of output leads to worse lookahead: if an LP receives two consecutive event-messages containing the same value, it does not compute new output based on the second event, because this does not constitute a change in value. Since no new event is computed, the *Delay* does not apply and, since the *Delay* is a non-fixed parameter given to the simulator in every event-computation, the LP only has the default minimum lookahead of 1. Correspondingly, a very large number of null-messages must be sent and progress is slow. Only for very high values of *Delay* (> 90 VTU) does the LVT-mode outperforms AVT- and GVT-modes, as every new event received from the input-LP (LP_0 in Figure 8.1) results in an event-computation in LP_1 that infuses lookahead into LP_1 's computation which is equal to the *Delay*.

These effects as described previously, become even more pronounced for *Output Unchanged Probability* = 1 (right-hand column). AVT/GVT performance is almost independent of the *Delay* as optimism always succeeds, whereas LVT-mode's is very poor,

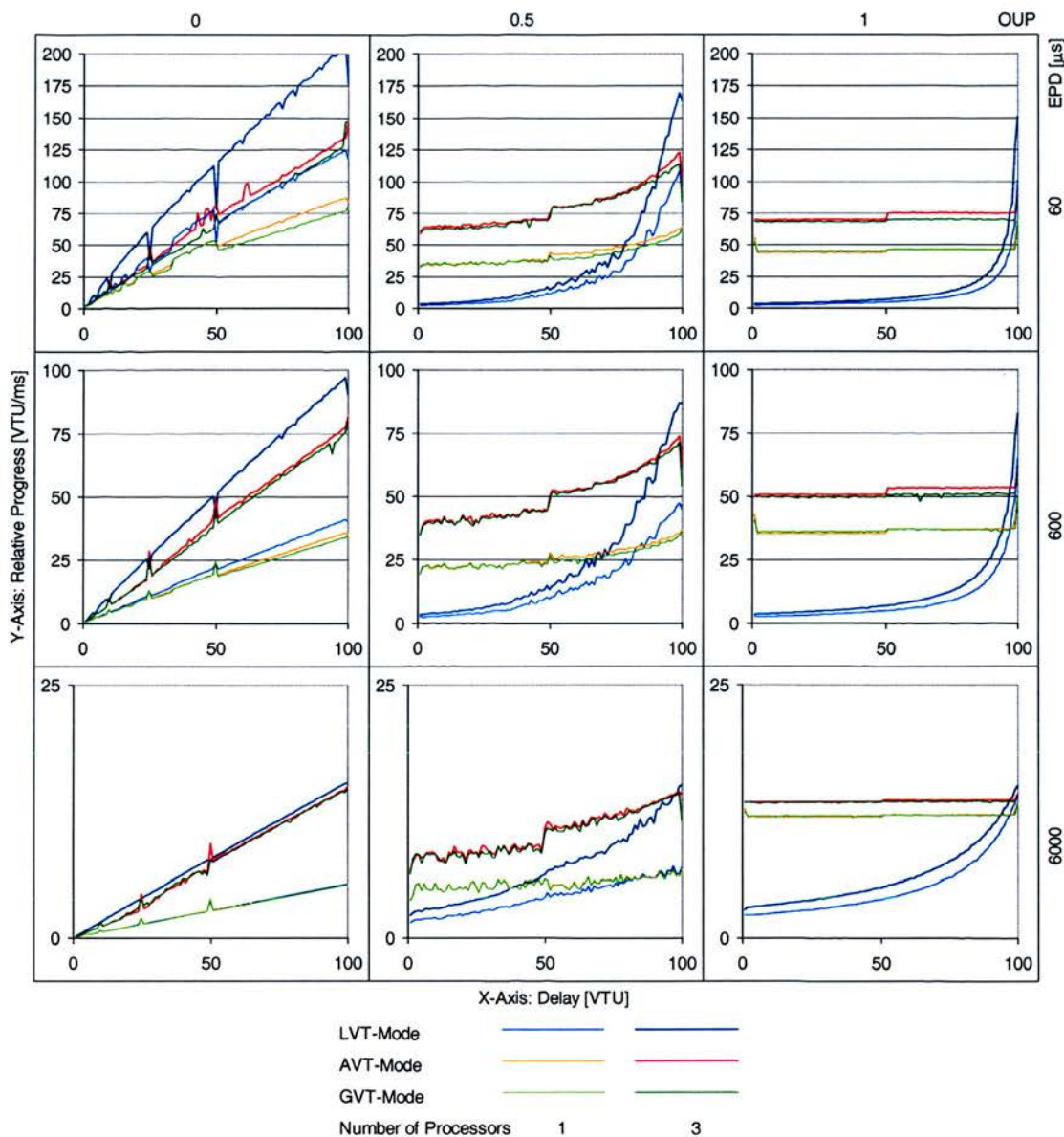


Figure 8.2: *Relative Progress* on 1 and 3 processors for *Delay* = 1, 2, ..., 100VTU; *Event Processing Delay* (EPD) = 60, 600, 6000 μ s; and *Output Unchanged Probability* (OUP) = 0, 0.5, 1. (Scales vary, but the distances between the grid lines along the y-axis represent the same interval in all the graphs.)

due to the lack of lookahead, except for high values of *Delay* when good lookahead is available at LP_2 .

The parameter *Event Processing Delay* (top to bottom row) has no specific effect on any of the simulation modes. As its value was increased, one sees an overall reduction in the performance for all the modes due mainly to the higher computational load,

and the difference between LVT and AVT- and GVT-modes is reduced, since a higher percentage of time is spent in event-computation, and thereby overshadowing the algorithmic overhead.

The difference in the performance of the AVT- and GVT-modes is not great, due to the small difference in size between the VT-areas in the two modes. The LVT-mode performs best for high values of *Delay* and/or low *Output Unchanged Probability*, i.e. when good lookahead is available, and/or optimism is prone to fail.

In general the distributed execution of the algorithm on three processors yields better performance than the corresponding execution in the same mode on a single processor.

8.1.2 Results for Parameter *Event Processing Delay*

The influence of the *Event Processing Delay* on the performance was measured under all combinations of the following parameters:

Number of Processors 1, 3

Execution Time 1s

Simulation Mode LVT, AVT, GVT

Event Processing Delay 60, 120, 180, ..., 6000 μ s

Output Unchanged Probability 0, 0.5, 1

Input Interval 100VTU

Delay 1, 10, 100VTU

Figure 8.3 illustrates the impact of changes in the *Event Processing Delay* on the progress of virtual time. As expected, the progress decreases as *Event Processing Delay* increases. In particular there is no visible differentiating effect of the *Event Processing Delay* on any of the simulation modes, which reduces its importance for comparisons between them.

The performance of the LVT-mode in the top right-hand quadrant (*Output Unchanged Probability* = 0.5, 1; *Delay* = 1, 10) is almost invariant to changes in the *Event Processing Delay*, because of very low lookahead due to low *Delay* and/or high *Output Unchanged Probability*. Most of the progress in these four graphs depend on null-

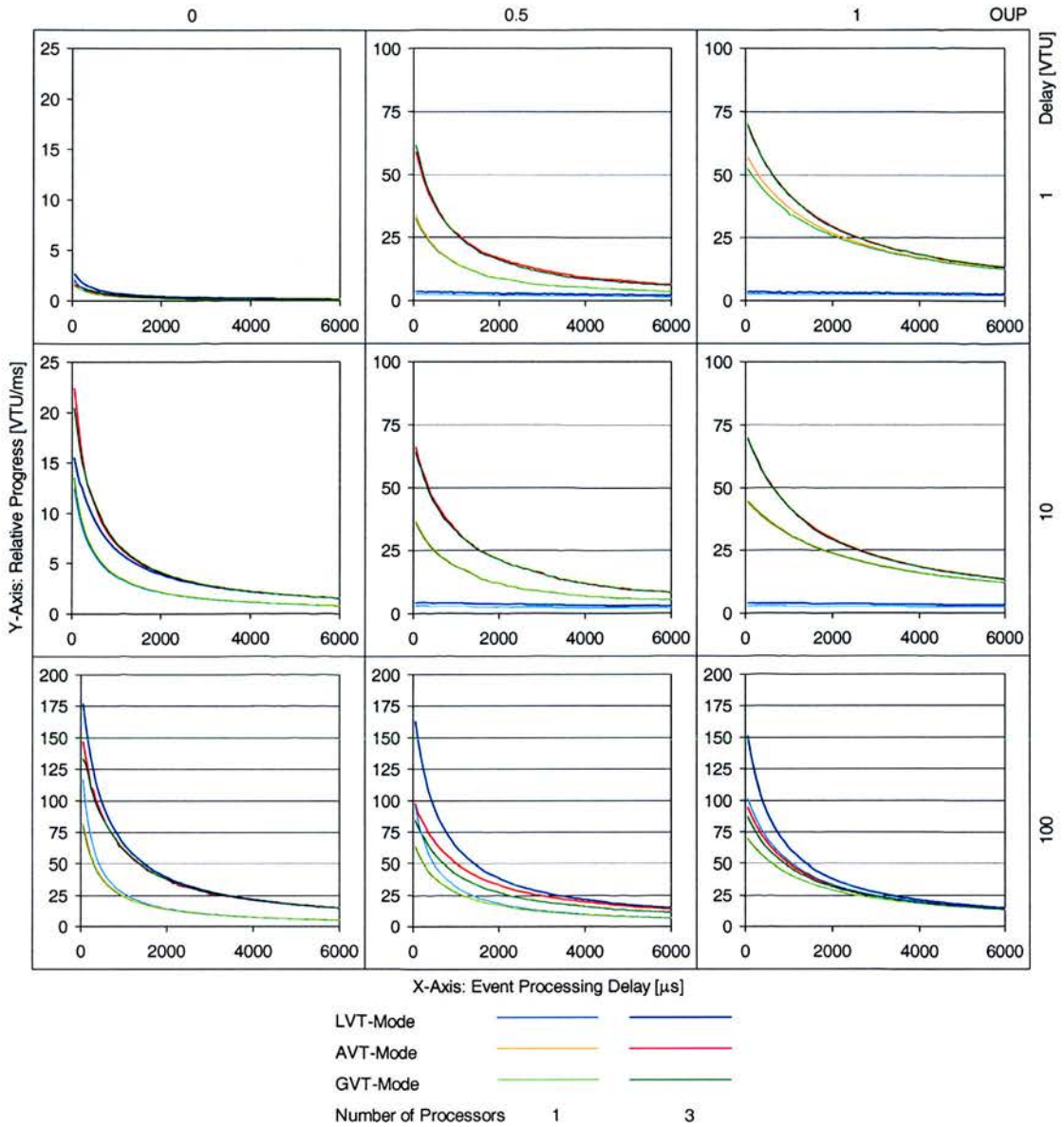


Figure 8.3: *Relative Progress* on 1 and 3 Processors for *Event Processing Delay* = 60, 120, 180, . . . , 6000 μ s; *Delay* = 1, 10, 100VTU; and *Output Unchanged Probability* (OUP) = 0, 0.5, 1. (Scales vary, but the distances between the grid lines along the y-axis represent the same interval in all the graphs.)

messages, i.e. very few event-computations are performed and hence *Event Processing Delay* has little influence on performance.

8.1.3 Results for Parameter *Output Unchanged Probability*

The influence of the *Output Unchanged Probability* on performance was next measured under all combinations of the following parameters.

Number of Processors 1, 3

Execution Time 1s

Simulation Mode LVT, AVT, GVT

Event Processing Delay 60, 600, 6000 μ s

Output Unchanged Probability 0, 0.01, 0.02, ..., 1

Input Interval 100VTU

Delay 1, 10, 100VTU

Figure 8.4 shows that with an increase in the *Output Unchanged Probability* the performance of the AVT- and GVT-modes increases, except for *Delay* = 100 (bottom row) when the lookahead matched the event inter-arrival time exactly. For the *Delay* = 1 (top row), progress in LVT-mode is independent of the *Output Unchanged Probability* since the lookahead is equal to 1, regardless of whether progress is made through event-computations (*Output Unchanged Probability* = 0) or null-messages (*Output Unchanged Probability* = 1). For *Delay* = 10 (middle row), for small values of *Output Unchanged Probability*, performance increased slightly compared to the row above, due to the slightly better lookahead for event-computations. Both the GVT- and AVT-modes outperform the LVT-mode for most of the *Output Unchanged Probability* parameter range.

The performance of LVT-mode for 3 processors and *Delay* = 100 (bottom row) decreases as the *Output Unchanged Probability* increases, as the lookahead infused by event-computations in LP_1 (Figure 8.1) is reduced due to increasing *Output Unchanged Probability*. On a single processor, LVT performance increases with *Output Unchanged Probability* as event-computation is replaced by null-message processing, except for the case when *Event Processing Delay* = 60 μ s, in which initially the effect of reduced lookahead dominates.

AVT-mode performance decreases for low values of *Output Unchanged Probability* (≤ 0.1) and remains level for greater values of *Output Unchanged Probability*. GVT-

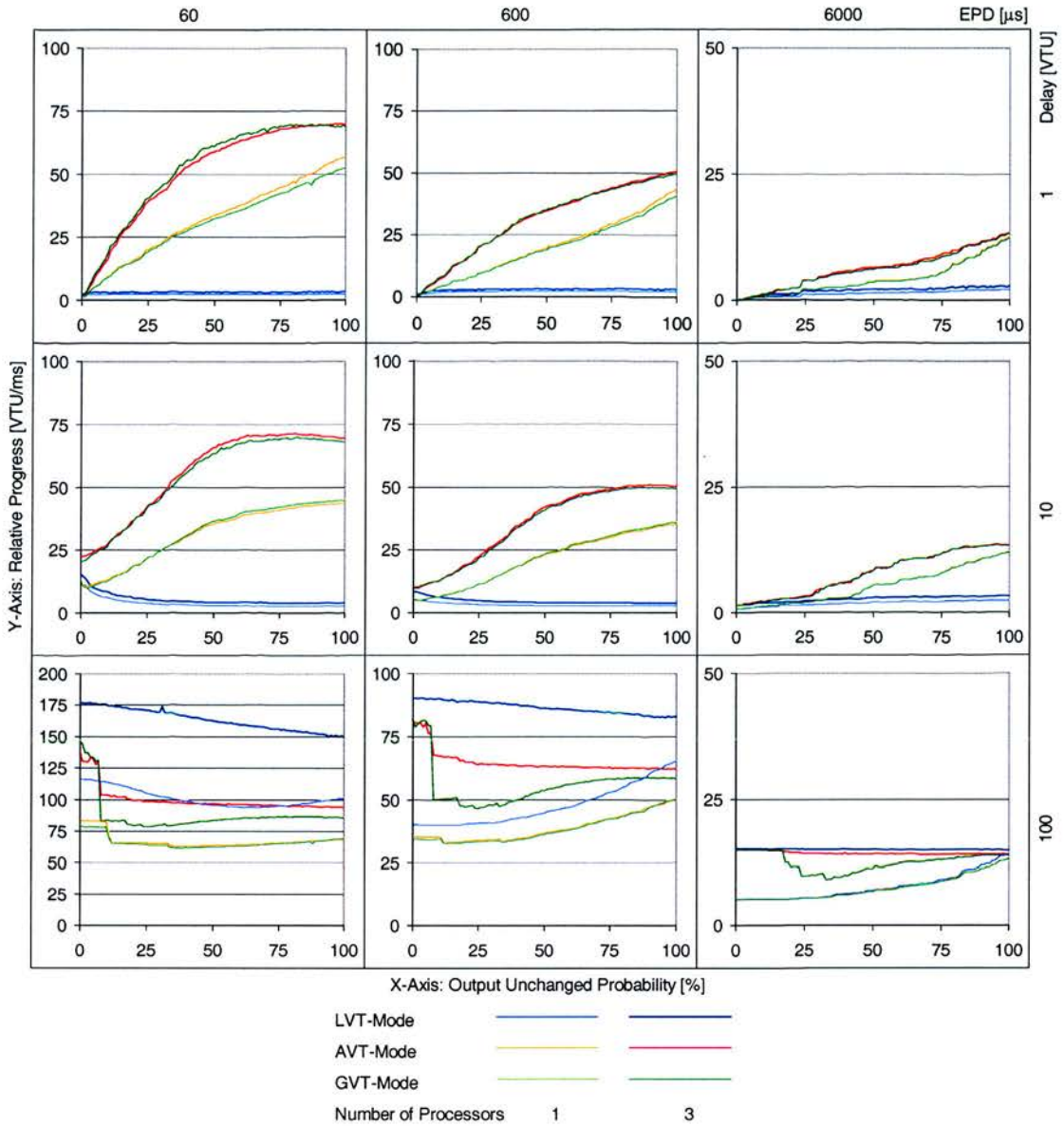


Figure 8.4: *Relative Progress* on 1 and 3 processors for *Output Unchanged Probability* = 0, 0.01, 0.02, ..., 1; *Delay* = 1, 10, 100VTU; and *Event Processing Delay* (EPD) = 60, 600, 6000 μs (Scales vary, but the distances between the grid lines along the y-axis represent the same interval in all the graphs.)

mode performance drops off more than AVT-mode performance but recovers for larger values of *Output Unchanged Probability* and almost matches AVT performance.

For very low *Output Unchanged Probability* and good lookahead, AVT-mode progresses based on LVT alone. AVT-updates are sent to the AVT-keeper, but either the LVT in the LPs is greater than any AVT computed and hence an AVT-update is not sent

to any LPs, or any AVT-update is received at the LPs after they have already updated their LVT beyond the AVT, and so the update has no effect. However, once an AVT-message arrives at an LP and actually causes an AVT-update, progress can only be made based on AVT, since there is no more committed input and hence no lookahead in the cycle. This effect slows down progress for low *Output Unchanged Probability*, but diminishes for greater *Output Unchanged Probability*.

With increases in *Event Processing Delay* (left to right column), no specific effect can be observed, but simulation progress is reduced due to the increased computational load.

8.1.4 Summary of Results for Model *Echo*

No significant differences between the performance of the AVT- and GVT-modes can be observed in model “Echo”, because the number of LPs in the VT-area in GVT-mode is not much larger than the those in the VT-area in AVT-mode. Differences were observed only for large *Output Unchanged Probability* and large *Delay*. In the next section we will study larger models containing larger areas of different time-keeping mechanisms.

The effect of the *Event Processing Delay* on the various *Simulation Modes* was not significant, and this parameter was not studied further in a fine-grained fashion.

The parameters *Output Unchanged Probability* and *Delay* have significant impact on the performance of LVT-mode performance due to their influence on lookahead: greater *Delay* improves lookahead, while higher *Output Unchanged Probability* reduces it. The *Output Unchanged Probability* has important consequences for the performance of AVT- and GVT-modes as good predictability of an LP’s output is a requirement for the success of optimism, which enables the AVT- and GVT-modes to perform well in the absence of lookahead. In the next section we concentrate on understanding the effect of parameters such as *Delay* and *Output Unchanged Probability* on the performance of the different time-keeping mechanisms.

8.2 Model Test

Test is a class of simulation models designed to exercise the AVT-algorithm, which

contain a mixture of cyclic and acyclic areas. The models were chosen to contain cyclic and acyclic areas of equal sizes, which situates them in the middle of the continuum between fully cyclic and fully acyclic simulation models. They are characterised by the number of LPs in each of the areas and the density of their communication topology, i.e. the number of connections per LP.

The smallest model in *Test* consists of two LPs in each area, and increasing to four in the other models. The communication density ranges from one, to two, and four connections per LP.

The exploration could be continued in this fashion by doubling the size of model *Test* and, for each new model instance, by also investigating varying communication densities, starting at 1 and doubling up to the maximum density.

For practical reasons, we did not venture beyond a model size of eight LPs given the number of available processors (12) available in the distributed platform.

The nomenclature of the models in *Test* is shown in Table 8.1 below.

Cyclic Part		Acyclic Part		Total LPs	Name
LPs	Connections per LP	LPs	Connections per LP		
2	1	2	2	4	<i>Test</i> _{2,1,2,2}
4	1	4	1	8	<i>Test</i> _{4,1,4,1}
4	2	4	2	8	<i>Test</i> _{4,2,4,2}
4	3	4	4	8	<i>Test</i> _{4,3,4,4}
Input and the output LPs have been excluded					

Table 8.1: Model *Test* configurations

The LPs are characterised in terms of the time taken to process an event in the LP (*Delay*), and the stability of an LP's output (*Output Unchanged Probability*, i.e. the likelihood of the input-event changing the value of the output event).

In order to reduce the number of data points to a manageable level, it was decided to take 10 samples each for *Delay* between 0 and 100VTU, and the *Output Unchanged Probability* between 0.0 and 1.0, which resulted in a grid of size 10×10 on the combined parameter range.

All instances of the models in *Test* shown in Table 8.1 were run under all combinations of the parameters described below:

Number of Processors 1, 2, 4, 8. Each instance of the models in *Test* was run using all processors, up to the useful limit for distribution, i.e. the total number of LP in the model as given in Table 8.1. The LPs were evenly distributed to the processors, as described in Section 6.1.3.

Execution Time 10s

Simulation Mode LVT, AVT, GVT. In AVT-mode, the cyclic sub-model connected to the input-LP (marked IN) in Figures 8.5, 8.9, 8.14, and 8.19, is allocated as the VT-area, whereas in GVT-mode all LPs except the input- and the output-LP use the GVT time-keeping mechanism.

Event Processing Delay 100, 1000, 10000 μ s. The lowest value of *Event Processing Delay* was chosen to be of the same order of magnitude as the message delay of the distributed computer (60 μ s), and the other values were one and two orders of magnitude greater, respectively.

Output Unchanged Probability Ten values between 0 and 1: 0, 0.11, 0.22, 0.33, 0.44, 0.56, 0.67, 0.78, 0.89, 1.

Input Interval 100 virtual time units (VTU)

Delay Ten integer values between 1 and 100VTU: 1, 12, 23, 34, 45, 56, 67, 78, 89, 100VTU.

The *relative progress*, i.e. virtual time divided by execution time, has been measured for each combination of the parameters resulting in 2700 data points ($3 \cdot 3 \cdot 3 \cdot 10 \cdot 10 = 2700$) being generated in 27000s = 7.5h for model *Test*_{2,1,2,2}, and 3600 data points ($4 \cdot 3 \cdot 3 \cdot 10 \cdot 10 = 3600$) being generated in 36000s = 10h for each model *Test*_{4,x,4,x}.

The results are presented in a graph with tuples (*Output Unchanged Probability*; *Delay*) along the x-axis and the *relative progress* along the y-axis. This results in a graph consisting of an array of plot-lines. Each individual plot-line shows results for all values of *Delay* for one choice of *Output Unchanged Probability*. In each graph the results for the three different *Simulation Modes* have been compared.

While the scales of the graphs may vary, the distances between grid-lines represent the same numerical interval. The interval between dotted grid-lines is one-tenth of the

interval between solid grid-lines.

8.2.1 Model $Test_{2,1,2,2}$

Figures 8.6 – 8.8 compare the performance of LVT-, AVT-, and GVT-modes for *Event Processing Delay* $\in \{100, 1000, 10000\mu s\}$. Each figure displays the results obtained by running on 1, 2, and 4 processors.

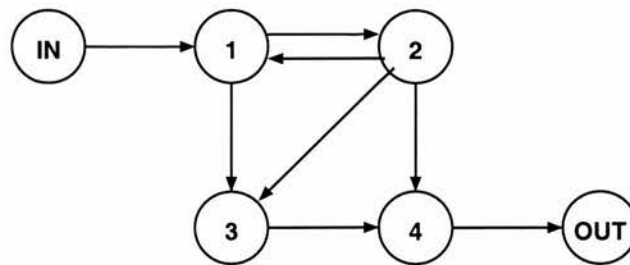


Figure 8.5: Model $Test_{2,1,2,2}$

The performance of the LVT-mode increases with the *Delay*. However, this increase is influenced by the *Output Unchanged Probability*: the higher the value, the more does the performance curve sag, i.e. it has a low incline for low *Delay* and rises more steeply with increasing *Delay*. This is due to the lookahead being dependent on *Delay* and *Output Unchanged Probability*: high *Delay* implies large lookahead and high *Output Unchanged Probability* reduces lookahead due to event-computation being replaced by processing of null-message which delivers no lookahead. LVT-mode's good performance for high *Delay* and high *Output Unchanged Probability* is caused by the input-LP's events sent every 100VTU, which infuses (through event-computation with $Delay = 100VTU$) enough lookahead to span the entire cycle.

AVT- and GVT-mode performance is similar to LVT-mode for low *Output Unchanged Probability*, i.e. it is dependent on lookahead. This is because the output of the LPs is very unpredictable, i.e. optimism fails, and the AVT-algorithm progresses based on LVT alone, since the AVT computed by the AVT-keeper does not exceed the LVT at each LP, and is therefore never propagated to any of them. The overhead of both sending AVT-messages to the AVT-keeper, and computing the AVT, reduces performance compared to LVT-mode.

As the *Output Unchanged Probability* increases, optimism succeeds more often and

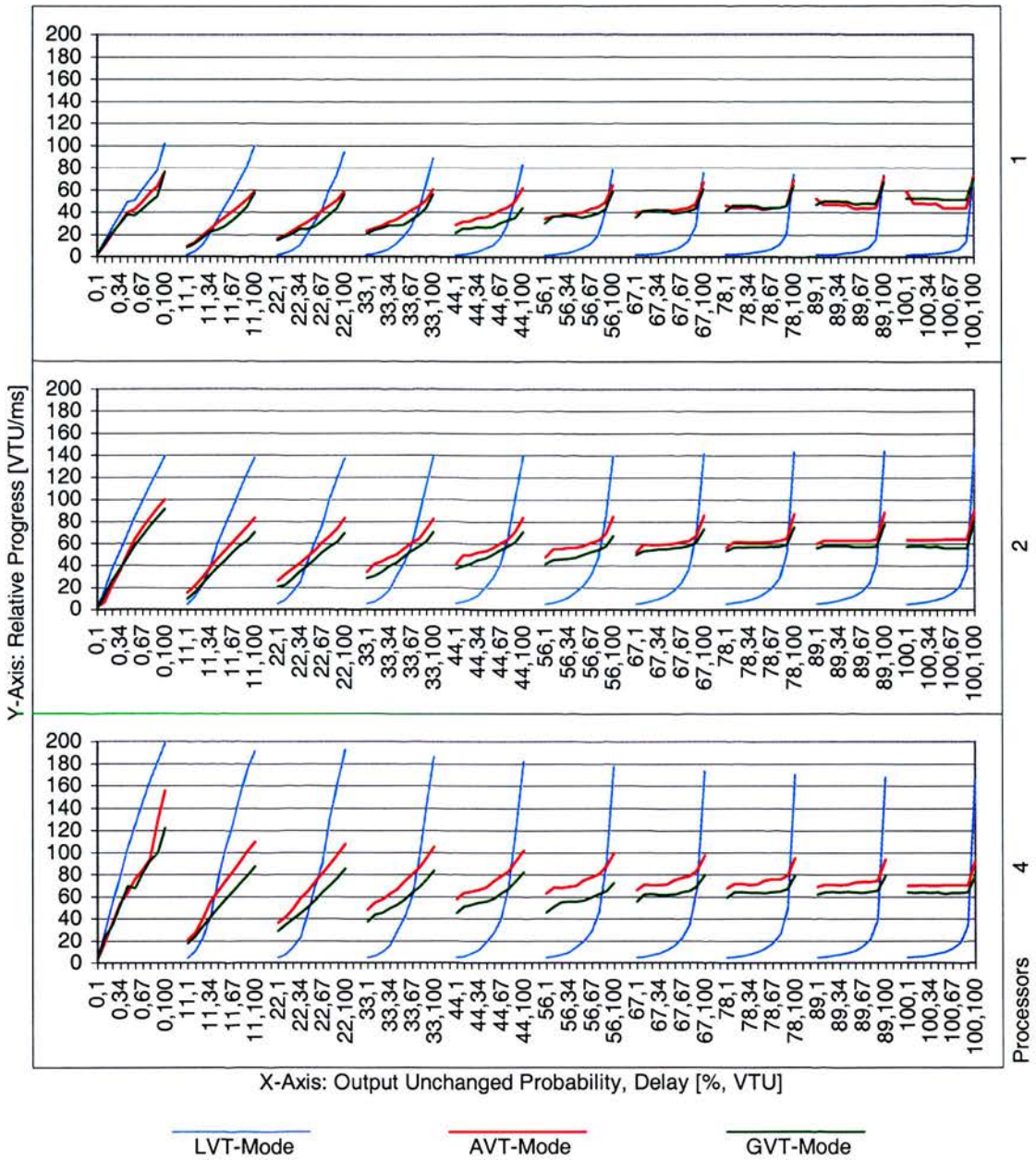


Figure 8.6: Model $Test_{2,1,2,2}$; Relative Progress versus Output Unchanged Probability and Delay on 1, 2, 4 Processors with Event Processing Delay = $100\mu s$

AVT can improve on the LVT of each LP. This results in the AVT-mode performance that is increasingly more stable regarding Delay, i.e. more independent of lookahead, for increasing values of Output Unchanged Probability, which is reflected by almost horizontal plot lines for high Output Unchanged Probability.

When comparing the AVT- and the GVT-mode, the former outperforms the latter as it has lower AVT propagation and computation cost.

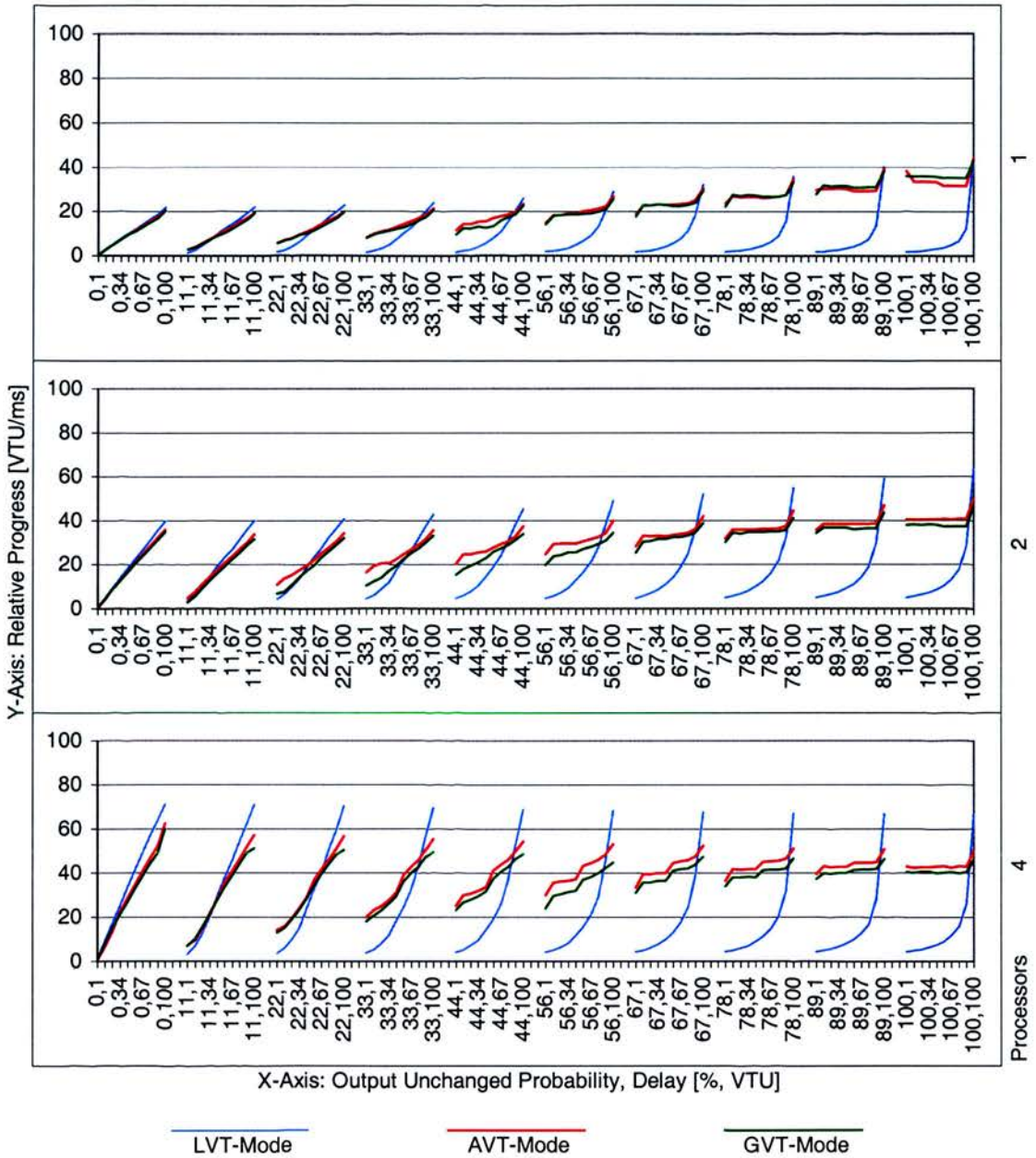


Figure 8.7: Model $Test_{2,1,2,2}$ Relative Progress versus Output Unchanged Probability and Delay on 1, 2, 4 Processors with Event Processing Delay = $1000\mu s$

Apart from an overall reduction in simulation progress, the *Event Processing Delay* influences the difference in the maximum performance of LVT ($Delay = 100$) compared to AVT. This is smaller for higher *Event Processing Delay* because algorithmic overheads of the GVT time-keeping are dwarfed by the computation requirements for event-processing (Figure 8.8).

Overall, AVT-mode performs best in around two-thirds (66%) of parameter sets on 2

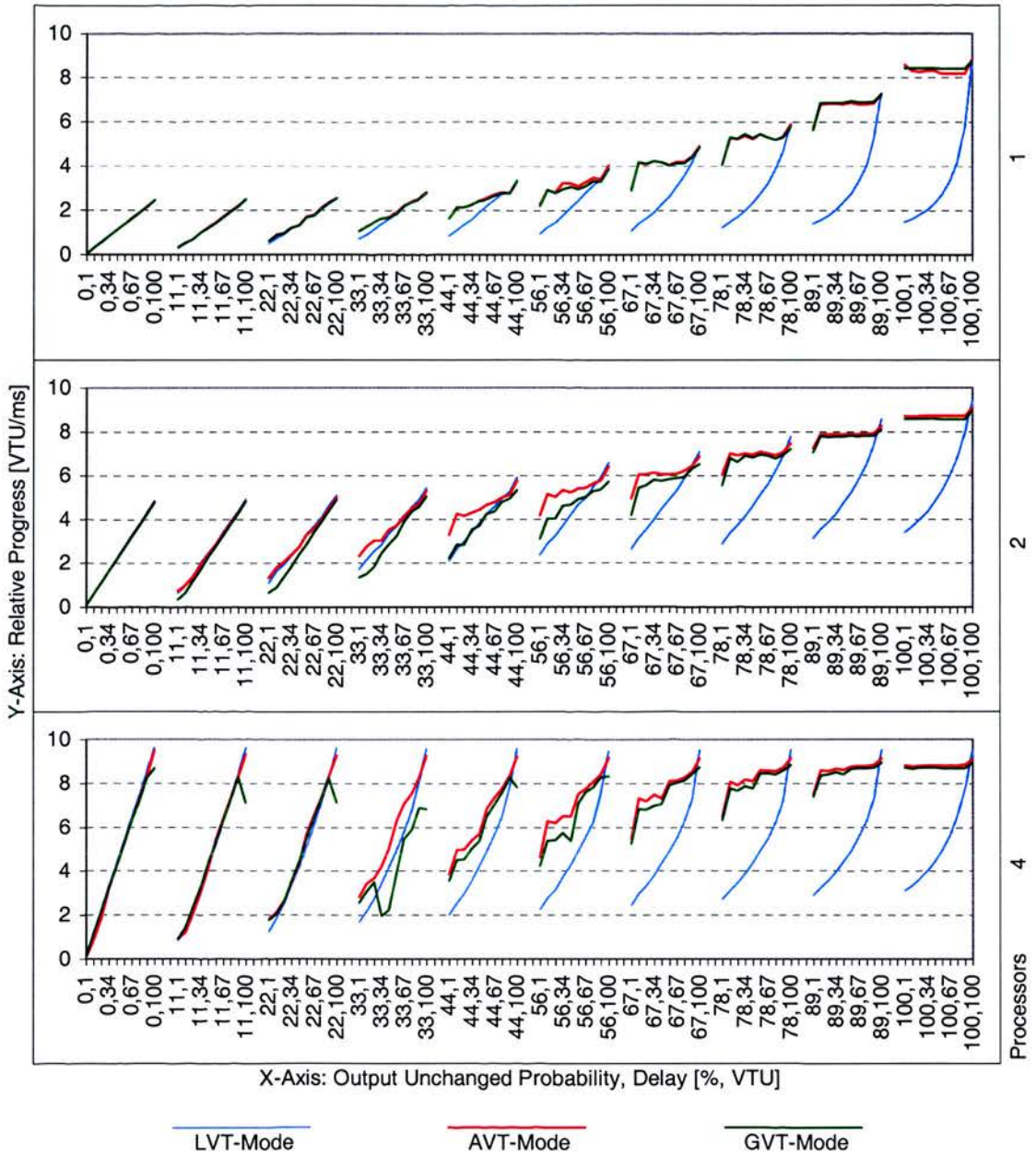


Figure 8.8: Model $Test_{2,1,2,2}$ Relative Progress versus Output Unchanged Probability and Delay on 1, 2, 4 Processors with Event Processing Delay = 10000 μ s

processors and 71% on 4 processors.

8.2.2 Model $Test_{4,1,4,1}$

The three modes were compared as before for model $Test_{4,1,4,1}$, a model with 1 connection per LP, for *Event Processing Delay* of $100\mu s$ (Figure 8.10), $1000\mu s$ (Figure 8.11), and $10000\mu s$ (Figure 8.12). Each figure displays the results obtained by running the simulation on 1, 2, 4, and 8 processors. As before displayed along the x-axis are tuples: (*Output Unchanged Probability*, *Delay*), with the *relative progress* along the other axis.

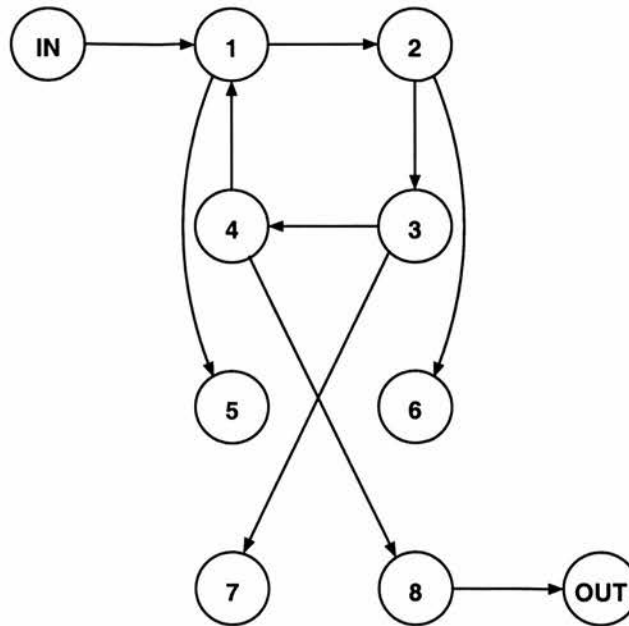


Figure 8.9: Model $Test_{4,1,4,1}$ topology

The results for model $Test_{4,1,4,1}$ are similar to the performance results for model $Test_{2,1,2,2}$: the performance of the LVT-mode increases with the *Delay*. This, however, is influenced by the *Output Unchanged Probability*: the higher the value, the more performance is reduced for low *Delay*. For low *Output Unchanged Probability*, AVT- and GVT-mode performance depends on lookahead, and as the *Output Unchanged Probability* increases, AVT-mode performance becomes more stable regarding *Delay*. As in Section 8.2.1, the *Event Processing Delay*'s effect is a reduction of the simulation progress.

Overall, the AVT-mode performs best in 63% of parameter sets on 2, 65% on 4, and 74% on 8 processors.

Figure 8.13 summarises the speedup of LVT-, AVT-, and GVT-mode for 2, 4, and 8

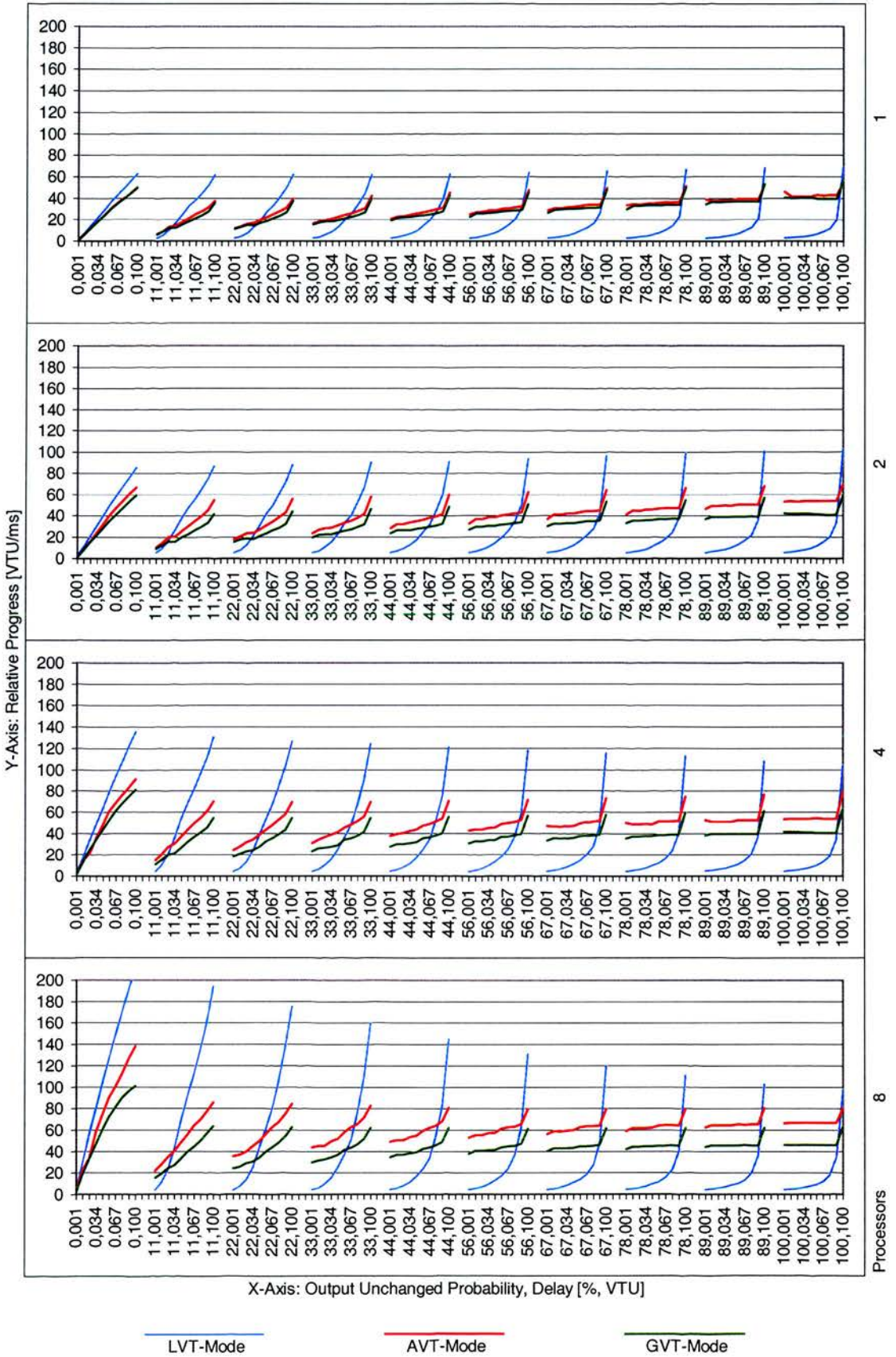


Figure 8.10: Model $Test_{4,1,4,1}$ Relative Progress versus Output Unchanged Probability and Delay on 1, 2, 4, 8 Processors with Event Processing Delay = $100\mu s$

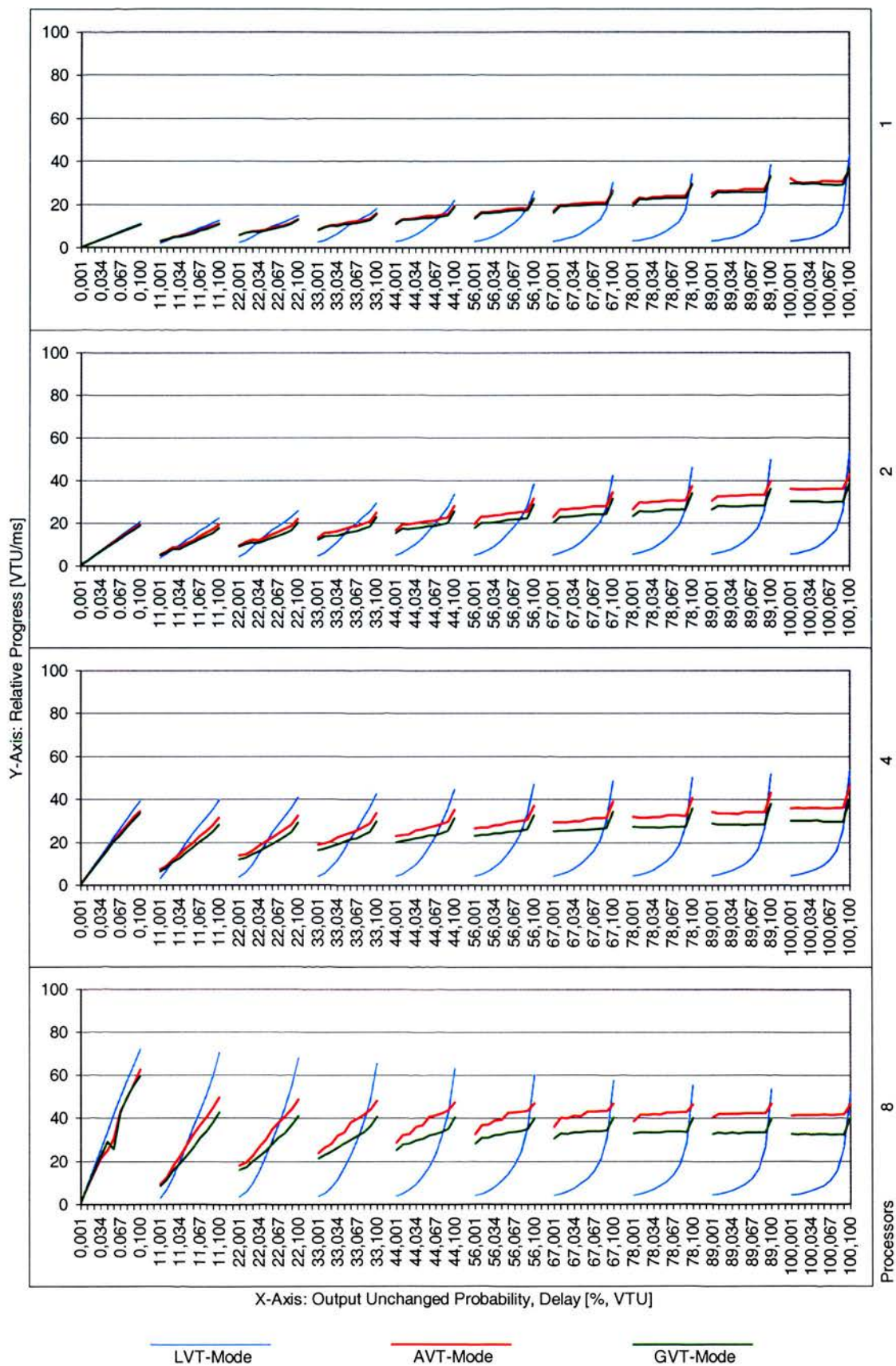


Figure 8.11: Model $Test_{4,1,4,1}$ Relative Progress versus Output Unchanged Probability and Delay on 1, 2, 4, 8 Processors with Event Processing Delay = $1000\mu s$

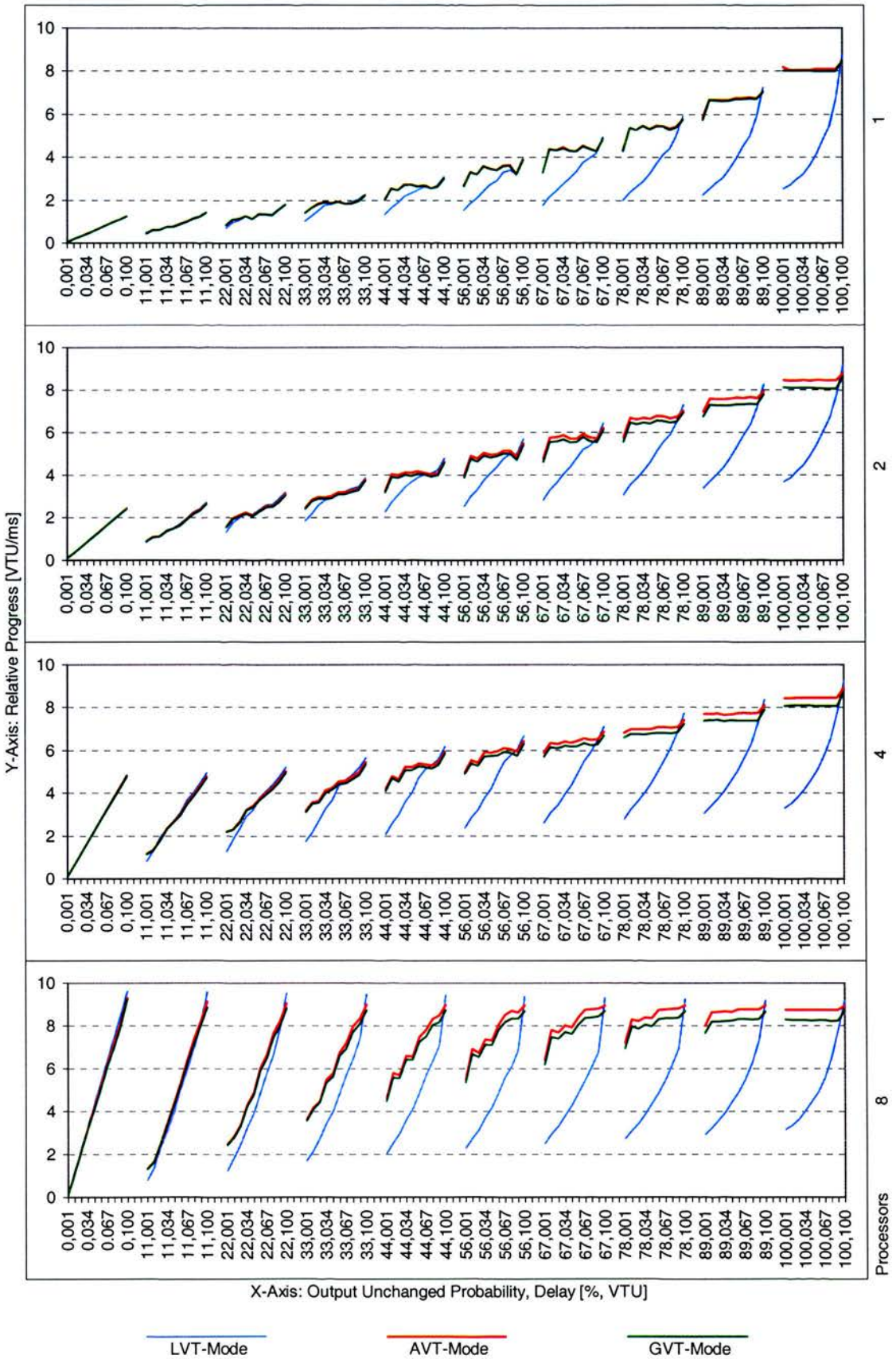


Figure 8.12: Model $Test_{4,1,4,1}$ Relative Progress versus Output Unchanged Probability and Delay on 1, 2, 4, 8 Processors with Event Processing Delay = $10000\mu s$

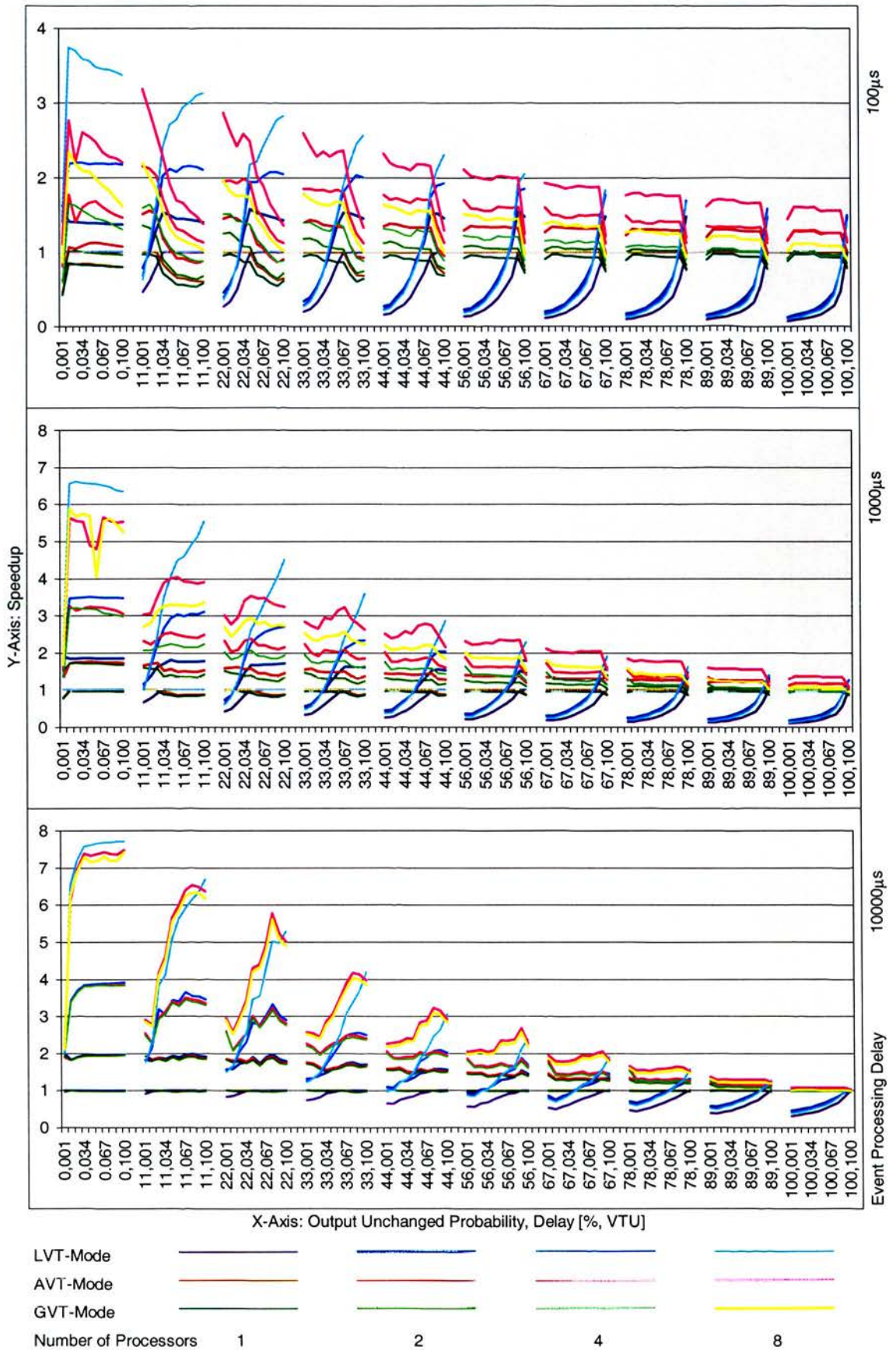


Figure 8.13: Model $Test_{4,1,4,1}$: speedup relative to the best case on uniprocessor for 2, 4, and 8 processors and Event Processing Delay = 100, 1000, 10000 μs

processors compared to the best performance on a single processor. The results for all the processors are displayed in a single graph, with colour-coded plot-lines. LVT-mode is represented by shades of blue, ranging from dark blue for a single processor to cyan for eight processors. Similarly, the AVT- and GVT-modes are represented by shades of red and green colours, respectively.

The graphs show that AVT-mode results in the best speedup except for low *Output Unchanged Probability* or very high *Delay*. The reasons lie in the differences in performance between the LVT- and AVT- and GVT-modes described previously. Speedup decreases notably with an increase in *Output Unchanged Probability* especially for *Event Processing Delay* = 1000 μ s and *Event Processing Delay* = 10000 μ s. This is caused by the high stability of the LP's output which results in very few event-computations and reduces the amount of parallelism that can be exploited. Any remaining speedup is an effect of distributing the computations performed by the synchronisation and time-keeping algorithm.

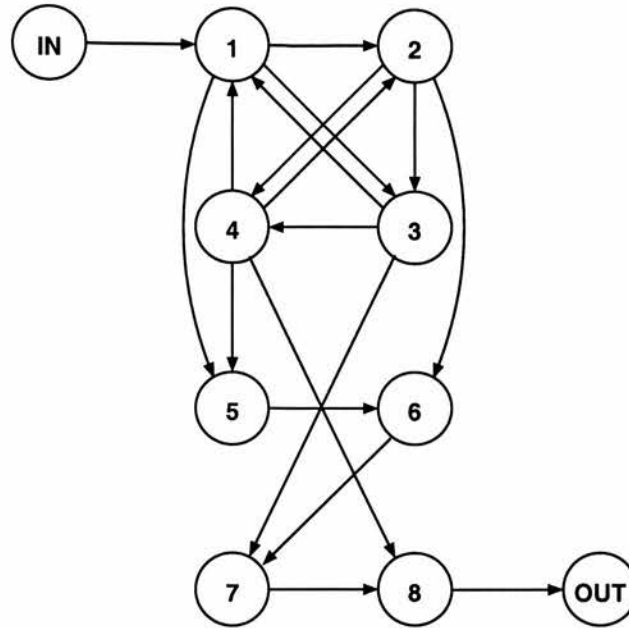
In the next two sections (8.2.3, 8.2.4) we compare the performance for model *Test*_{4,1,4,1} under all the above parameter sets with that of its "sister" models which have greater communication densities, i.e. instead of 1, they have 2 (model *Test*_{4,2,4,2}), and 4 (model *Test*_{4,3,4,4}) connections per LP.

8.2.3 Model *Test*_{4,2,4,2}

Figures 8.15 – 8.17 show the performance of LVT, AVT, and GVT-mode for the model *Test*_{4,2,4,2}, a model with 2 connections per LP, for *Event Processing Delay* of 100, 1000, and 10000 μ s. Each figure displays the results obtained running the simulation on 1, 2, and 4 processors from top to bottom. The x-axis is a two dimensional axis showing *Output Unchanged Probability* and *Delay*; the y-axis shows *relative progress*.

LVT-mode performance is similar to model *Test*_{4,1,4,1} for low *Output Unchanged Probability*, i.e. progress increases with the *Delay*, however, in contrast to model *Test*_{4,1,4,1}, with increasing *Output Unchanged Probability* performance of LVT-mode dramatically decreases due to the higher number of connections on which null-messages must be sent for time-keeping.

As in model *Test*_{4,1,4,1} AVT- and GVT-mode performance is dependent on lookahead for low *Output Unchanged Probability* as the output of the LPs is very unpredictable,

Figure 8.14: Model $Test_{4,2,4,2}$

which prevents optimism from succeeding. With increasing *Output Unchanged Probability* optimism succeeds more often and AVT-mode performance is more stable regarding *Delay*. AVT-mode outperforms GVT-mode because the former reduces AVT computation and propagation cost in comparison with the latter.

As for model $Test_{4,1,4,1}$ *Event Processing Delay* has an overall reducing effect on simulation progress.

Generally, performance of AVT-mode in model $Test_{4,2,4,2}$ is reduced compared to model $Test_{4,1,4,1}$ – the effect of the increased number of connections between LPs.

Overall AVT-mode performs best in 68% of parameter sets on 2 processors, 72% on 4, and 84% on 8 processors.

Figure 8.13 summarises the speedup of LVT, AVT, and GVT-mode on 2, 4, and 8 processors. The graphs show that in contrast to model $Test_{4,1,4,1}$ LVT-mode results in the best speedup only for low *Output Unchanged Probability*. For higher values of *Output Unchanged Probability* AVT-mode is best, even for high *Delay*.

The graphs show that AVT-mode results in the best speedup except for low *Output Unchanged Probability* caused by the differences in performance between LVT-mode and the AVT- and GVT-modes described above. Speedup for all *Execution Modes* decreases notably with an increase in *Output Unchanged Probability* especially for

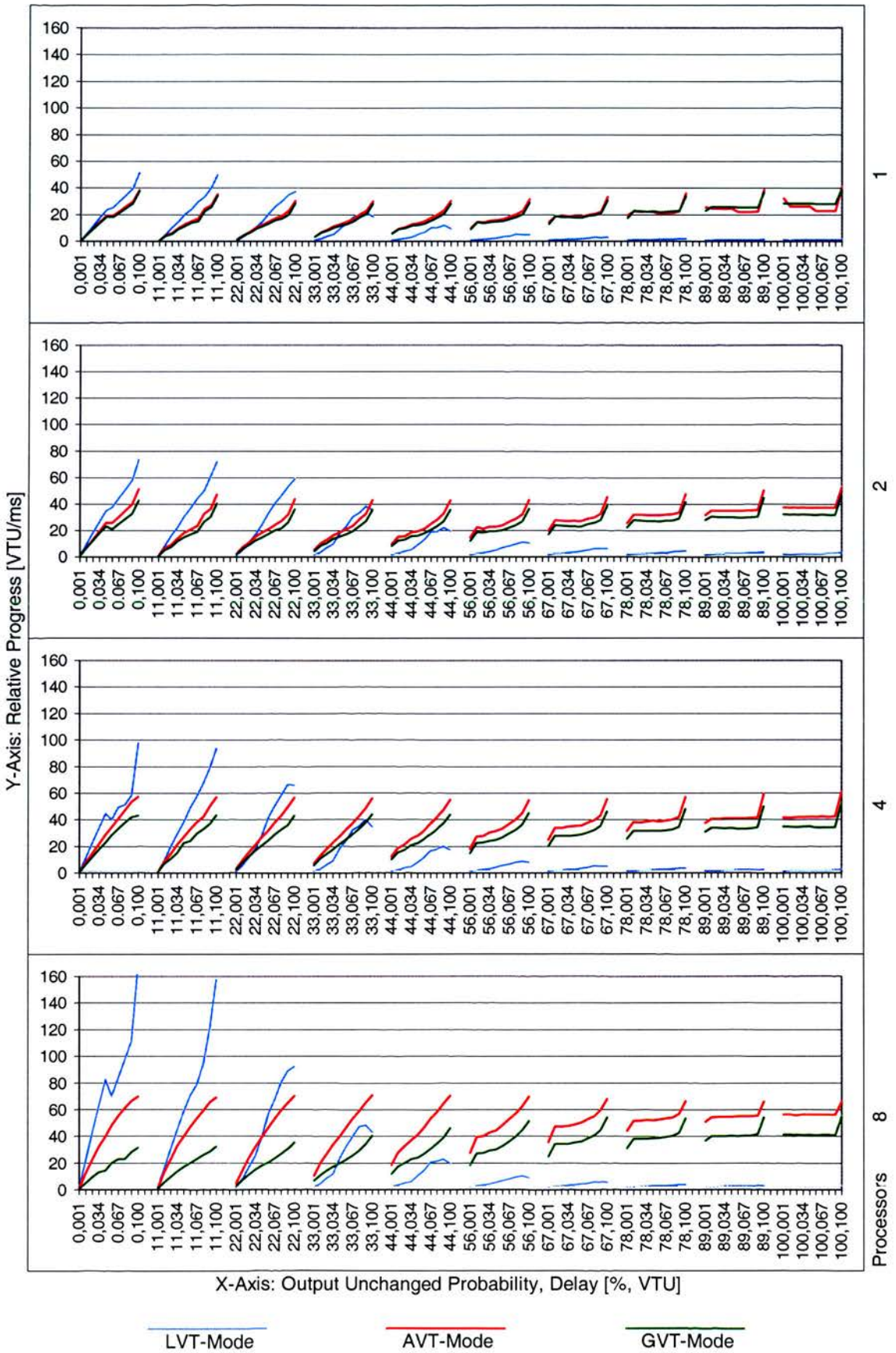


Figure 8.15: Model $Test_{4,2,4,2}$ Relative Progress versus Output Unchanged Probability and Delay on 1, 2, 4, 8 Processors with Event Processing Delay = $100\mu s$

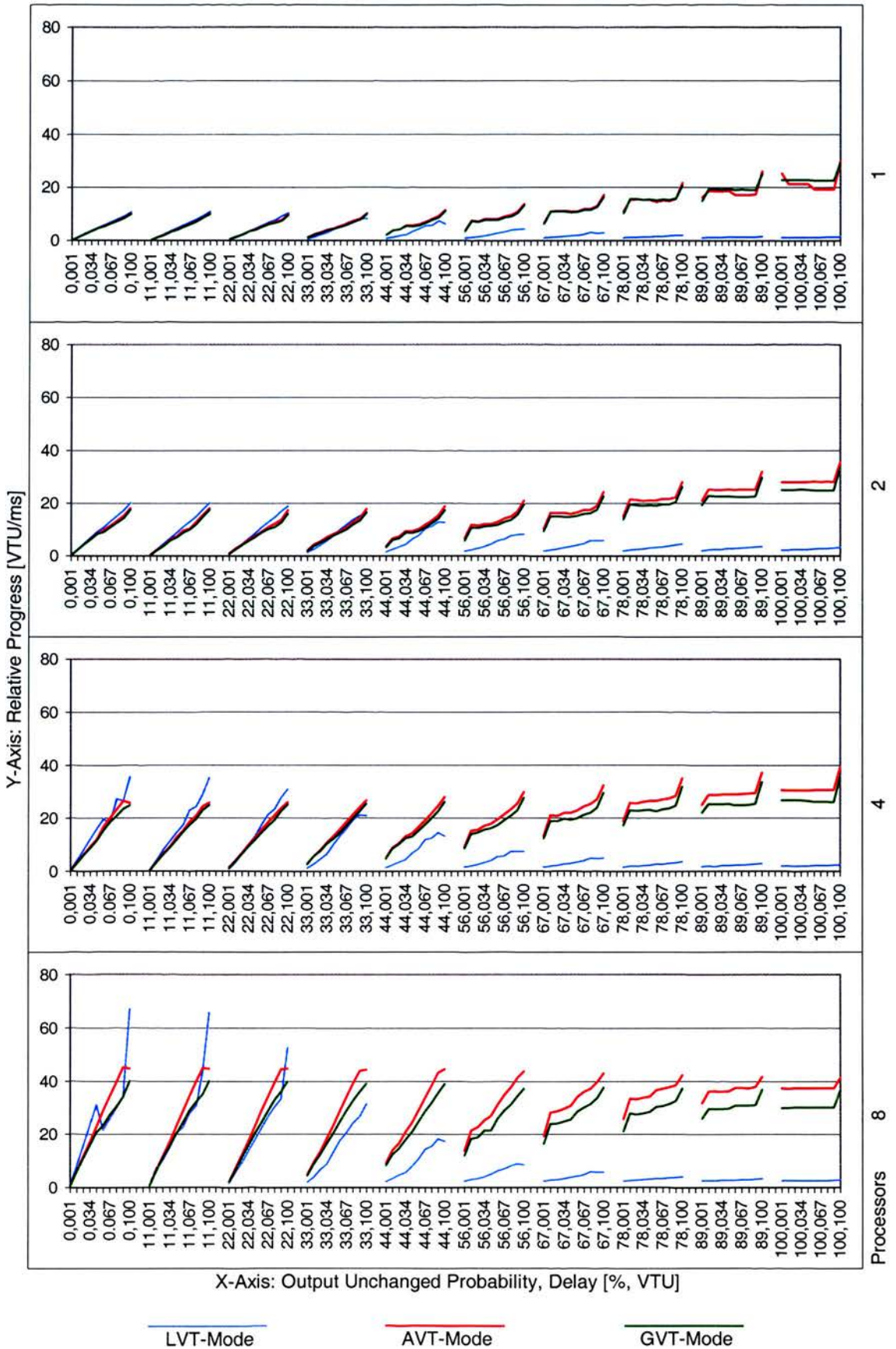


Figure 8.16: Model $Test_{4,2,4,2}$ Relative Progress versus Output Unchanged Probability and Delay on 1, 2, 4, 8 Processors with Event Processing Delay = $1000\mu s$

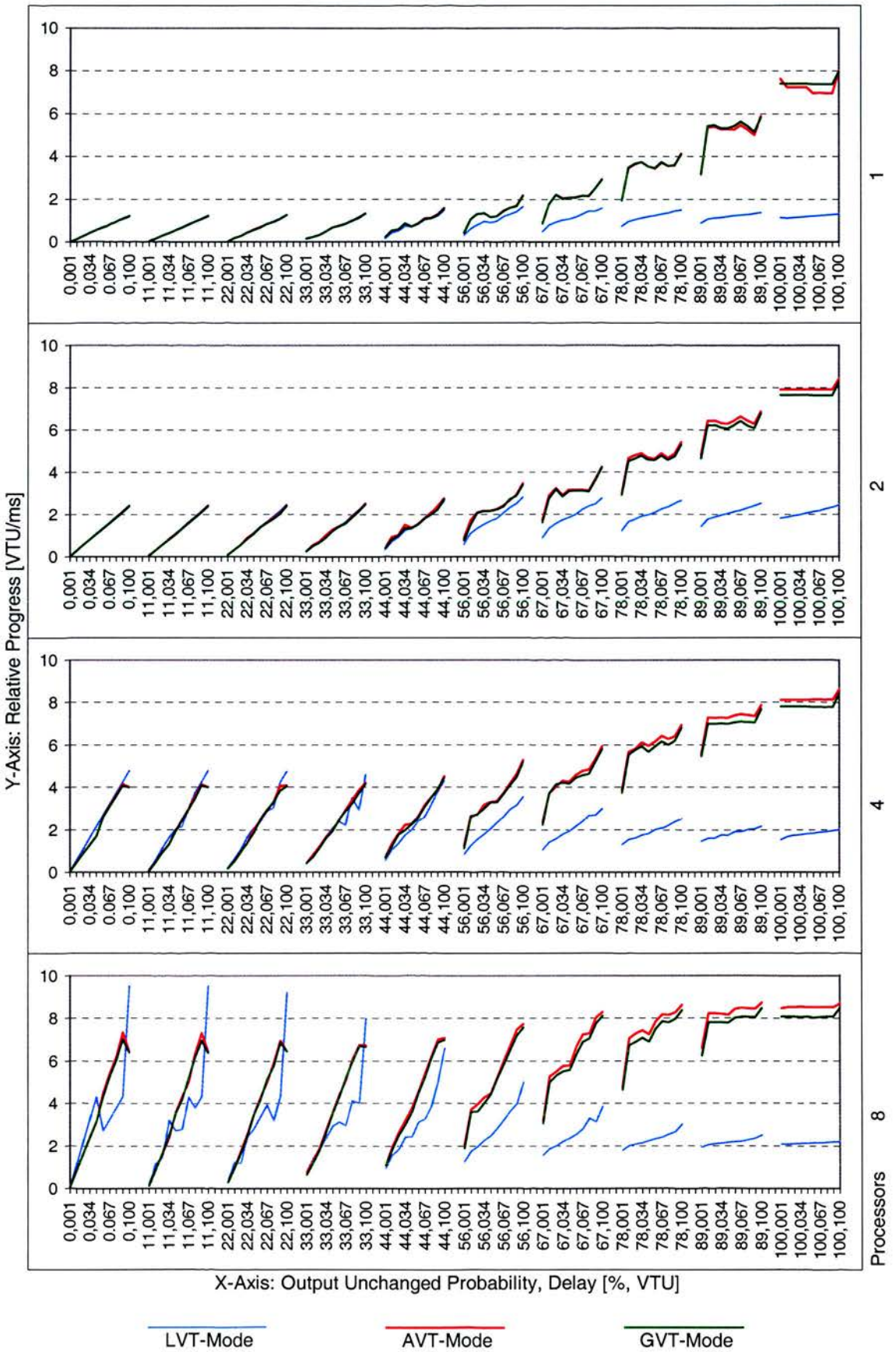


Figure 8.17: Model $Test_{4,2,4,2}$ Relative Progress versus Output Unchanged Probability and Delay on 1, 2, 4, 8 Processors with Event Processing Delay = $10000\mu s$

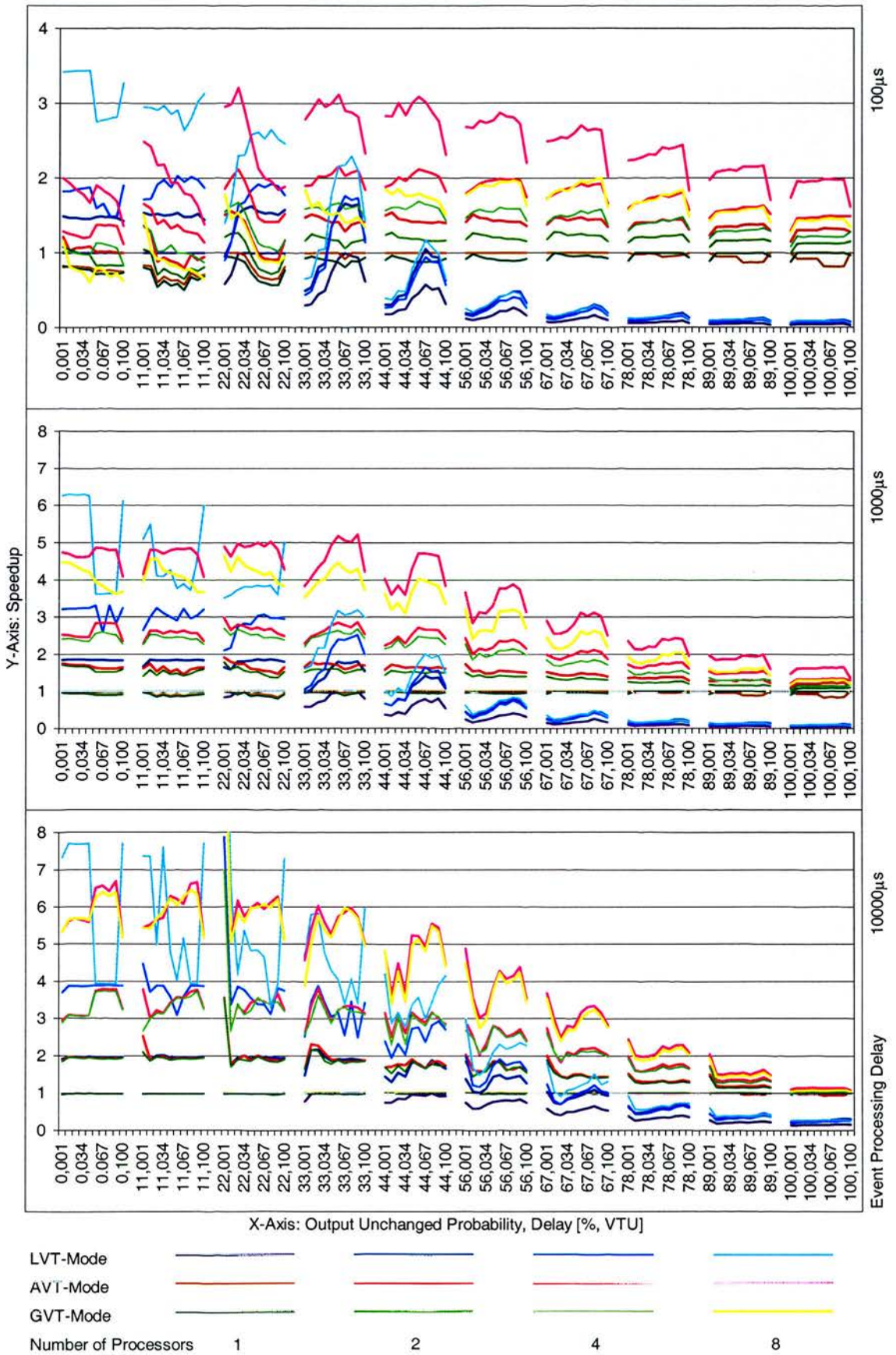


Figure 8.18: Model $Test_{4,2,4,2}$: speedup on 2, 4, and 8 processors relative to the best case on uniprocessor for Event Processing Delay = 100, 1000, 10000 μs

Event Processing Delay = $1000\mu\text{s}$ and *Event Processing Delay* = $10000\mu\text{s}$. This is caused by the high stability of the LP's output for high *Output Unchanged Probability*, which results in very few event-computations and reduces the amount of parallelism that can be exploited.

8.2.4 Model $Test_{4,3,4,4}$

Figure 8.19 shows the model $Test_{4,3,4,4}$, one with four connections per LP, and the graphs in Figures 8.20 – 8.22 depict the results for the same parameter sets used in Sections 8.2.2 and 8.2.3.

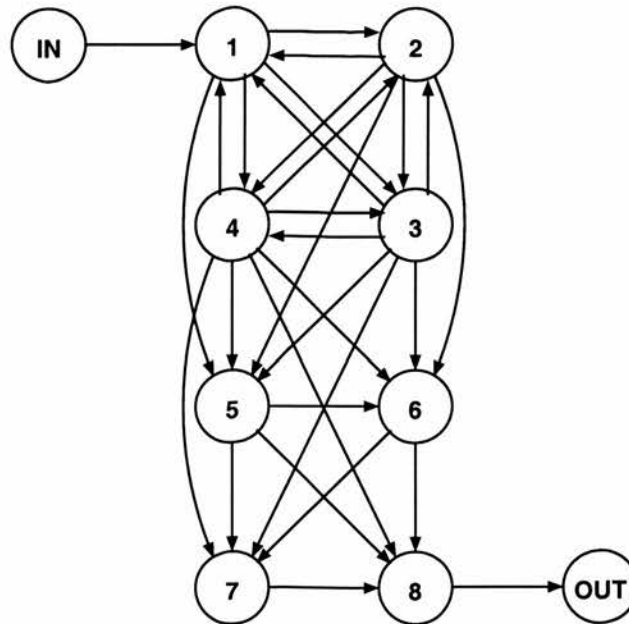


Figure 8.19: Model $Test_{4,3,4,4}$

These results confirm the trends in models $Test_{4,1,4,1}$ and $Test_{4,2,4,2}$: the performance of LVT-mode is reduced even further for higher values of *Output Unchanged Probability* due to the even higher number of connections on which null-messages must be sent for time-keeping purposes. AVT- and GVT-mode performances are more independent of *Delay*, with larger values of *Output Unchanged Probability* as optimism succeeds more often. The difference between the AVT- and GVT-mode is smaller than in the case of model $Test_{4,2,4,2}$ due to the larger number of null-messages sent by the LVT time-keeping mechanism used in the tightly-connected acyclic part of the model.

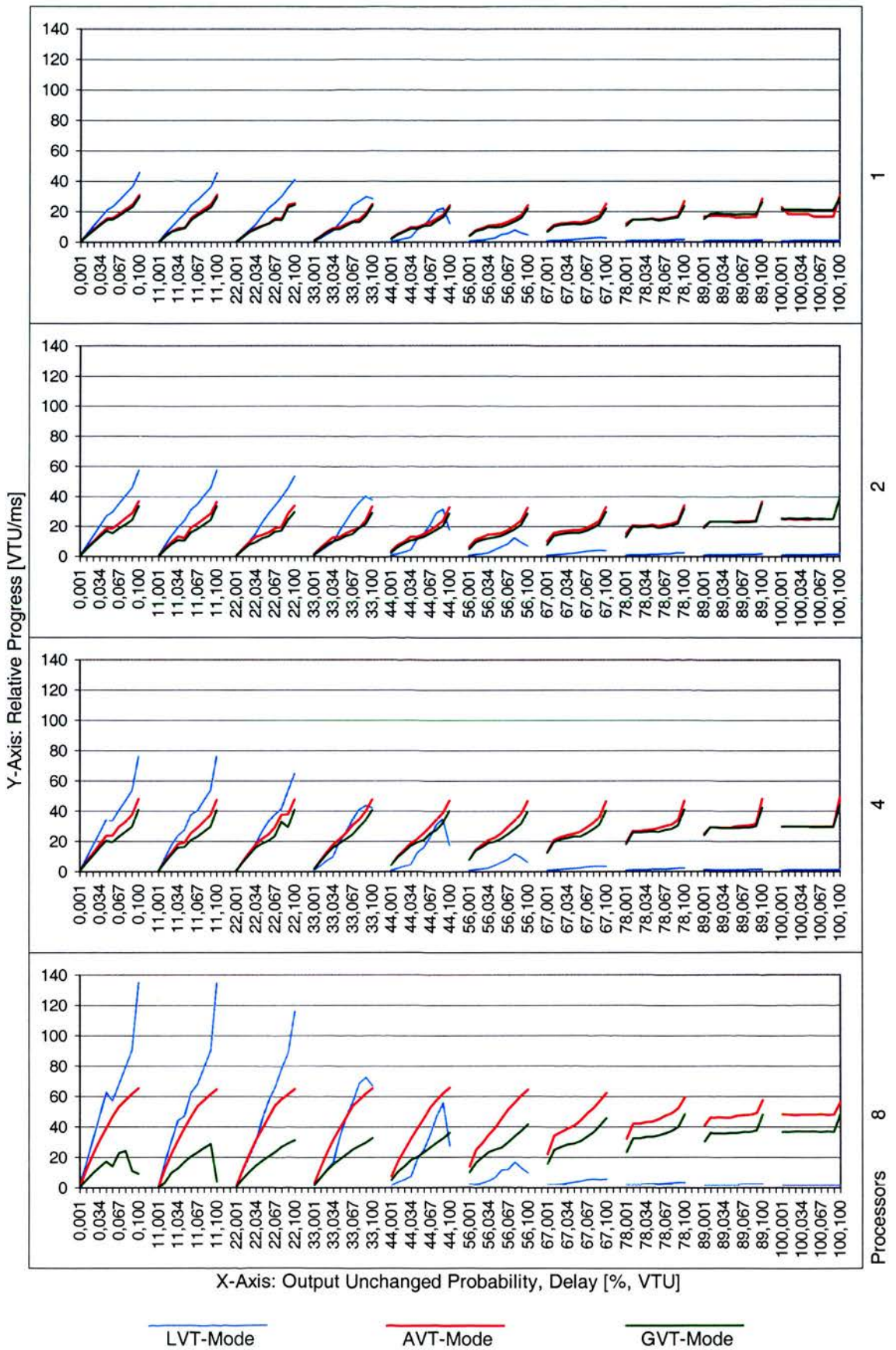


Figure 8.20: Model $Test_{4,3,4,4}$ Relative Progress versus Output Unchanged Probability and Delay on 1, 2, 4, 8 Processors with Event Processing Delay = $100\mu s$

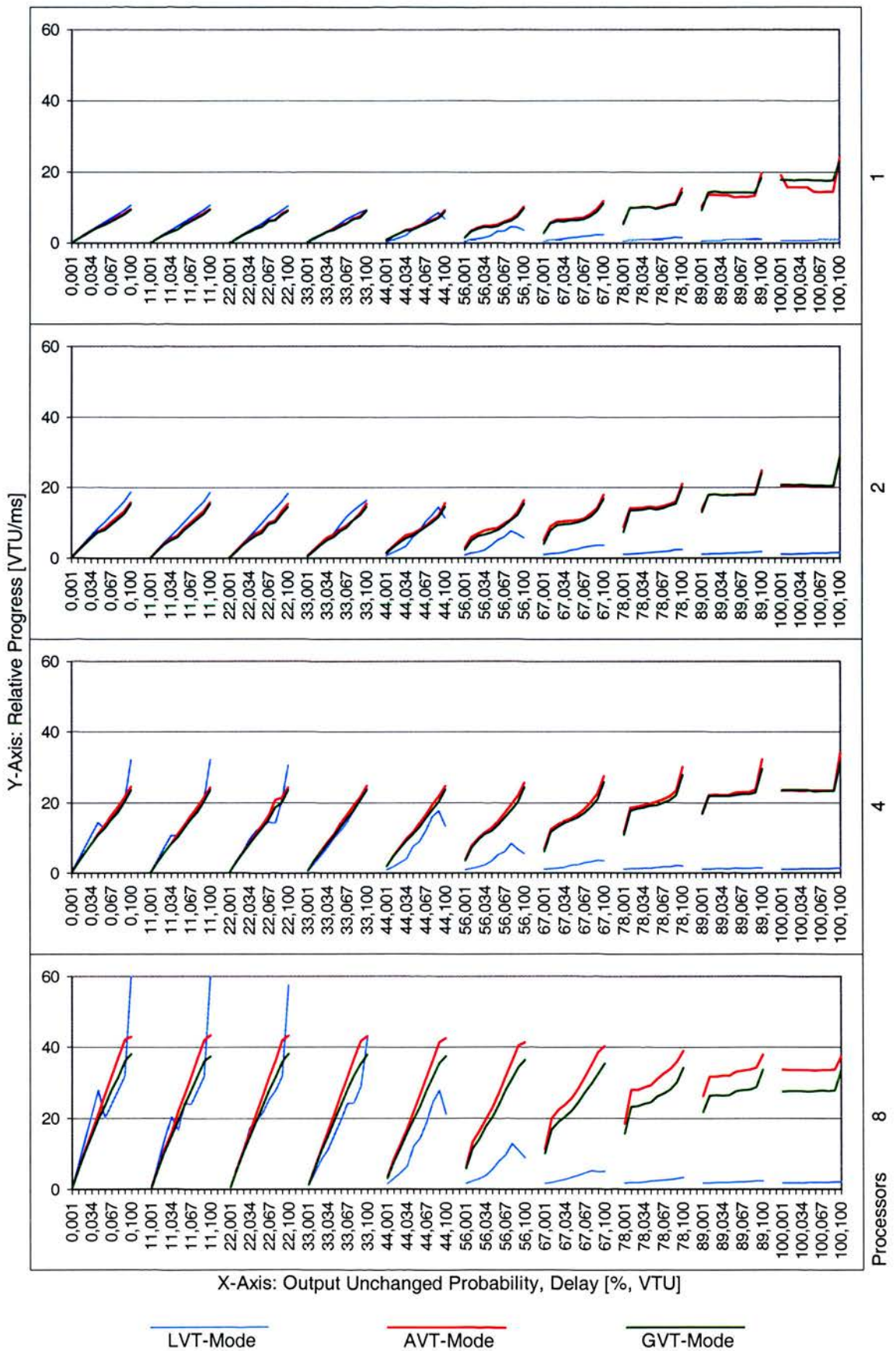


Figure 8.21: Model $Test_{4,3,4,4}$ Relative Progress versus Output Unchanged Probability and Delay on 1, 2, 4, 8 Processors with Event Processing Delay = $1000\mu s$

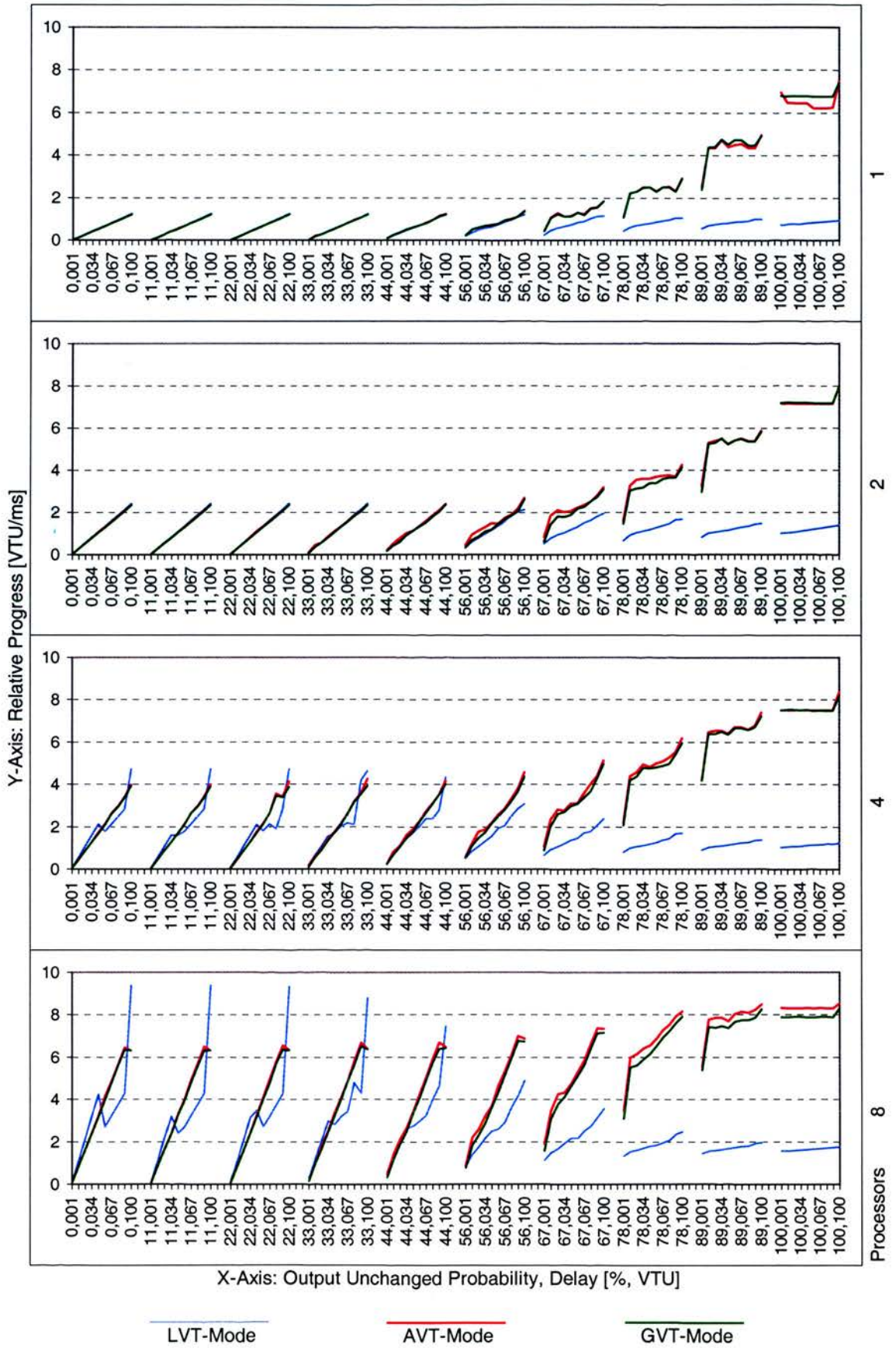


Figure 8.22: Model $Test_{4,3,4,4}$ Relative Progress versus Output Unchanged Probability and Delay on 1, 2, 4, 8 Processors with Event Processing Delay = 10000 μ s

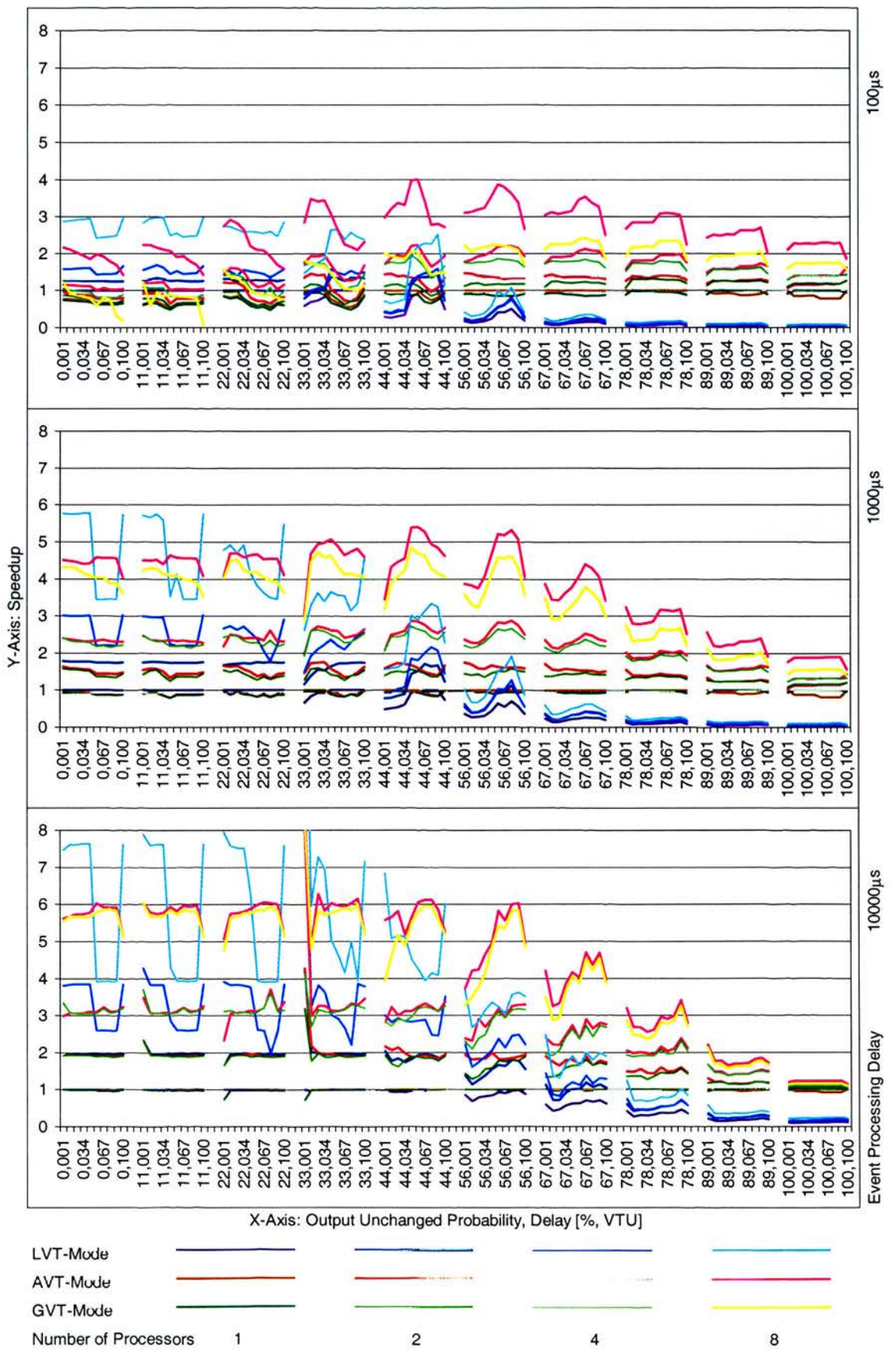


Figure 8.23: Model $Test_{4,3,4,4}$: speedup relative to the best case on uniprocessor on 2, 4, and 8 processors for *Event Processing Delay* = 100, 1000, 10000μs

In general, the performance of AVT-mode in model $Test_{4,3,4,4}$ is reduced in comparison to model $Test_{4,2,4,2}$ due to the large number of connections between LPs. As in the previous models, *Event Processing Delay* has a general effect of reducing the simulation progress. Overall, the AVT-mode performs best in 61% of the parameter sets on 2 processors, 75% on 4, and 76% on 8 processors.

Figure 8.23 summarises the speedup of LVT, AVT, and GVT-mode on 2, 4, and 8 processors. As in model $Test_{4,2,4,2}$ the LVT-mode results in the best speedup only for low *Output Unchanged Probability*. For higher values of *Output Unchanged Probability* AVT-mode is best, even for high *Delay*. Speedup decreases notably with an increase in *Output Unchanged Probability* especially for *Event Processing Delay* = 1000 μ s and *Event Processing Delay* = 10000 μ s. This is caused by the high stability of the LP's output which results in very few event-computations and reduces the amount of parallelism that can be exploited.

8.3 Summary

The results presented in this chapter demonstrate that for models with a mixture of cyclic and acyclic parts, the AVT-mode performs best for a large proportion of the parameter space, compared to the LVT- and GVT-modes. On average, the AVT-mode outperforms the LVT-mode in 70% of the distributed cases (Table 8.2), and performs better or just as well as the GVT-mode. The percentage of cases with better AVT-mode performance increases slightly as the number of processors increases. LVT-mode outperforms the AVT- and GVT-modes only for high values of *Delay*, and low values of *Output Unchanged Probability*, and low number of connections between LPs.

Model Name	Number of Processors		
	2	4	8
$Test_{2,1,2,2}$	66%	71%	–
$Test_{4,1,4,1}$	63%	65%	74%
$Test_{4,2,4,2}$	68%	72%	84%
$Test_{4,3,4,4}$	61%	75%	76%

Table 8.2: Model *Test* – percentage of AVT-mode outperforming LVT-mode

The performance of the LVT-mode depends on parameters such as the *Delay*, *Output*

Unchanged Probability, and the number of connections between LPs. The LVT-mode performance increases with the *Delay* and decreases with *Output Unchanged Probability* caused by the dependency of the lookahead on both the *Delay* and the *Output Unchanged Probability*: high *Delay* implies large lookahead while for high *Output Unchanged Probability* event-computations are replaced by null-message-processing which delivers little lookahead (= 1VTU).

In models with one connection per LP in the cyclic area (models *Test*_{2,1,2,2} and *Test*_{4,1,4,1}), the input-LP (LP_{IN} in Figures 8.5 and 8.9) sends a new event every 100VTU. This causes event-computations with *Delay* = 100, which infuses enough lookahead in the cycle to result in good progress for high *Delay* and high *Output Unchanged Probability*; this is slightly reduced in the cases of 4 and 8 processors, due to more null-messages being sent to remote processors.

However, in models with more than one connection per LP in the cyclic area (models *Test*_{4,2,4,2}, *Test*_{4,3,4,4}), the performance of the LVT-mode is dramatically reduced with *Output Unchanged Probability* increasing due to the larger number of connections on which LPs are depending for lookahead information which results in a large number of null-messages being sent for LVT time-keeping.

The AVT- and GVT-mode performance is more tolerant to *Delay* and the number of connections between LPs. The AVT- and GVT-modes outperform LVT-mode for higher values of *Output Unchanged Probability* and greater number of connections.

For low *Output Unchanged Probability*, AVT- and GVT-mode performance is similar to LVT-mode, i.e. it is dependent on lookahead. This is because for low *Output Unchanged Probability* the output of the LPs is very unpredictable, which prevents optimism from succeeding, and the AVT-algorithm progresses on LVT alone. In this case AVT- and GVT-mode performance is slightly reduced in comparison to LVT, because of the overhead of sending AVT-messages to the AVT-keeper and computing AVT itself.

With increasing *Output Unchanged Probability*, optimism succeeds more often and AVT improves on the LVT in each LP. This results in AVT-mode performance that is increasingly stable with respect to *Delay*, i.e. it is independent of lookahead, as the *Output Unchanged Probability* increases, which is reflected in the almost horizontal plot lines for high *Output Unchanged Probability*.

The performances of the AVT- and GVT-modes decrease slowly as the number of con-

nections between LPs increases. The AVT-mode outperforms the GVT-mode because the former reduces AVT computation and propagation cost in comparison with the latter. This difference in performance is reduced for the model with the highest number of connections between LPs (model *Test_{4,3,4,4}*) compared to the other two models of the same size as many more null-messages are sent by the LVT time-keeping mechanism used in the tightly-connected acyclic part of the model.

The *Event Processing Delay* has an overall effect of reducing the simulation progress. It also influences the difference in progress for parameter sets in which LVT-mode outperforms AVT-mode. This is reduced for higher *Event Processing Delay* as algorithmic overheads of the GVT time-keeping are eclipsed by the computation requirements for event-processing.

AVT-mode results in the best speedup except in cases when the *Output Unchanged Probability* is low or the *Delay* is very high. This speedup decreases notably with an increase in *Output Unchanged Probability*, due to the high stability of the LP's output. This in turn reduces the number of event-computations performed and therefore the amount of parallelism that can be exploited. Any remaining speedup for *Output Unchanged Probability* = 100 is an effect of distributing the computations performed by the synchronisation and time-keeping algorithm.

The results obtained from the implementation closely match the simulation results in Chapter 5. Any differences were due to the implementation version allowing variable *Delays*, while the simulated one assumes these to be fixed and both the scheduling and the AVT-computation costs were set to zero.

The results have demonstrated that the AVT-algorithm outperforms both the pure LVT and GVT time-keeping mechanisms in a majority of cases in the parameters sets.

Chapter 9

Conclusions

The thesis is based on two orthogonal ideas, that of the synchronisation policy, and the time-keeping mechanism for Distributed Discrete Event Simulation (DDES). The conservative and the optimistic synchronisation policies had previously been integrated, as exemplified in the Composite ELSA DDES algorithm [ArvindS92].

This thesis has proposed a new simulation scheme called the AVT-algorithm which for the first time combined the LVT and the GVT time-keeping scheme on a per-LP basis. Existing optimisations such as the adaptation of degree of optimism can be equally applied to the AVT-algorithm.

The design of the AVT-algorithm was informed by the observation that the overhead on the time-keeping mechanism is dependent to a large extent on the topology of the simulation models. Acyclic models are generally better suited to operate under LVT time-keeping schemes as the overhead due to null-messages is lower; whereas, cyclic models are usually more efficiently simulated under a GVT time-keeping mechanism, as the overhead due to messages for updating the global clock is lower.

The AVT is therefore appropriate for simulation graphs which contain a combination of cyclic and acyclic parts. The former is mapped to a GVT time-keeping mechanism and the latter to an LVT one which results in a network of virtual time regions. In these cases mapping the model entirely to either one of the time-keeping schemes would be less efficient for the respective parts of the simulation model.

The key contribution of the thesis is that it expands existing approaches to address inefficiencies in simulation algorithms by extending the integration of synchronisation

policies to incorporate two different time-keeping mechanisms.

Jha and Bagrodia [JhaB94] had proposed the idea of combining a Global Control Mechanism (GCM), based either on null-messages or GVT, and a Local Control Mechanism (LCM) which can be either conservative or optimistic. Each LP can be configured to use either LCM, and the GCM is used over the entire system. Unlike the AVT-algorithm, there is no notion of areas of virtual time, and individual LPs cannot be mapped to different time-keeping mechanisms.

Rajaei et al [RajaeiAT93] use Time Warp to synchronise between clusters of LPs, where each cluster is allocated to a processor and runs a sequential simulation algorithm. The AVT-algorithm is distributed over all available processors and does not distinguish between processes allocated on the same processor or a remote processor.

Avril and Tropper [AvrilT95] divide the simulation model into clusters each with its own Cluster Virtual Time (CVT). Each cluster has one designated input and output process in addition to the LPs of the cluster. The input process keeps track of the Input Virtual Time, and the output process records an Output Virtual Time. The output process will only send events to other clusters once the CVT reaches the timestamp of the respective events. A Conservative Time Window Algorithm is used for synchronisation between clusters. In contrast, the AVT-based approach does not have designated input/output nodes per cluster, and an LVT-based protocol is used between AVT regions, which can operate either conservatively or optimistically.

The AVT algorithm was first simulated which yielded promising preliminary results. It was then implemented in C++ with MPI on a distributed computer: a Beowulf cluster. The algorithm has been evaluated by means of a rigorous although not exhaustive search of the parameter space, using generic parameterised models.

The results of the implementation and the simulation closely matched. Any differences were due to the simulations assuming a fixed *Delay* and zero overhead for both the scheduling and the AVT computation.

The results have demonstrated that the AVT-algorithm outperforms both the LVT and the GVT for a majority of the parameters sets. On average, the AVT-algorithm outperforms the LVT time-keeping scheme in 70% of the distributed cases, and performs better or just as well as the GVT time-keeping.

The AVT-algorithm will also perform just as well as the LVT or the GVT for mod-

els which are completely acyclic or cyclic because in these cases all the LPs will be mapped to the LVT or GVT time-keeping mechanism, respectively.

Another important feature of the AVT-algorithm is that it is less sensitive to model and communication parameters which is desirable for any distributed algorithm.

The contribution of this work is significant as it extends DDES algorithms by combining and integrating time-keeping mechanisms in an elegant manner.

Future research on the AVT-algorithm could concentrate on dynamically adapting the time-keeping mechanism to simulation characteristics which vary during run-time in order to use the appropriate time-keeping mechanism for the parts of the parameter space where AVT is not better than LVT. The idea of adaptation has already been applied to the synchronisation policy, i.e. between conservative or optimistic. A logical extension would be to investigate an algorithm that is able to adapt both the time-keeping mechanism and the synchronisation policy.

The optimistic synchronisation policy could be applied to the LPs which employ an LVT time-keeping mechanism and which are conservative in the current implementation in order to allow for the same range of configuration options as available in Composite ELSA.

A further area of interest is to determine whether characterising simulation models on the basis of the stability of the LPs' output to changes in the input, i.e. *Output Unchanged Probability*, the *Delay*, and other simulation parameters, is useful for performance prediction. This raises further questions regarding the computation of the *Output Unchanged Probability* for all imaginable simulations, and furthermore, whether it is possible to compute a compound *Output Unchanged Probability a priori*, and whether this can be used to select the optimal configuration of the AVT algorithm before the start of the simulation. Alternatively, the time-keeping scheme may be adapted at run-time based on observed and expected *Output Unchanged Probability* and *Delay*. A different approach would be to use the self-simulating capability of AVTSIM for performance estimation instead of performance prediction.

Finally, the AVT-algorithm could be ported to other platforms, such as a large multiprocessor computer, e.g. the Cray T3E, in order to investigate how the algorithm would scale on a massively parallel architecture; and a network of PCs running Microsoft Windows, given that a suitable implementation of MPI has become available.

Appendix A

DS-RT 2001: Area Virtual Time

D. K. Arvind and J. Schneiders. Area Virtual Time. In *Proceedings 5th IEEE International Workshop on Distributed Simulation and Real Time Applications*, Cincinnati, Ohio, USA, Pages 105–112, August 2001.

Area Virtual Time

D. K. Arvind and J. Schneiders

Institute for Computing Systems Architecture, Division of Informatics,
The University of Edinburgh, Mayfield Road, Edinburgh EH9 3JZ, SCOTLAND.

Email: dka|josch@dcs.ed.ac.uk

Abstract

We present a novel synchronisation algorithm for distributed discrete-event simulation (DDES), called the Area Virtual Time (AVT) algorithm. We first expose two orthogonal ideas of the synchronisation policy for DDES, which is either conservative or optimistic, and the time-keeping mechanism, which is based on either Local or Global Virtual Times. The AVT algorithm is based on a network of virtual time regions, which is a happy medium between the Local Virtual Time (LVT) and the Global Virtual Time (GVT). The AVT algorithm permits the different parts of the simulation model to run either under LVT or GVT time-keeping mechanisms. This is particularly suited to models which are less than homogeneous. In those cases, mapping the models entirely to either one of the time-keeping schemes would not be efficient; or, the real-time nature of the interfaces precludes the use of GVT in those parts of the model. Our results demonstrate that the AVT algorithm progresses the simulation times faster than either the LVT or the GVT schemes, and is less sensitive to variations in some key model and communication parameters - a desirable property in distributed computation.

1. Introduction

The simulation model can be represented as a directed graph, in which nodes and arcs correspond to logical processes (LP) and communication channels, respectively. The LPs exchange information in the form of messages containing time-stamped data which are termed as *events*.

Distributed Discrete Event Simulation (DDES) can run either under a conservative or an optimistic synchronisation policy. The policy determines the manner in which simulation time is progressed. *Conservative* DDES algorithms observe the strict partial ordering

in the sequence of evaluation, i.e. result at a node is deemed safe to commit if, and only if, no message with a timestamp less than the local time will ever arrive at that node. This condition is relaxed in *Optimistic* DDES algorithms, in which events can be processed at the nodes in advance of their arrival. Errors due to mis-speculation are detected should they occur, and the algorithm recovers by rolling back in time to a consistent state, and then proceeding with the simulation. The time-keeping mechanism is orthogonal to the synchronisation policy. Time is maintained in DDES in one of two ways: each LP maintains a local clock, called the *Local Virtual Time (LVT)*, based on the timestamps of incoming events; or, the algorithm maintains a *Global Virtual Time (GVT)*, which is the minimum time-stamp of all the incoming events and those in transit. An increment in GVT is notified to all the LPs, which update their respective local clocks.

Optimistic DDES algorithms such as Time Warp [JeffersonS85] use a GVT-based time-keeping mechanism, whereas conservative ones based on the Chandy-Misra-Bryant algorithms [ChandyM79, Bryant81] use LVT-based time-keeping schemes. The Composite ELSA algorithm [ArvindS92] is an exception to this rule, since it integrates both the conservative and optimistic synchronisation policies, but uses only an LVT-based time-keeping mechanism.

We propose a novel algorithm for DDES called the Area Virtual Time (AVT) algorithm which supports both conservative and optimistic synchronisation policies, and is based on a combination of LVT and GVT time-keeping mechanisms. This is a happy medium between the extremes of either each LP maintaining its LVT, or calculating the GVT over all the LPs in the simulation model.

In the AVT algorithm, a set of LPs is mapped to a Virtual Time Area (VT-Area), which gives rise to a network of VT-Areas. The choice of mapping a particular set of LPs to a VT-Area is determined by the topology of the simulation graph. In short, cyclic sub-

graphs are mapped to VT-Areas and the rationale for this is next explained.

The following observations are made regarding the influence of the synchronisation policy on the performance of DDES. Conservative algorithms, irrespective of the time-keeping mechanisms, generally have lower run-time overheads, but their performance is dependent on the *lookahead*, i.e., the time into the future that a value is known to hold. Optimistic algorithms which use either LVT or GVT time-keeping mechanisms, can potentially infuse more concurrency at run-time by relaxing the strict ordering of evaluation, but at a price: the overheads of completion of non-preemptable tasks and saving of states can attenuate this gain.

The influence, albeit simplified, of the time-keeping mechanism on the performance of DDES is as follows: in acyclic graphs, LVT-based time-keeping mechanisms are more efficient due to the lower communication overhead; and in cyclic graphs, the GVT-based time-keeping schemes are more efficient as they are less prone to fragmenting events. In more detail: GVT-based time-keeping mechanism coupled with an optimistic synchronisation policy (assuming good predictability) is better suited for cyclic graphs. On the other hand, the LVT-based time-keeping mechanisms are unable to exploit predictability due to event fragmentation. For instance, for an event received at time, t_{now} , at an LP, a committed result needs to be passed round the cycle, before another result can be committed for time t_{next} . The amount of fragmentation depends on the relationship between the inter-arrival time for events and their execution time in an LP. As in acyclic graphs, event fragmentation does not hamper LVT-based time-keeping mechanisms as they are better suited due to their lower communication overheads.

2. Related Work

Jha and Bagrodia [JhaB94] had proposed the idea of a combination of a Global Control Mechanism (GCM), which is based either on null messages or GVT, and a Local Control Mechanism (LCM) which can be either conservative or optimistic. Each LP can be configured to use either LCM, and the GCM is used over the entire system. Unlike the AVT algorithm presented in this paper, there is no notion of multiple areas of virtual time. Hammes and Tripathi [HammesT94] adapt the degree of optimism on a per-LP and per-channel basis and use GVT as the only Global Control Mechanism. Wood and Turner [WoodT94] address the inefficiency of using a null-message time-keeping mechanism for models with cycles, in which the LPs use a local clock and are conservative. In order to distinguish which

outputs should transmit carrier-null-messages (as opposed to ordinary null-messages), the cyclic structure of the graph is determined before run-time, and the cyclic and non-cyclic outputs are marked.

Rajaei et al [RajaeiAT93] use Time Warp to synchronise between clusters of LPs, where each cluster is allocated to a processor and runs a sequential simulation algorithm. Avril and Tropper [AvrilT95] divide the simulation model into clusters each with its own Cluster Virtual Time (CVT). Each cluster has one designated input and output process in addition to the LPs of the cluster. The input process keeps track of the Input Virtual Time (IVT), and the output process records an Output Virtual Time (OVT). The output process will only send events to other clusters once the CVT reaches the timestamp of the respective events. A conservative time window (CTW) algorithm is used for synchronisation between clusters.

In contrast, the AVT-based approach does not have designated input/output nodes per cluster, and an LVT-based protocol is used between AVT regions, which can operate either conservatively or optimistically.

3. The AVT Synchronisation Algorithm

Each LP in the AVT algorithm can be configured to use any combination of the modes of synchronisation mentioned in the previous section. In the current instantiation of the algorithm, a GVT node has bounded optimism and the LVT node is conservative. However, the GVT and LVT nodes could just as well have either limited or unlimited optimism [ArvindS92].

Each cyclic area of the simulation graph is assigned to a Virtual Time Area (VT-Area), in which a GVT-style time-keeping scheme is used.

3.1. Definitions

LVT: Local Virtual Time – the time up to when the inputs are committed.

LOT: Local Optimistic Time – the time up to when an LP has speculated based on its input

Event-Message is of the form:

$(t_{start}, t_{end}, t_{end_{opt}}, data)$, where:

t_{start} is the start-time of an event

t_{end} is the time until when the event is committed to hold

$t_{end_{opt}}$ is the time until when an event is presumed to hold based on the input

$data$ is the value of the event.

Event-messages can be either: *committed*, i.e., $t_{start} < t_{end}$, and $t_{end} = t_{end_{opt}}$; *uncommitted*, i.e., $t_{start} = t_{end}$, and $t_{end} < t_{end_{opt}}$; or *partially committed*, i.e., $t_{start} < t_{end} < t_{end_{opt}}$.

AVT: Area Virtual Time is defined as the minimum LOT of the members of the VT-Area and the minimum ($t_{end_{opt}}$) of all the messages in transit in the VT-Area.

AVT-Message is used to communicate between the LPs and the AVT-manager, and is of the form (*LVT*, *AVT*) and defined as:

LVT – the sending LP’s LVT, or empty when sent by the AVT-manager

AVT – the sending LP’s LOT, or the updated AVT sent by the AVT-manager

Delay (δ) – the virtual execution time of an LP.






$\min(t_{end}^{in})$ – denotes the minimum t_{end} of all inputs to an LP

$\min(t_{end}^{in_{LVT}})$ – denotes the minimum t_{end} of all inputs to an LP from predecessors outside the VT-Area.

$\min(t_{end}^{in_{AVT}})$ – denotes the minimum t_{end} of all inputs to an LP from predecessors inside the VT-Area.

Predecessor (Successor) of an LP, *A*, is any other LP from which *A* receives its input (to which *A* sends its output).

Graphic Symbols: The height of a box corresponds to the interval between t_{start} and t_{end} (or $t_{end_{opt}}$).

	Committed part of event-message
	Uncommitted part of event-message
	AVT-message
	Input different from previously received/guessed
	Discarded state

3.2. Description of the AVT Algorithm

The instantiation of the AVT algorithm, as described in this section, assumes fixed connections between the LPs, and that the results of the evaluation are committed before event-messages are sent to the neighbouring LPs.

The format of the event-messages are similar to those used by Composite ELSA, i.e., a time interval defines the period for which a value is guaranteed to hold. The AVT messages have an additional time-stamp to denote the time until when the value is presumed to hold optimistically. The LVT nodes employ a conservative synchronisation policy, while the GVT nodes in the VT-Area employ an optimistic one. The optimism is limited by the look-ahead in the event-messages from

outwith the VT-Area. When viewed from beyond, a VT-Area behaves as if it were a conservative LVT node, albeit a conglomerate one. The VT-Areas never emit an uncommitted event-message, although this is not a fundamental restriction: nodes in the VT-Areas (and indeed any LVT node) can range from conservative, to locally or globally optimistic - similar to nodes in Composite ELSA.

The AVT algorithm can be applied to LPs which can be synchronised either locally (LVT-node) or globally (GVT-node), or both (hybrid-node). Consider the logical processes in Figure 1

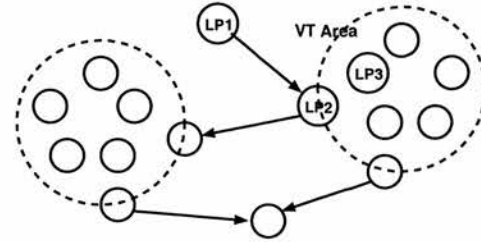


Figure 1. Simulation Graph

LP1 is an LVT-node. It can either be a conservative node (corresponding to a CMB or conservative ELSA node), or an optimistic node (corresponding to an optimistic ELSA node).

LP2 is a hybrid-node, i.e. it behaves both as an LVT- and as a GVT-node.

LP3 is a GVT-node. It can either execute either optimistically or conservatively.

3.2.1 The LVT-node

In the instantiation of the AVT algorithm described in this paper, the LVT nodes synchronise conservatively. The node is ready to progress its time once all the end-times of the input event are greater than its LVT, i.e., $\min(t_{end}^{in}) > LVT$ (Figure 2). This node sends the following event to its successors:

$$(t_{start} = LVT + \delta, t_{end} = \min(t_{end}^{in}) + \delta, t_{end_{opt}} = t_{end}, f(input))$$

Its LVT is advanced to the minimum of the end-times of the input (i.e., $\min(t_{end}^{in})$), and any state prior to the LVT is discarded.

3.2.2 The GVT-node

A GVT-node is ready to compute once all its input-queues contain unprocessed events (Figure 3), i.e.,

$$\min(t_{end}^{in_{AVT}}) > LVT \vee \min(t_{end_{opt}}^{in}) > LOT$$

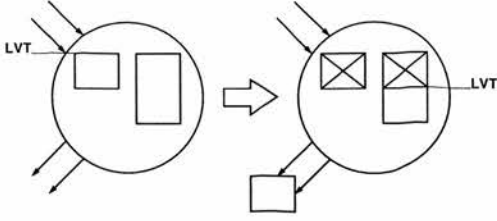


Figure 2. LVT-node in conservative mode

The following event is sent to its successors in the VT-Area:

$$(t_{start} = LVT + \delta, t_{end} = \min(t_{end}^{in}) + \delta, \\ t_{end_{opt}} = \min(t_{end_{opt}}^{in}), f(input))$$

The LVT is advanced to $\min(t_{end}^{in})$, the LOT to $\min(t_{end_{opt}}^{in})$, and an update AVT-message, (LVT, LOT) , is sent to the AVT-manager.

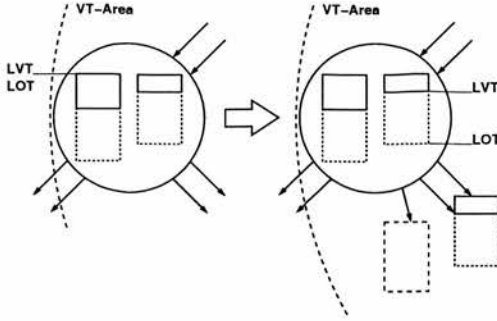


Figure 3. Computing and updating AVT in a GVT-node

A GVT-node is ready to progress conservatively if the AVT has been updated (Figure 4). The following event is sent to all its successors outside the VT-Area:

$$(t_{start} = LVT + \delta, t_{end} = AVT + \delta, \\ t_{end_{opt}} = t_{end}, f(input))$$

LVT is advanced to AVT, and all the states before the LVT are discarded.

3.2.3 The Hybrid-node

Hybrid nodes have split personalities: they appear as GVT/optimistic nodes to those inside the VT-Area, and as LVT/conservative nodes to those outside. When the committed inputs from outwith the VT-area are available, the hybrid node guesses the values for others from within the VT-area, evaluates based on all the inputs (both guessed and committed), and fires optimistically, but only to its neighbours within the VT-area.

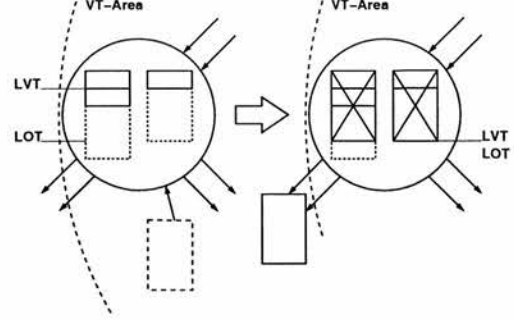


Figure 4. GVT-node progressing conservatively

The dual behaviour of the hybrid-node is described as follows:

- Once all the input-queues to the LVT part of the hybrid-node contain events, i.e.,

$$\min(t_{end}^{in_{LVT}}) > \min(t_{end}^{in_{AVT}}) \wedge \\ \min(t_{end}^{in_{LVT}}) > LOT$$

the LP fires optimistically (Figure 5). The following event is generated, which is stored in the LP's output-queue, and is sent to the LP's successors in the VT-Area.

$$(t_{start} = LVT + \delta, t_{end} = \min(t_{end}^{in}) + \delta), \\ t_{end_{opt}} = \min(t_{end}^{in_{LVT}}), f(input))$$

The LOT is advanced to $\min(t_{end}^{in_{LVT}})$.

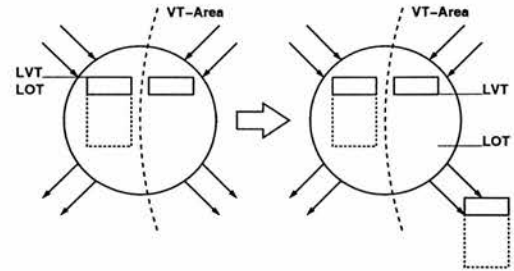


Figure 5. Speculation in a hybrid-node

- An input received from within the VT-Area is checked against the speculated value. In the event of a mis-speculation, the LP is rolled back to a safe state and the correct result is resent to its successors within the VT-Area. Otherwise, an update AVT-message is sent to the AVT-manager. This is described in more detail below:

1. When the speculation was correct:

- (a) The input is committed up to the LOT: $\min(t_{end}^{in}) \geq LOT$ results in a committed output being sent to the successors outside the VT-Area; the LVT is advanced to $\min(t_{end}^{in})$; and the AVT-message, (LVT, LVT) , is sent to the AVT-Manager.
- (b) If the input is *not* committed up to LOT, i.e. $\min(t_{end}^{in}) < LOT$, then an update AVT-message, (LVT, LOT) , is sent to the AVT-manager (Figure 6).

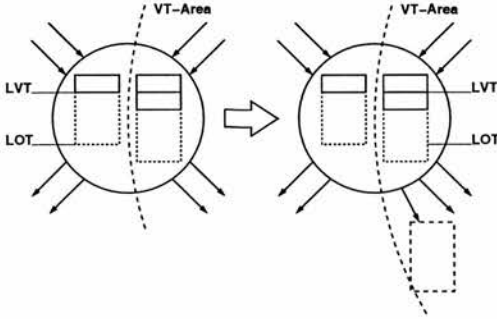


Figure 6. Update in a hybrid-node

2. Should the received input be different from the speculated one, then the LOT is rolled back to LVT, or the result is computed if it had not already been done so, and compared with the speculated one. The following three cases arise:

- (a) If there is no previous result, then the following event is sent to the successors:

$$\begin{aligned} t_{start} &= LVT + \delta, \\ t_{end} &= \min(t_{end}^{in}) + \delta, \\ t_{end_{opt}} &= \min(t_{end}^{in_{LVT}}) + \delta, \\ &f(input) \end{aligned}$$

and, the following update AVT-message is sent to the AVT-manager:

$$(LVT = LVT, AVT = \min(t_{end}^{in_{LVT}}))$$

- (b) If the recomputed result is unchanged, then the same event and AVT-message (as in Case 1) is sent.
- (c) Should the result differ from the one propagated (Figure 7), then the correct result is sent to its successors in the VT-Area, in the following event:

$$\begin{aligned} t_{start} &= LVT + \delta, \\ t_{end} &= \min(t_{end}^{in}) + \delta, \\ t_{end_{opt}} &= \min(t_{end}^{in_{LVT}}) + \delta, \\ &f(input) \end{aligned}$$

The following event is sent to the successors outside the VT-Area, with the LVT being advanced to $\min(t_{end}^{in})$, and any state before LVT being discarded:

$$\begin{aligned} t_{start} &= LVT + \delta, \\ t_{end} &= \min(t_{end}^{in}) + \delta, \\ t_{end_{opt}} &= t_{end}, \\ &f(input) \end{aligned}$$

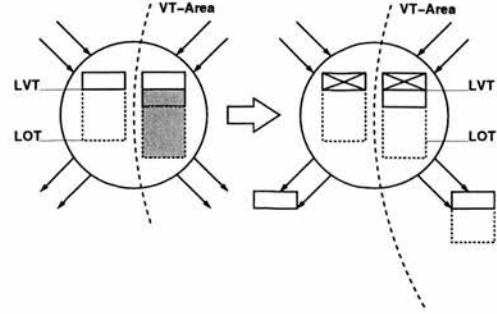


Figure 7. Roll-back in a hybrid-node

- Once an LP receives an update AVT-message from the AVT-manager, it fires conservatively (Figure 8). The committed result is sent to all the successors outside the VT-Area.

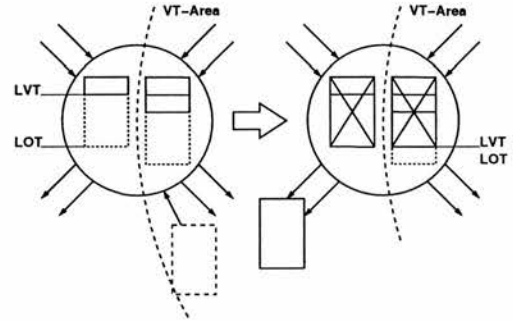


Figure 8. Hybrid-node progression based on AVT

A hybrid node progresses its LVT based on the AVT when the following conditions hold: if the updated AVT is larger than the minimum end-time of all the committed inputs, i.e., $AVT > \min(t_{end}^{in})$, then the following event is sent to its successors outside the VT-Area

$$\begin{aligned} t_{start} &= LVT + \delta, t_{end} = AVT + \delta, \\ t_{end_{opt}} &= t_{end}, f(input) \end{aligned}$$

The LVT is advanced to the AVT, and any state before the LVT is discarded.

3.2.4 The AVT-manager

The behaviour of the AVT-manager is next summarised.

The AVT-manager computes the AVT as the minimum of all the estimates received by the GVT and hybrid nodes in the VT-Area, and the minimum, committed time-stamp of any event-message in transit within the VT-area. If AVT can be advanced, then the AVT-manager sends the updated AVT to all the nodes in the VT-area. All events up to the AVT are then committed and sent to neighbours outwith the VT-area. The LVT is brought forward to the AVT and all state information before the AVT is discarded.

When the AVT-manager receives an AVT-message, (*LVT*, *LOT*), from an LP within its assigned VT-Area, then it is stored and the AVT is re-estimated as the minimum of the *LOT* of all the LPs in the VT-Area and $t_{end_{opt}}$ of all events in transit within the VT-Area. If the new estimate is greater than the previous one, then the AVT-manager notifies all the LPs in the VT-Area whose LVT is less than the AVT.

4. The Simulation Environment

The behaviour of the AVT algorithm was modelled in C++ and simulated in a sequential event-driven simulator (also written in C++). The simulator also models a distributed system connected by a network. All the significant operations, such as computing results and sending event messages, are assigned costs.

The network model assumes a fixed cost and a minimum fixed delay for each message. The *bandwidth* of the network determines the number of messages that can be transmitted over the network at any time. Messages are delayed until the network bandwidth becomes available. Thus the network model neither takes into account the relative sizes of messages nor the effects of thrashing, although congestion is accounted for.

We distinguish between two notions of virtual time. The virtual time according to the AVT algorithm is called *virtual time*. The virtual time according to the simulated distributed system on which the AVT algorithm is run is referred to as *simulated real time*.

The parameters of interest are:

System Parameters

Number of Processors

Communication Bandwidth This determines the number of messages the network can transmit at any time. Bandwidth can be made unlimited, in which case every message is delayed by a fixed amount of *simulated real time*.

Event Propagation Delay The amount of *simulated real time* required to send an event from one processor to another.

AVT Propagation Delay The number of units of *simulated real time* needed to send an AVT-message.

Input Delay The *simulated real time* required to read an input value from an external source.

End of *simulated real time* The number of units of *simulated real time* for which the distributed system is simulated.

Algorithm Parameters

Mode One parameter each from the two sets - {*LVT*, *AVT*, *GVT*} and {*Conservative*, *Optimistic*} - is selected.

AVT Processing Delay The amount of *simulated real time* needed to process an AVT update at the AVT-manager.

Model Parameters

Event Processing Delay The amount of *simulated real time* needed to compute a result in an LP.

Input Interval The amount of virtual time between the arrival of two input events to the model.

Delay The virtual time between receiving an event in an LP and producing an output event.

Output Change Probability A value between 0% and 100% which gives the probability of the output of an LP changing should one of the inputs change

5. Results

The benchmark under simulation has cyclic (2 LPs) and acyclic parts (3 LPs). The model was run under all the different combinations of the following parameters:

Number of Processors {1, 2, 5}

Communication Bandwidth 1

Event Propagation Delay {1, 3, 10, 32, 100} units of *simulated real time*

AVT Propagation Delay Same as *Event Propagation Delay*

Input Delay 0 units of *simulated real time*

End of *Simulated Real Time* 10000 units

Mode {*LVT*, *AVT*, *GVT*} and *Conservative* (i.e., the VT-area as a unit operates conservatively although the LPs within the VT-area may progress optimistically).

AVT Processing Delay 0 units of *simulated real time*

Event Processing Delay 100 units of *simulated real time*

Input Interval 100 units of virtual time.

Delay {1, 3, 10, 32, 100} units of virtual time.

Output Change Probability {0%, 25%, 50%, 75%, 100%}

The graph in Figure 9 compares the progress in Virtual Time for the three time-keeping schemes (y-axis) while changing the simulation parameters along four dimensions - Delay, Event Propagation Delay (denoted as "Message Delay" in the graph), Number of Processors, and Output Change Probability ("Change Prob." in the graph). In the two-processor case, the cyclic part was mapped to one processor and the acyclic part to the other. The graphs in Figures 10–13 show details of Figure 9.

As the value of the *Delay* is incremented, the progress of the virtual time improves, and in the extreme, to a spike, when the *Delay* is the same as the *Input Interval*. In this case the lookahead is right up to the arrival of the next event, resulting in no event fragmentation.

We also observe that in the case of mapping to two processors, the performance of the AVT algorithm is more stable over changes to the parameters, compared to the GVT time-keeping scheme, which is hit by network delays, and to the LVT time-keeping mechanism, which only performs well in the cases of good look-ahead (aka Delay (δ)). In the mapping to five processors, it is evident that the AVT algorithm degrades more gracefully with increases in network delays.

The AVT algorithm maintains this superiority over the other two time-keeping mechanisms for different *Output Change Probability* values.

The results demonstrate that the AVT algorithm performs better or just as well as the other two mechanisms for simulation models containing a mixture of cyclic and acyclic parts.

6. Conclusions and Future Work

We have proposed a novel synchronisation scheme called the Area Virtual Time (AVT) algorithm which uses the apt time-keeping mechanism for simulation graphs containing a combination of cyclic and acyclic parts.

The cyclic part of the graph is mapped to a GVT-based time-keeping mechanism, and the acyclic part to an LVT-based time-keeping mechanism.

Our results demonstrate that the AVT algorithm progresses simulation time more efficiently than either the

GVT- or LVT-based time-keeping schemes, and the algorithm is less sensitive to variations in key model and communication parameters - a desirable property in a distributed computation.

The AVT algorithm is currently being extended to simulate efficiently simulation graphs in which connections between nodes change dynamically.

References

- [ArvindS92] D. K. Arvind and C. Smart. Hierarchical Parallel Discrete Event Simulation in Composite ELSA. In *6th Workshop on Parallel and Distributed Simulation (PADS92)*, Pages 147–156, 1992.
- [AvrilT95] H. Avril and C. Tropper. Clustered time warp and logic simulation. *Proceedings of the 9th workshop on Parallel and distributed simulation*, 1995, Pages 112–119
- [Bryant81] R. E. Bryant. A switchlevel model and simulator for MOS digital systems. *IEEE Transactions on Computers*, C-33(2):160177, Feb. 1981.
- [ChandyM79] K. Chandy and J. Misra. Distributed Simulation: A Case Study in Design and Verification of Distributed Programs. *IEEE Transactions on Software Engineering*, SE-5(5):440–452, 1979.
- [HamnesT94] Donald O. Hamnes and Anand Tripathi. Investigations in adaptive distributed simulation. *Proceedings of the 1994 Workshop on Parallel and Distributed Simulation*, 1994, Pages 20–23
- [JeffersonS85] D. Jefferson and H. Sowizral. Fast Concurrent Simulation Using the Time Warp Mechanism. In *Proceedings of the Conference on Distributed Simulation*, pages 63–69, July 1985.
- [JhaB94] Vikas Jha and Rajive L. Bagrodia. A unified framework for conservative and optimistic distributed simulation. *Proceedings of the 1994 Workshop on Parallel and Distributed Simulation*, 1994, Pages 12–19
- [RajaeiAT93] Hassan Rajaei, Rassul Ayani and Lars-Erik Thorelli. The local Time Warp approach to parallel simulation. *Proceedings of the 1993 Workshop on Parallel and Distributed Simulation*, 1993, Pages 119–126
- [WoodT94] Kenneth R. Wood and Stephen J. Turner. A generalized carrier-null method for conservative parallel simulation. *Proceedings of the 1994 Workshop on Parallel and Distributed Simulation*, 1994, Pages 50–57

Abbreviations in Figures 10–13:

EPD = Event Propagation Delay (message delay)

NP = Number of Processors

OCP = Output Change Probability

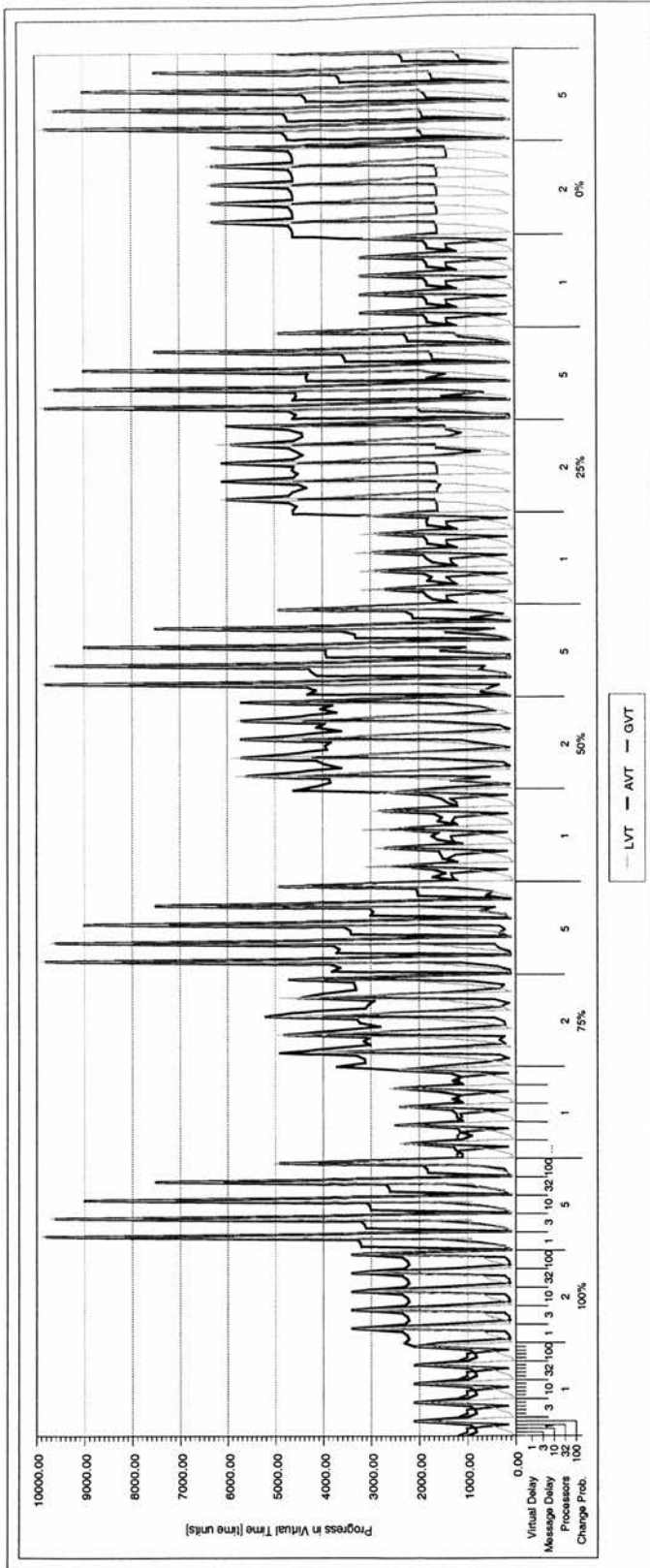


Figure 9. Progress of Virtual Time for 10000 Units of Simulated Real Time

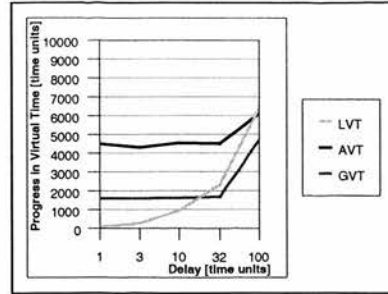


Figure 10. Progress for Delay=1, 3, 10, 32, 100; EPD=1; NP=2; OCP=75%

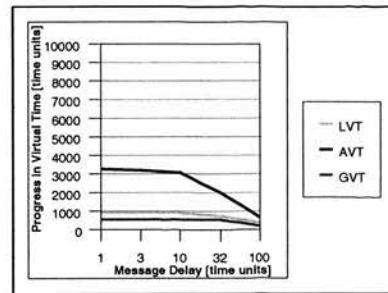


Figure 11. Progress for Delay=32; EPD=1, 3, 10, 32, 100; NP=5; OCP=0%

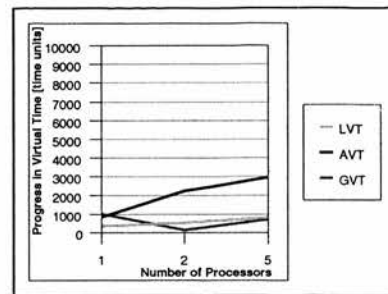


Figure 12. Progress for Delay=10; EPD=10; NP=1, 2, 5; OCP=0%

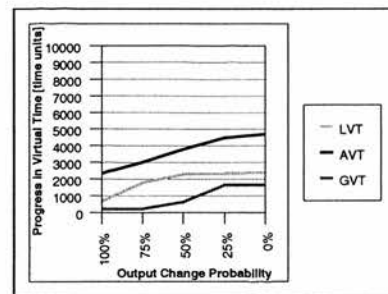


Figure 13. Progress for Delay=32; EPD=32; NP=2; OCP=0%, 25%, 50%, 75%, 100%

Bibliography

- [ArvindS92] D. K. Arvind and C. Smart.
Hierarchical Parallel Discrete Event Simulation in Composite ELSA.
In *Proceedings of the 6th Workshop on Parallel and Distributed Simulation*, Pages 147–156, 1992.
- [ArvindS2001] D. K. Arvind and J. Schneiders.
Area Virtual Time.
In *Proceedings 5th IEEE International Workshop on Distributed Simulation and Real Time Applications*, Cincinnati, Ohio, USA, Pages 105–112, August 2001.
- [AvrilT95] H. Avril and C. Tropper.
Clustered Time Warp and Logic Simulation.
Proceedings of the 9th Workshop on Parallel and Distributed Simulation, Pages 112–119, 1995.
- [BauerSK91] H. Bauer, C. Sporrer, and T. H. Krodel.
On Distributed Logic Simulation Using Time Warp.
In A. Halaas and P. B. Denyer, editors, *VLSI 91*, Pages 127–136, Edinburgh, Scotland, 1991.
- [Bellenot90] S. Bellenot.
Global Virtual Time Algorithms.
In *Proceedings of the Multiconference on Distributed Simulation*, Pages 122–127, 1990.
- [Bryant77] R. E. Bryant.
Simulation of Packet Communication Architecture Computer Systems.

- Computer Science Laboratory, MIT-LCS-TR-188*. Massachusetts Institute of Technology, Cambridge, Massachusetts, 1977.
- [Bryant81] R. E. Bryant.
A Switchlevel Model and Simulator for MOS Digital Systems.
IEEE Transactions on Computers, C-33(2), Pages 160–177, Feb. 1981.
- [CaiT90] W. Cai and S. J. Turner.
An Algorithm for Distributed Discrete-event Simulation: The “Carrier Null message” Approach.
In *Distributed Simulation: Proceedings of the 1990 SCS Multiconference on Distributed Simulation*, Pages 3–8, January 1990.
- [CarothersPF99] C. D. Carothers, K. S. Perumalla and R. M. Fujimoto.
Efficient Optimistic Parallel Simulations Using Reverse Computation.
ACM Transactions on Modeling and Computer Simulation, Vol. 9, No. 3, Pages 224–253, July 1999.
- [ChandyMH83] K. M. Chandy, J. Misra, and L. M. Haas.
Distributed Deadlock Detection.
ACM Transactions On Computer Systems, Vol. 1, No. 2, Pages 144–156, May 1983.
- [ChandyM79] K. Chandy and J. Misra.
Distributed Simulation: A Case Study in Design and Verification of Distributed Programs.
IEEE Transactions on Software Engineering, SE-5(5), Pages 440–452, 1979.
- [ChandyM81] K. M. Chandy and J. Misra.
Asynchronous Distributed Simulation via a Sequence of Parallel Computations.
Communications of the ACM, Vol. 24, No. 11, Pages 198–206, Nov. 1981.
- [DickensR90] P. M. Dickens and P. F. Reynolds.
SRADS with Local Rollback.
Proceedings of the 1990 SCS Multiconference on Distributed Simulation, Pages 137–143, January 1990

- [FerschaC94] A. Ferscha and G. Chiola.
Self-Adaptive Logical Processes: the Probabilistic Distributed Simulation Protocol.
In *Proceedings of the 27th Annual Simulation Symposium*, IEEE Computer Society Press, Pages 78–88, 1994.
- [Gafni88] A. Gafni.
Rollback Mechanisms for Optimistic Distributed Simulation Systems.
In *Proceedings of the SCS Multiconference on Distributed Simulation*, 19 (3), Pages. 61–67, February 1988.
- [HamnesT94] Donald O. Hamnes and Anand Tripathi.
Investigations in Adaptive Distributed Simulation.
Proceedings of the 1994 Workshop on Parallel and Distributed Simulation, Pages 20–23, 1994.
- [Jefferson85] D. Jefferson.
Virtual Time.
ACM Transactions on Programming Languages and Systems 7 (3), Pages: 404–425, 1985.
- [JeffersonS85] D. Jefferson and H. Sowizral.
Fast Concurrent Simulation Using the Time Warp Mechanism.
In *Proceedings of the Conference on Distributed Simulation*, Pages 63–69, July 1985.
- [Jefferson90] D. Jefferson.
Virtual Time II: the Cancelback Protocol for Storage Management in Time Warp.
In *Proceedings of the 9th Annual ACM Symposium on Principles of Distributed Computing*, Pages 75–90, New York, 1990.
- [JhaB94] Vikas Jha and Rajive L. Bagrodia.
A Unified Framework for Conservative and Optimistic Distributed Simulation.
Proceedings of the 1994 Workshop on Parallel and Distributed Simulation, Pages 12–19, 1994.
- [LinL89] Y.-B. Lin and E. D. Lazowska

- The Optimal Checkpoint Interval in Time Warp Parallel Simulation.
Department of Computer Science and Engineering, Technical Report 89-09-04, University of Washington, Seattle, Washington, 1989.
- [LinP91] Y.B. Lin and B. R. Preiss.
Optimal Memory Management for Time Warp Parallel Simulation.
ACM Transactions on Modeling and Computer Simulation, Vol. 1, No. 4, Pages 283–307, October 1991.
- [Mattern93] F. Mattern.
Efficient Algorithms for Distributed Snapshots and Global Virtual Time Approximation.
Journal of Parallel and Distributed Computing, Vol. 18, No. 4, Pages 423–434, August 1993.
- [MisraC82] J. Misra and K. M. Chandy.
Termination Detection of Diffusing Computations in Communicating Sequential Processes
ACM Transactions on Programming Languages and Systems, Vol. 4, No. 1, Pages 37–43, January 1982.
- [Misra86] J. Misra.
Distributed DiscreteEvent Simulation.
ACM Computing Surveys, Vol. 18, No. 1, Pages 39–65, 1986.
- [RajaeiAT93] Hassan Rajaei, Rassul Ayani and Lars-Erik Thorelli.
The Local Time Warp Approach to Parallel Simulation.
Proceedings of the 1993 Workshop on Parallel and Distributed Simulation, Pages 119–126, 1993.
- [Samadi85] B. Samadi.
Distributed Simulation: Performance and Analysis.
Ph.D. dissertation, Department of Computer Science, UCLA, Los Angeles, 1985.
- [SokolBW88] L. Sokol, D. Briscoe, A. Wieland.
MTW: A strategy for Scheduling Discrete Simulation Events for Concurrent Execution.

Proceedings Distributed Simulation Conference, Society for Computer Simulation, February 1988.

- [SrinivasanR98] S. Srinivasan and P. F. Reynolds.
Elastic Time.
ACM Transactions on Modeling and Computer Simulation, Vol. 8,
No. 2, Pages 103–139, April 1998.
- [Steinman91] J. Steinman.
SPEEDES: Synchronous Parallel Environment for Emulation and Dis-
crete Event Simulation.
In *Proceedings of the SCS Multiconference on Advances in Parallel
and Distributed Simulation*, Pages 95–103, 1991.
- [West88] D. West.
Optimizing Time Warp: Lazy Rollback and Lazy Re-evaluation.
Masters Thesis, University of Calgary, Calgary, Alberta, 1988.
- [WoodT94] Kenneth R. Wood and Stephen J. Turner.
A Generalized Carrier-null Method for Conservative Parallel Simula-
tion.
*Proceedings of the 1994 Workshop on Parallel and Distributed Simu-
lation*, 1994, Pages 50–57.