An Adaptive Time Window Algorithm for Large Scale Network Emulation

Surya Ravikiran Kodukula

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Computer Science

25 January 2002
Blacksburg, Virginia

Keywords: Parallel Simulation, Optimistic Algorithm, Time Window, Emulation

An Adaptive Time Window Algorithm for Large Scale Network Emulation

Surya Ravikiran Kodukula

(ABSTRACT)

With the continuing growth of the Internet and network protocols, there is a need for Protocol Development Environments. Simulation environments like ns and OPNET require protocol code to be rewritten in a discrete event model. Direct Code Execution Environments (DCEE) solve the Verification and Validation problems by supporting the execution of unmodified protocol code in a controlled environment. Open Network Emulator (ONE) is a system supporting Direct Code Execution in a parallel environment - allowing unmodified protocol code to run on top of a parallel simulation layer, capable of simulating complex network topologies. Traditional approaches to the problem of Parallel Discrete Event Simulation (PDES) broadly fall into two categories. Conservative approaches allow processing of events only after it has been asserted that the event handling would not result in a causality error. Optimistic approaches allow for causality errors and support means of restoring state – i.e., rollback. All standard approaches to the problem of PDES are either flawed by their assumption of existing event patterns in the system or cannot be applied to ONE due to their restricted analysis on simplified models like queues and Petri-nets.

The Adaptive Time Window algorithm is a bounded optimistic parallel simulation algorithm with the capability to change the degree of optimism with changes in the degree of causality in the network. The optimism at any instant is bounded by the amount of virtual time called the *time window*. The algorithm assumes efficient rollback capabilities supported by the 'Weaves' framework. The algorithm is reactive and responds to changes in the degree of causality in the system by adjusting the length of its time window. With sufficient history gathered the algorithm adjusts to the increasing causality in the system with a small time window (conservative approach) and increases to a higher value (optimistic approach) during idle periods.

The problem of splitting the entire simulation run into time windows of arbitrary length, whereby the total number of rollbacks in the system is minimal, is NP-complete. The Adaptive Time Window algorithm is compared against offline greedy approaches to the NP-complete problem called Oracle Computations. The total number of rollbacks in the system and the total execution time for the Adaptive Time Window algorithm were comparable to the ones for Oracle Computations.

Acknowledgements

I would like to thank my advisor, Dr. Srinidhi Varadarajan for his guidance and support during the progress of my thesis. I would also like to thank my committee members Dr. Richard Nance and Dr. James Arthur for their patience and providing useful comments on my work.

I thank my colleagues working with Dr. Srinidhi Varadarajan on the ONE for their cooperation.

I would like to thank my family and friends for their encouragement. I would also like to thank my colleagues at Comsat Laboratories for their cooperation.

Surya Ravikiran Kodukula

List of Tables

List of Algorithms

List of Figures

Chapter 1

1 Introduction

As the Internet continues to grow in size and scope, the increasing complexity of network protocols poses several problems in evaluation, verification and validation. Hitherto, protocol evaluation was done in isolation, but the growing interaction between protocols, results in unforeseen behavior. Over the last several years, simulation has appeared as powerful aid to state diagram analysis, providing a cost-effective alternative to testing by real world deployment.

Small-scale evaluations in a controlled lab environment, test beds and custom simulators are useful but they lack the scale and complexity of their eventual target environment. Evaluating the protocol for its correctness and performance in an environment not representative of the target environment - the Internet in our case - is a Type III error (solving the wrong model). Simulating the Internet is almost an impossible undertaking as the heterogeneity in the links, protocols, applications that run on the Internet and the load on the different links is large [Paxson]. Many networking protocols and algorithms work fine in a small network; small compared to the size of the Internet and yet might become impractical when the network scale is comparable to the size of the Internet. The protocol or the algorithm's scalability is an important factor to be evaluated if the protocol is to be claimed to be acceptable for use on the Internet. With the millions of hosts on the Internet, the large range of heterogeneity in the nature of links, load on these links, operating systems, and applications is large. This raises doubts concerning any claim that a protocol would perform as expected in the target environment based on the results of its evaluation in a simulated environment. Added to this, evaluating a protocol by real-world "trial by fire" deployment can be expensive and time-consuming and in many cases, not even practical. Evaluating a protocol or an algorithm for the Internet in a *real world like* environment is the best we can to increase our confidence in any claims about the performance of that protocol when implemented on the Internet.

1.1 Background

Recent years have seen a large amount of research aimed at providing network researchers with tools to facilitate design and analysis of new protocols. Network simulation continues to be an important approach to explore how the Internet behaves versus experimentation, measurement and analysis [Paxson]. The main advantage of live testing of network protocols/applications is that they are evaluated directly in their target environment. However, live testing has several disadvantages: (a) it can only be done late in the development cycle (b) it is expensive and (c) analysis is difficult as tests are performed in an uncontrollable environment and hence results are not always reproducible. Analytical methods are generally limited by simplifying assumptions, which reduce the usefulness of their results. Simulation is the most promising tool as it allows complicated scenarios beyond the scope of Network Analysis, and aids in analysis as the simulation runs are controlled and reproducible. However, simulation suffers from problems such as rewriting the code for the simulation environment and the validity of the implemented code. So simulation results hold good when the focus is on topics such as interaction of different protocols, their functionality etc, but it really cannot make any claims about the performance of the protocol in the Internet. If the simulation implementation of a protocol is considerably different from the real protocol, the claims about the protocol behavior are invalid [Carson]. When evaluating the performance of a protocol like TCP on a simulator 's', we are evaluating the s-implementation of TCP (and of course the bugs) and not any of the standard TCP implementations (and its bugs). Finally a simulated environment is always just an approximation of the reality, and factors such as the CPU load, scheduling of processes at the nodes etc are often hard to model accurately in simulation.

Network Emulation is an intermediate solution between Live Testing and Simulation for protocol and application evaluation. Network Emulation or Direct Code Execution (DCE) is the ability to experiment and run unmodified protocol code in a simulated/emulated network environment. This enables rapid protocol development as it allows developers to move code from their development environment to the Internet and vice-versa. This was one of the major goals of *Entrapid* [Keshav]. Network simulators such as provide an emulator interface that provides the ability to introduce the simulator into a live network. This allows objects in the simulator to introduce live traffic into the live network and the simulator. Emulation though is more realistic - less likely to inadversely neglect things – it requires effort during the development phases due to intensive kernel programming. These reasons limit the size of networks in DCE environments.

When simulating a real sized network the computing power and the resources of a single machine are being stressed. Speedup results by the use of high performance clusters (HPC) based on commodity hardware versus traditional parallel machines [Pham-99] encourages our idea of using HPC to reduce simulation time and exploit any inherent parallelism in the simulated network. The Internet is inherently a distributed environment, and a great deal of parallelism can be exploited when trying to simulate it. A Parallel Simulator is constructed as a set of Logical Processes LP0, LP1… one per physical processor. Each processor simulates events according to its Local Virtual Time. Interactions that exist between the processors are through time stamped messages. As no concept of a *global time* exists, different LPs may advance at different rates and synchronization problems are likely to occur. From the vast research on this problem there are two basic approaches to this problem of parallel simulation.

- *Conservative* mechanisms allow the processing of an event only if the LP is sure that it is safe to do so i.e. an LP should never receive a message in its logical past.
- *Optimistic* mechanisms allow processing of events locally with enough state saved to recover when there is a message effecting causality.

The performance of these mechanisms is application specific. Conservative approaches tend to work well in systems where there is enough *lookahead* to minimize the time the LP spends blocking. The performance of an optimistic mechanism on the other hand is dependent on the state saving mechanism used in the system. Though optimistic mechanisms may intuitively appear to perform well, the overhead of state saving and recovery may be high. Secondly, programming a network protocol in an event driven model with protocol specific user-level checkpointing is a difficult task.

1.2 Thesis Contributions

The goal of this research is to design a new parallel simulation algorithm to act as the underlying framework for the Open Network Emulator (ONE). As the underlying network topology and its characteristics are arbitrary, any assumptions about the temporal ordering of the messages in the system are not likely to hold good. With almost zero look ahead, conservative approaches will not work well. We follow the bounded optimistic approach and define a metric that bounds the optimism at an LP. We compare our algorithm with an offline greedy algorithm that computes the optimal window size taking into consideration the cost of rollback and the cost of synchronization. A large number of adaptive algorithms are reducible to the algorithm proposed. With encouraging results from the other threads of ONE project - '*Weaves and Tapestries'* and the '*User Level Network Protocol Stacks"*, from our knowledge ONE is the only parallel environment to support Direct Code Execution. The main contributions of this work are:

- ✓ The rationale, design and implementation of an bounded time window algorithm that adaptively controls the degree of optimism for the LPs in the parallel environment. The algorithm is reactive and aims at minimizing the total simulation cost by offsetting the cost of rollbacks against the gain in computation speedup due to increased parallelism.
- ✓ Devising an upper bound metric called the oracle, which is an offline greedy algorithm for deriving locally optimal time window lengths. The results from the oracle are used for comparative analysis of the adaptive time window algorithm.
- ✓ Analysis of the parallel simulation algorithm on emulated network traces, which are collected directly from the target environment.
- ✓ A flexible framework for analyzing several classes of parallel algorithms in our environment. When provided with implementations of modules specific to a parallel algorithm, the performance of the algorithm can be compared against another.

1.3 Thesis Outline

The rest of the work is divided into four chapters. Chapter 2 puts the overall ONE project and the current work in perspective with related work in the areas of network simulation, network emulation, direct code execution environments and different approaches to the problem of parallel simulation. In Chapter 3 we present a detailed description of the various parallel simulation algorithms that were studied for the parallel simulation layer for ONE. In this chapter we present the detailed description of a new parallel simulation algorithm – The Adaptive Time window algorithm. In Chapter 4 we describe the idea of oracle computations and present the comparative performance of the adaptive time window algorithm. In this chapter we also present the implementation issues, the traces used for our comparative analysis. Finally we conclude the thesis with a summary of the current work and directions for possible further work in this area in Chapter 5.

Chapter 2

2 Related Work

During the past decade many network simulators and emulators were built, most of them developed for a relatively narrow purpose and for use by a particular research group. Much of the research towards protocol and network performance analysis was centered on the development of a custom simulation environment to validate any new protocols or algorithms proposed.

2.1 Network Simulation

Custom simulators suited for the problem in question address the exact problem faced by the simulator. In most cases, the gain from using a generic simulation framework or a generic simulator is not high enough to offset the learning costs involved in using the generic simulator, resulting in a huge number of network simulators, many of them specific to a particular problem.

Ernst gives a compilation of some of the network simulators available [Ernst]. NEST, developed by the Columbia University department of Computer Science is a general-purpose communication network simulator and it allows actual code to be plugged into the simulator with minor modifications. The simulation runs as a single Unix process and each node has its own thread of control and it shares the global variables and dynamically allocated memory with other nodes. The simulation engine is a synchronous and it proceeds in a series of synchronization passes. The node with the earliest simulated time is scheduled for execution in a round robin fashion. NEST is well suited for understanding the behavior of routing protocols and claims the capability to simulate hundreds to thousands of nodes. REAL is a network simulator based on a modified version of NEST. MIT's NETSIM, U.C. Berkeley's INSANE, MARS etc are some of the other simulators developed during the last decade. A majority of the simulators that were developed were discrete event based and at the core of the simulation engine is the processing of an *event list*. Nodes generate messages and events correspond to messages and other queries.

The VINT project [VINT] provides a composable simulation framework for the simulation modules. The goal of this project is not the building of a new simulator, but to unify research efforts in network simulation. VINT addresses a very basic problem in the field of network simulation – lot of effort wasted in *'reinventing the wheel'* and many of the protocols are studied under isolation when their inter-dependencies are just as important. At the core of this project is the 'ns' simulator which is an object oriented discrete event simulator based on REAL.

Anecdotally apart from the network simulators from academia, commercial network simulators like OPNET, COMNET and BONes are used widely for network planning, management and performance analysis of protocols.

2.2 Network Emulation

None of the network simulators mentioned above support direct execution of network protocols. These network simulators have code modules to simulate the major characteristics of a protocol. The disadvantages of this approach in protocol evaluation are:

1. The protocol code has to be rewritten. For example the protocols like TCP, IP, Ethernet etc are to be coded for the particular simulation environment.
2. Rewriting the code introduces the problem of validation. The simulation implementation might differ from the real world implementation.
3. Another disadvantage of rewriting the code is that it misses some of the behavior present in the common protocol implementation. The authors for *x-sim* [Peterson-96] from their experience mention that while evaluating different BSD implementations of TCP, running the actual code was preferred to running an abstract specification of the protocol.

In such cases simulation is useful only for rapid experimentation and prototyping. With these problems in simulation, emulation appears to be a more promising approach. Emulation according to [Peterson-96] is defined as 'a technique that uses workstations connected by a real network, with modifications to the operating system to simulate links and propagation delays'. In our system by emulation we mean the ability to support *direct code execution*.

Advantages with emulation over simulation are

1. Real protocols can be run and the processing, scheduling overheads need not be simulated.
2. The simulation (emulation) time is the same as the real time to run.
3. It is controlled and reproducible, providing the best of the simulation and live testing.

The advantages with emulation come at the price of some disadvantages.

1. The cost of the necessary hardware to emulate the given network
2. The speed of the emulated host or link is limited by the speed of the underlying hardware; the behavior of the software is tied to specific hardware.
3. Timer resolution limits the accuracy with which delays can be emulated.
4. Due to intensive kernel programming, the development phase might consume large amounts of human resources.

Many of the emulators were built by inserting hooks into the actual networking implementations. In the protocol stack the emulation layer sits above layer 1 - just above the network interface modules. All the packets from the network layer traverse the emulation layer on their way to the network interface. All network interfaces in UNIX transfer to output routines are through function pointers. The emulation layer modifies these function pointers to call custom read and write functions.

Literature suggests that early work in Network Emulation was the use of *flakeways* (gateways that alter or drop packets) for early TCP/IP tests [Estrin]. A WAN Emulator built from *hitbox*, by Thomas Skibos was used for the evaluation of TCP Vegas [Danzig]. Analytical models are used to calculate the queuing, transmission and propagation delays. Packet transmissions are scheduled using the timeout mechanism in BSD UNIX systems. This emulator overcomes the problem of clock granularity by forcing a *softclock* to interrupt every 1000μs, supplanting the default 10 ms BSD scheduler with microsecond scheduling capable of accurate delay emulation. This was the first software implementation of a wide area network emulator that served as a testbed to check flow and switch scheduling algorithms for stability and correctness [Danzig].

Similar protocol testing tools that intercept communication between the protocol layer under analysis and the underlying layers include *dummynet* which allows experiments to run on a standalone system and can be used to simulate networks of any arbitrary topology [Rizzo]. The approximations introduced by *dummynet* are from the granularity and precision of the operating system's timer. The overhead introduced in communication is negligible, so experiments can be done up to the maximum speed supported by the operating system.

x-Sim [Peterson-96] running over the x-kernel [Peterson-88] supports direct execution of transport layer protocol implementations within the x-kernel. A primitive form of process emulation enables it to run only a single instance of a protocol running at any point of time.

Ericsson IPI (formerly Torrent Networks) has an extension to FreeBSD that provides exact emulation with extensibility, controllability and scalability [Torrent]. This allows a single FreeBSD kernel to maintain multiple copies of the kernel's networking state. Though this suffers from the problem of debugging and working at kernel-level it is used for developing and testing protocols, especially routing protocols that span multiple kernels on a single machine.

ENTRAPID is a protocol development environment (**PDE**) that combines the best features of a multi-kernel approach and a general-purpose network simulation. Exact emulation, which allows the protocol developer to move code from the Internet to the protocol development environment, is an important feature of ENTRAPID. This PDE supports multiple Virtualized Networking Kernels (**VNKs**). Each of these VNKs provides the same functionality as a networking stack in 4.4 BSD Kernel. The entire system resides in user space and the VNKs provide the same interface to the programmer as a BSD machine. Each of these VNKs can be configured using an identical set of commands used to configure an actual BSD machine. Applications built on BSD socket API are directly portable to a VNK. ENTRAPID claims it can be used for several thousand VNKs allowing developers to work with really large and complex topologies when developing and debugging protocols [Keshav]. ENTRAPID implements kernel virtualization, process virtualization and external process support to achieve exact emulation. In the survey conducted for this research, ENTRAPID is the only environment that supports process and kernel virtualization, allowing developers to use a single machine for running multiple kernels.

Emulation has additional advantages over network simulators in that end system protocol implementations can be subjected to selective packet dynamics like drop, re-ordering and delays. To gain additional advantages of emulation, an emulation interface is being added to the *ns* simulator. Here emulation is defined as the ability to introduce a real world network node as a component in the simulation [VINT, Kevin]. The simulation in *Opaque mode* treats network data as un-interpreted packets and in protocol mode the simulator generates and interprets live network traffic containing arbitrary fields. The *ns* emulation facility is currently under development [Estrin].

2.2.1 Summary

Emulation typically is achieved by inserting hooks into the operating system and allows the developer to access these from user space. This allows the developer to customize the kernel from user space. A large number of emulators like dummynet, WAN emulator, hitbox, NIST [Carson], use this method. This approach is good for studying protocol modifications, but this comes at a cost of reduced accuracy. Virtualization – both kernel and process make exact emulation possible but none of the emulations except for ENTRAPID scale beyond a few hundred nodes. Although *ns* supports an emulation interface, it still needs the simulation modules for different protocols as only a particular node could be placed in emulation mode. Our project is unique in that it supports emulation – direct code execution, completely in user space with a simulation layer underneath that simulates an arbitrary network topology.

2.3 Parallel Simulation

As the computational demands for simulating a large sized network are high, protocol evaluation practices based on sequential simulations become inadequate. To enhance performance, we need to exploit the natural parallelism inherent in a computer network. This leads us to consider network simulation as a Parallel Discrete Event Simulation (PDES) problem. However, extending a sequential simulator to work on a multiprocessor machine or in a distributed network is not an easy task. The problem continues from existing issues in PDES – sequencing constraints that determine the order in which computations (events) must be executed relative to each other are complex and application dependent [Fujimoto]. So building a parallel network simulator capable of running in a distributed environment or on a multi-processor machine that guarantees performance for any arbitrary network with any arbitrary applications and configurations is still an open research area.

2.3.1 Problem of Parallel Discrete Event Simulation (PDES)

Parallel Discrete Event Simulation relates to executing a single discrete event simulation on multi processor machine or a network of workstation. The simulation is broken into Logical Processes that are mapped to different physical processors. Event processing in a sequential simulator is relatively straightforward. The events generated in the system are inserted in an event list and the task of the simulator is to handle the events in strict non-decreasing time order. In a parallel environment, events generated for the local Logical process LP_i are called *local events* and events generated for other Logical Processes (LP_j, j≠i) are termed *external events*. As there is no global clock in a parallel/distributed system, communication between the logical processes takes by means of time stamped messages. The Local Causality constraint states that "discrete event simulation, consisting of logical processes (LPs) that interact exclusively by exchanging time stamped messages, obey the local causality constraints if and only if each LP processes events in non-decreasing timestamp order" [Fujimoto]. This is a sufficient condition and in our current system, it is always guaranteed that events within the single LP though independent are processed in strict time order.

A causality error results when an LP receives a message that corresponds to an event with a timestamp less than the LPs current simulation time. Consider two LPs LP1 and LP2 in a system. LP1 has event e1 with timestamp 10 and LP2 has event e2 with a timestamp 5. If processing event e2 results (Figure 2-1) in a message m destined to LP1 and this results in an event e3 with timestamp 7 at LP1, processing e1 before the arrival of message m would result in a causality error.

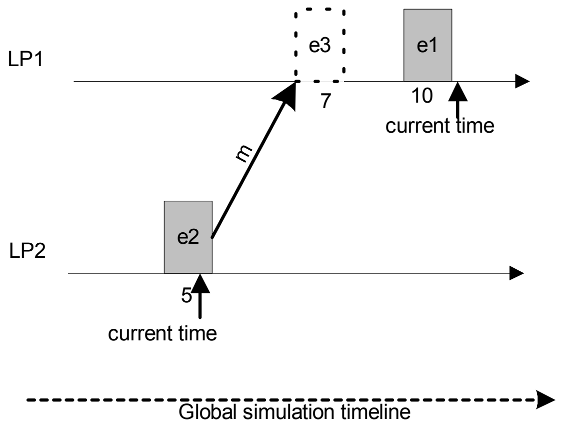

Figure 2-1Causality in a PDES

Approaches to solve the problem of maintaining correct event ordering broadly fall into two categories *conservative* and *optimistic*. A more detailed classification is described in [Reyonlds1], which describes a spectrum of options that includes these two categories. For the rest of the discussion we consider only the two broad approaches.

2.3.2 Conservative Approach

In a conservative approach an LP processes those events that are safe to process. A safe event is defined as an event that would not result in a causality error in the system, i.e. an LP processes an event *ev* only if it is ensured that it would not receive any external event (caused by a message from another LP) with a timestamp less than *t(ev)*. LPs having no safe events need to block. The Chandy-Misra algorithm is the one of the first algorithms to solve this synchronization problem in PDES [Chandy]. A more mathematical definition for a conservative algorithm is given by [Reynolds-88].

"$\forall$i,j guarantee that LP_j never receives a message from LP_i such that the message timestamp representing LC_i at the time the message was sent is less than LC_j." (LC_i is the logical clock or the current simulation time of LP_i)

Two approaches to guarantee that processing of an event *ev* does not result in a causality error at any LP are:

1. Strictly Conservative and
2. Conservative with look ahead.

In a conservative strategy each LP needs to know the logical clock value of all the other LPs to decide whether to process the smallest event ev1 in its event list. Communication between LPs is in the form of messages and a message *m* from LP_i to LP_j contains the logical clock value of LP_i in the message (other implementation specific information such as the time for which this event is scheduled at the destination LP_j can also be included). The input channels are associated with a logical clock, corresponding to the time stamp of the smallest message on that link. C_{ij} represents the logical clock associated with the link connecting LP_i with LP_j and it corresponds to the smallest message from LPi. So each LP processes event *ev* only if it is certain that t(*ev*) is not greater than any of the its channel logic clocks. This is a sufficient condition for maintaining the causal ordering of messages in the entire system.

Lookahead of an LP is defined as the capability to predict the (external) events it generates. An LP with a lookahead of *l* time units guarantees that the next external event it would generate would be at *l* time units from now. So events with time stamp less than *t+l* (where t is the simulation time on this LP) can be simulated sequentially without affecting the causality in the entire system. In systems where the look ahead is small or zero a strictly conservative approach is followed. In a strictly conservative approach each LP processes events whose timestamp is the smallest in the entire system and communicate among themselves, the next event in its input lists. An implementation of strictly conservative approach is [Cilk-97, Cilk-90], using the *Cilk* runtime library shows good speedup and efficiency. In this system, safe time is calculated taking into consideration the local safe time and the global simulation time. It implements the *Qheap* data structure [Steinman] for its event list management. This approach is applicable to systems which have several events scheduled for the same simulation time, which increases the number of events processed in every lock step and efficiency of the merge operation on the *Qheap* data structure. These experiments were run on a shared memory environment and there hasn't been any analysis as to its performance in a distributed network of computers.

In the conservative approach each LP computes *safe time* based on the logical clock values on its input channels. This could lead to the LPs blocked in a circular wait and might lead to a deadlock. To avoid deadlock, an LP sends *Null messages* [Chandy] to all its connected LPs after it has completed processing an event. Null messages add processing and communication overhead. A number of other techniques to reduce null message overhead, avoid deadlock and deadlock recovery can be found in [Fujimoto].

In systems where look ahead is available, an LP can process more events in a single lock step before all the LPs synchronize. Empirical studies [Nicol-88, YAWNS, Pham-98, Cleary] suggest that look ahead plays an important role in performance – as it reduces synchronization costs and blocking times. An analysis on the impact of lookahead [Abrams] shows that the performance gain due to lookahead is entirely dependent on the model, but lookahead guarantees no worse performance. The conservative approach is efficient if the system has a good look ahead. The larger the lookahead, the better is the performance of this approach.

2.3.3 Optimistic Approach

Another approach to the problem of synchronization in PDES is the *optimistic approach*. In an optimistic approach, the system detects causality errors and recovers from these errors. Each LP processes events in its event list optimistically; instead of determining when it is safe to process an event. LPs also detect the occurrence of a causality error and restore their state to a known correct state. This state restoration is termed as a *rollback*. A message that causes a rollback is termed a *straggler*. After rollback all messages sent by this LP to other LPs in the system should be invalidated. This is achieved by sending *anti-messages* or by committing state after the LP decides that it need not restore to that state.

The *Time Warp* protocol [Jefferson] is the first and the most well known optimistic algorithm. Time Warp is based on a paradigm that is an abstraction of the real time called *Virtual Time*. [Jefferson] describes virtual time as "global, one-dimensional temporal coordinate system" and "the virtual clocks tend to go forward toward higher virtual times, they occasionally jump backward". The idea of virtual time allows the LPs to progress at different rates, which is very much likely to happen in a parallel simulation environment. Initial time warp was demonstrated on a multiprocessor machine without any modifications to the underlying operating system and the idea progressed to the development of the Time Warp Operating System. Time warp assumes that the state of every LP is saved which the LP can restore to when it receives a straggler. For global control mechanism the concept of *Global Virtual Time* (GVT) was introduced. GVT is defined as the minimum of all the virtual times in the entire system, so GVT corresponds to the last safe message in the entire system. As GVT is calculated globally utilizing local information, GVT cannot go backwards in time. In time warp, GVT information is used for efficient memory management and flow control. The concept of virtual time is applied in other areas like Distributed database concurrency control, virtual circuit communication. Virtual Time concept was a completely novel solution when proposed and it is an extended analogy to the concept of virtual memory in operating system. [Jefferson] gives a very good comparison of the concepts in parallel simulation to its equivalent in virtual memory. Events are compared to pages in virtual memory, events to be processed to pages in main memory and past events to pages not in main memory. The important aspects of this analogy that guided out current approach were:

1. Just as the spatial locality principle reduces the number of page faults, "under a virtual time system it is only cost effective to run programs that obey the ***temporal locality*** principle.
2. Increasing the number of pages in main memory can reduce the number of page faults; similarly the number of rollbacks in a virtual system can be reduced by increasing the number of events in its future, which translates to introducing delays.

After the initial time warp protocol, there has been a considerable amount of work in improving the performance of this optimistic approach. Computation of GVT, reducing the amount of memory required, cancellation of the incorrect events processed, check-pointing state are some of the areas on which research has been focused to improve the performance of time warp. A good reference to these optimizations is given in [Fujimoto].

Time warp (the optimistic approach) was developed to reduce the blocking times of the LPs found in conservative systems and exploit the parallelism in the system. The gain in performance is highly dependent on the state saving mechanism employed. The cost of state saving directly effect to the amount of unnecessary computations by the processor. The commonly used technique is saving the state before the event computation. State saving has been implemented in many systems and well studied. A detailed analysis of the effects of saving state incrementally and periodically is found in [Wilsey]. These state saving mechanisms constitute severe overhead on the processor in fine-grained simulations or when the event density is high, as the number of calls to the state saving routine per time is large. *Reverse Computation*, a compiler-based approach [Kalyan], restores state by performing the inverse of the operations performed in the event execution. Results showed that this method performs an order of magnitude faster than the traditional stack copying methods. This algorithm has been analyzed on simulated protocol code and its performance has not been analyzed for real network code that is complex when compared to its simulated counterpart. Moreover the modeler has a very little control over the code in our emulation environment than simulated code for a protocol to apply such techniques.

In the original time warp, logical clocks can run at different speeds, causing increasing dissonance as the simulation progresses. This requires the ability to save an unbounded amount of process state to ensure correct recovery from causality errors. The above drawback led to idea of bounding the amount of optimism of an LP in the system. Approaches like Moving Time Window (MTW) [Sokol] and Bounded Time Warp (BTW) [Turner] use a fixed time window of size W and the LPs process all events till the end of the current window. All the LPs synchronize at the end of each time window. MTW and BTW use a fixed window size and the performance depends on the choice of the window size. Both the algorithms make several runs to determine an optimal size for the size of the window. These time window mechanisms cannot differentiate events, as ones leading to a rollback or ones that do not. This may reduce the progress of an LP by blocking it at the end of the window even if there weren't any incorrect computations. There hasn't been any clear study as to how the size of the window should be determined for achieving good performance [Fujimoto]. The current work is based on the idea of time windows and is different from the previous time window techniques in that the size of the time window adaptively grows and shrinks to achieve an optimal performance. Also the previous techniques do not have a complete analysis and definition of the *optimal window size*.

In approaches like Local Time Warp LTW [Rajei] optimism is bounded by spatial boundaries, by defining a hierarchy among the LPs of the entire system. This approach combines the conservative and the optimistic approaches by allowing the LPs within a subsystem to proceed by the optimistic approach and the synchronization between these subsystems is by a conservative approach. The idea of using a hierarchical synchronization was a novel idea, but it suffers from the pitfalls of both the conservative and the optimistic approaches. Applying hierarchy among LPs could be a viable approach when the network is partitioned and there is very little communication among the components. But when the modeler has very little information about the network characteristics a priori and no control over the mapping of the network elements to the physical processors this approach as discussed suffers from pitfalls of both the traditional approaches.

Some approaches restrict the number of rollbacks and propagation of the rollbacks by the LP sending a broadcast message to the remaining LPs as soon as it observes a rollback [Madisetti, Damani, Prakash]. These approaches are reactive in that they limit the optimism of an LP after a rollback in an attempt to reduce further rollbacks in the system.

2.3.4 Adaptive Protocols

The choice of a particular protocol, conservative or optimistic for a particular system is a problem with a large parameter space. [Reynolds-88] defines adaptability as "changing design variable bindings based on knowledge of selected aspects of the simulation state". Optimistic protocols incur costs for state saving, rollback and memory management. There is another cost for optimistic protocols – *lost opportunity cost*. This cost refers to the possible loss in performance when an LP blocks even when it was safe for it to process events. Referring to Fig 2.2 [Reynolds-98] the increase in the optimism (lower restriction of an LP) increases the rollback (state saving) and memory management costs. Lower optimism translates to a higher lost opportunity cost. All adaptive protocols try to work in the knee of the total cost curve for optimal performance. The requirements for an adaptive algorithm to achieve optimal performance are dynamic and completely dependent on the temporal ordering of events in the system.

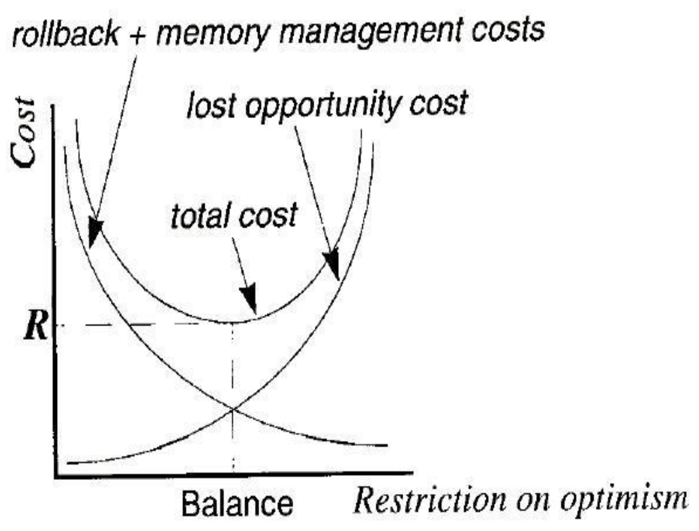

Figure 2-2 Costs in an optimistic approach

A number of adaptive protocols [Tripathi, Fershca1, Rego, Ferscha-95] were proposed which make use of input channel information like the arrival of messages on channels. These algorithms collect channel specific information and determine whether to block for a certain amount of time or allows the LP to process the next event. These algorithms monitor the real time when the last event arrived on a specific channel and based on this information an optimal CPU delay is computed, taking into count the rollback probability and cost of rollback. Utilizing information about the arrival time of messages on input channels and blocking an LP has shown positive results; adaptive protocols have shown better performance than the classical Time warp [Ferscha-95]. The disadvantage in such approaches is the efficiency is determined by the granularity of the system clock. If the system clock granularity is not fine, the LPs end up blocking unnecessarily. A Probabilistic adaptive protocol [Ferscha-94] has been proposed which using a time window approach and changes the size of the time window adaptively by calculating rollback probabilities and processing events with a probability related to the confidence in the LPs estimate of the next event. Approaches like this are highly dependent on the temporal pattern of messages between LPs. Moreover this protocol was analyzed on *petri-net*s where the message characteristics can be well exploited. The results and analysis are not based on a general model to apply them to any new system.

As perfect state information required for the optimal performance of an adaptive algorithm is impossible [Reynolds-98], *Elastic time* uses the concept of *'Near State perfect Information'* (NPSI). This approach assumes the existence of a feedback system that provides the LPs with NPSI without extra computational overhead. NPSI protocols include two phases:

1. State information that controls the decision of optimism. This constitutes computing an "*error potential* indicating the likelihood of an LPs computation becoming incorrect" [Reynolds-98].

2. Mechanisms to translate this information into the amount of optimism. Choose an appropriate mapping from the NPSI to the degree of optimism; many of the adaptive algorithms can be reduced to the NPSI protocol.

The *Elastic Time* algorithm is an NPSI adaptive protocol. [Reynolds-98] gives an analogy for understanding the algorithm. Each of the LPs is bound by an elastic band with all its predecessors and all the LPs tend to move forward in time. Its predecessor LPs from which it receives messages controls an LPs forward movement in virtual time. This algorithm very well captures the idea of a feedback system in achieving optimal performance. Results show that the Elastic time outperforms the Time Warp for a wide range of workloads.

The disadvantage of the Elastic time algorithm is that the algorithm requires every LP to know LPs in the system that constitutes its predecessor set. For topologies used in the analysis [Reynolds-98], each LP knows the predecessor set. For a general network simulation, predecessor information for any LP cannot be determined beforehand. This results in the topology being a strongly connected mesh, each LP connected to every other LP (or every physical processor to every other processor in the system, if LPs are aggregated). A simple analysis of the Elastic Time algorithms reduces it to a conservative approach when each LP (physical processor) has all other LPs (physical processors) in its predecessor set. Though the *Elastic Time* utilizes the idea of a feedback system to bound a processor's optimism, it is based on an overly simplified network connectivity graphs.

A general model of rollback is given in [Lubachevsky], which proposes an adaptive optimistic algorithm called *Filtered Rollback*. This algorithm uses the concept of timed window and a mathematical proof of its efficiency. The analysis of this algorithm reveals the viability of rollbacks and gives an upper bound on the overhead of rollbacks. Our present work is similar to this in the idea of a time window, but the metric defining optimality and analysis is completely different. *Filtered Rollback* was analyzed on a shared memory system. No experimental results of this algorithm on a network of workstations and its behavior when the target system is similar to ONE are given. There is no complete analysis of the cost associated with the loss of optimism in any of the above approaches. We try to analyze the cost of lost opportunity in our experiments.

2.3.5 Parallel Network Simulators / Emulators

DistREAL is a distributed version of the REAL network simulator. In DistREAL, simulation time is divided into passes and barrier synchronization is used to synchronize different instances of the simulator. Gain in execution time is limited as the simulator is strictly conservative and has no support for rollback. Language based simulators like Maisie and TED separate the underlying simulation kernel (sequential or parallel) from the simulation programmer. Several other parallel network simulators have been developed, but for specific networks like ATM and ISDN. Results from [Cleary] show that assumptions about traffic density in ATM networks and a non-zero lookahead make conservative approach give a better performance than the optimistic approach. This study extends the idea of lookahead and defines a term Mean Lookahead time (MLT). When the network is modeled as a graph, with the processes and the links as vertices, the Loop Time (LT) is the sum of the lookaheads on all the links in the loop. Them MLT of a channel is the minimum LT of any loop that passes through this link. A lower bound for the average number of suspensions per simulation time unit, is given by $\Sigma 1 / MLT_i$ (for all links i). They propose several optimizations based on this statistic. But the analysis is on a multiprocessor and cannot be directly applied to our system.

Results from [Pham-99] also show efficient results for the conservative algorithm with dynamic load balancing. Interesting results from these experiments were

a) The aggregation of switches aimed at reducing the inter-processor communication did not improve speedup but improved only the efficiency. Possible reason was the low communication cost in their setup.
b) Another surprising result was that the distributed execution outperformed sequential execution even for low loads. These experimental results cannot be directly applied to our problem as they make assumptions on the look ahead of the link, which enhances the performance in the case of routers but not hosts and the analysis is on a shared memory multi processor.

In simulators like *ns*, which are not designed for simulation of large networks it is almost impossible to simulate large networks due to excessive memory requirements and computational demands. PDNS is an extension to *ns* to allow a network simulation to be run in a parallel and a distributed environment [Riley]. PDNS uses a conservative approach to synchronize the LPs. Communication between processors is minimized to reduce the causality effect of each processor on the other. The performance of the PDNS entirely depends on the topology specified by the user and the mapping between the different hosts / parts of the network on to physical processors. The links in a physical topology when specified to the simulator need to be classified as remote link (*rlink*) for links that connect nodes on different processors. With no state saving mechanisms being used, there is no support for dynamic load balancing between the processors. PDNS has implementations that work in a distributed environment - workstations connected either via a Myrinet network or a standard Ethernet network running over TCP/IP. PDNS was designed as an extension to the popular ns simulator rather than a true ground-up parallel simulator. Hence, the key goal was to minimize the changes to the ns source code to avoid validation issues. In contrast, ONE is being developed to run in a distributed/parallel environment on a network of workstations.

Commercially available network simulators like OPNET aided with libraries to communicate between workstations like ScramNet [ScramNet] are used for the simulation of Radio Networks. They use the conservative approach for synchronization. OPNET also provides parallel simulation modules and is targeted towards modeling wireless simulations [Opnet].

2.3.6 Summary

In systems where there is very little or no look ahead, optimistic approach is a feasible solution. Unbounded optimism like the classical *Time Warp* could lead to poor performance. Performance results of bounded optimistic algorithms are promising for use in network simulators. Almost all the parallel simulation algorithms proposed do not include any analysis on the performance of the algorithm when the target environment is similar to ONE. Restricted analysis of algorithms on models like a simple queuing model, petri-nets etc leave the connection between network emulation / simulation and parallel simulation unexplored. The current work focuses on this unexplored portion of parallel network simulation.

2.4 Chapter Summary

Dr. Fujimoto's '*Holy Grail'*, that provides arbitrary modeling capability to automatically parallelize an existing system yielding significant speedups does not currently exist. For research in the area of PDES to have maximum impact, the goal is to provide application specific PDES tools [Nicol-96]. Applying parallel techniques to an inherently sequential simulation environment would not serve our purpose. According to Dr. Nicol, many of the experimental designs in PADS are flawed either by analyzing on one model or the important factors effecting the model are ignored. Experiments are limited to simplified models like queuing systems, Petri-nets and hardware system. The present work started with a goal of choosing an appropriate algorithm for a parallel simulation engine for ONE and to provide sufficient analysis to support our choice. We concentrate on the synchronization and optimization techniques. We justify our choice of the technique with analysis using traces similar to the target system.

Figure 2-3 illustrates the relative position of our project ONE in the area of parallel network simulation / emulation. The axes represent the nature of the environment – sequential, parallel and emulation support. A particular environment is plotted in this space by its relative position based on its support for sequential, parallel and emulation features.

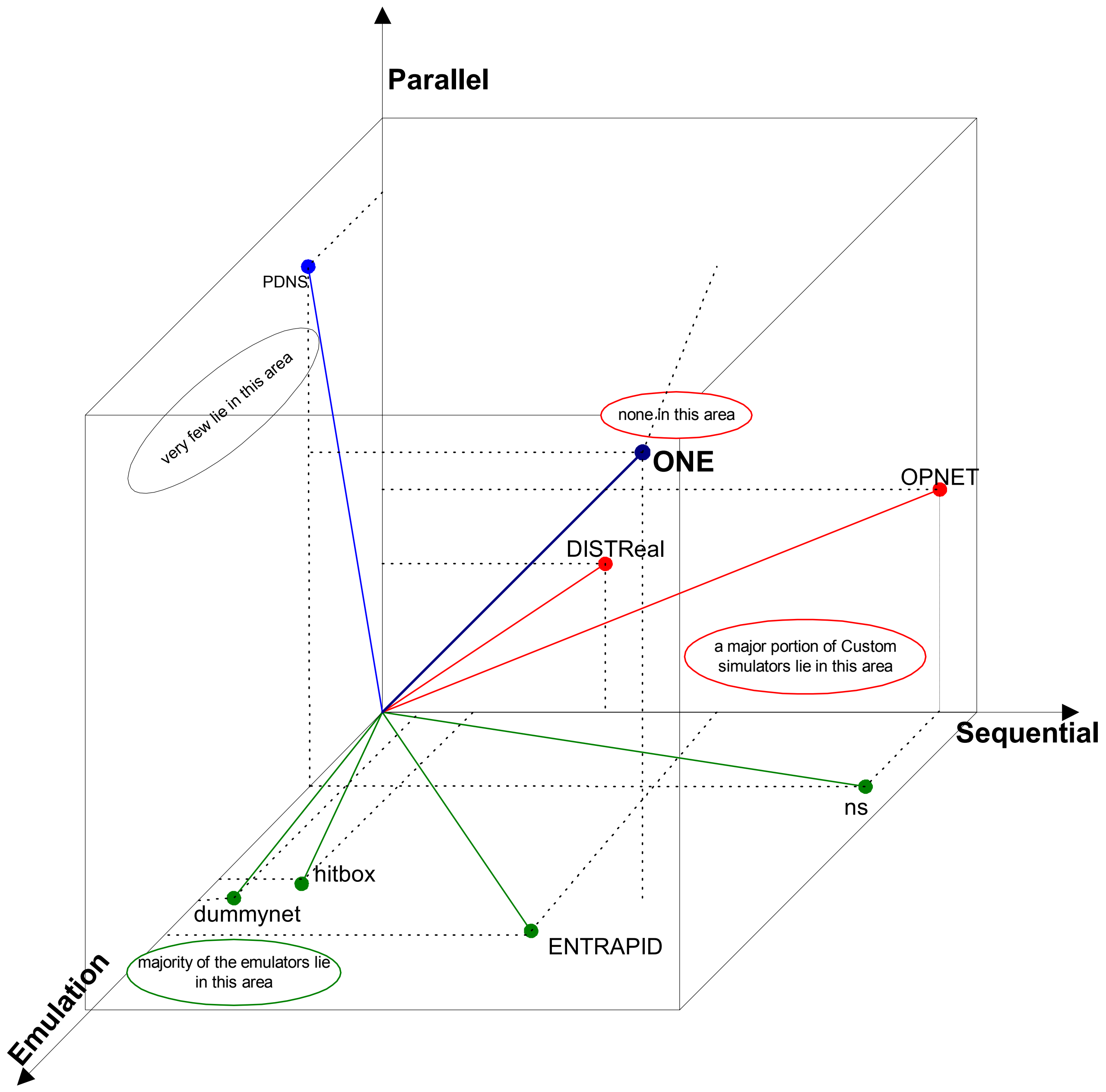
Parallel
Sequential
Emulation
PDNS
very few lie in this area
none in this area
ONE
OPNET
DISTReal
a major portion of Custom simulators lie in this area
ns
hitbox
dummynet
ENTRAPID
majority of the emulators lie in this area

Figure 2-3 Emphasis of the current work in the area network simulation / emulation

Chapter 3

3 Architecture

The primary research goal of the Open Network Emulator (**ONE**) is to provide a realistic testbed environment that supports execution of unmodified network protocol stack and application code. This facilitates the development and testing of protocols and applications in an environment that is representative of the intended target environment. The research presents several interesting challenges. First, such a protocol development environment requires the simulation of a network complex enough to be representative of a real network. Since traditional network simulations either (a) do not support simulation of large networks or (b) were not designed for a large-scale emulation, we need a simulation engine that performs well for ONE. The second task is to provide a framework for direct code execution of unmodified application and protocol code. This task is complicated further when multiple virtual hosts are emulated on a single workstation, each running its own set of applications. The current work is focused on the first task taking into consideration the overall issues involved in ONE. This chapter discusses these issues in detail.

3.1 Components

The goal of this research is to develop a distributed simulation engine for the Open Network Emulator, capable of operating over a cluster of workstations. This section discusses the various components of the system and their requirements.

3.1.1 Parallel Simulation Algorithm

The architecture of the ONE uses a parallel simulation engine as its bottom layer to simulate an arbitrary network. The simulation engine is responsible for simulating large complex network topologies, representative of the Internet. Since the simulation layer has no control or knowledge of the network application being simulated, it cannot make any assumptions about traffic patterns or temporal ordering of the communication events. This leaves the simulation engine with a lookahead equal to the least of the transmission delays of the link – a value in the order of microseconds. When each of the physical processors (workstations) simulates a number of hosts, the lookahead effectively tends to zero. Existing parallel network simulations discussed in Chapter 2 require the simulation modeler to manually specify a load balanced mapping between simulated virtual hosts and the physical processors. We would like to build and analyze the behavior of a parallel simulator when:

- The mapping of the hosts to the processors is arbitrary and
- The mapping is within the simulation layer, transparent to the user.
- The events are generated from an environment that supports direct code execution.

The simulation layer hence needs an algorithm to address the problem of PDES with sufficient speedup and efficiency. For fair comparative analysis of the speedup and efficiency the simulation algorithm should support a sequential version, which excludes the communication and synchronization overheads.

3.1.2 Simulation Engine

We define virtual host as the host with all its applications and its protocol stack bundled together and that is being simulated in the system. Several such virtual hosts with their application and protocol code are linked to the simulation layer on a single physical processor. Communications between the virtual hosts on the same physical processor translate to local events and those between virtual hosts on different processors translate to external events. Figure 3-1 shows the simulation engine and its interaction with the virtual hosts. The simulation layer beneath the different hosts simulated on each of the physical processors is transparent to the upper layers. There should be a complete mapping from the functionality the hosts expect from network devices they are connected, to the simulation layer.

So the simulation layer needs to provide:
- *pseudo devices* imitating the functionality of network interfaces of hosts.
- The ability to query these pseudo devices for statistics and control data.
- The ability to transmit messages to other hosts in the network on a particular (pseudo) device.
- The ability to receive messages from other hosts in the network on a particular (pseudo) device.
- The ability to insert and delete timers.
- The ability to query the simulation time and the clock frequency.
- Support for migrating a virtual host with its associated state to another processor in the system, based on a load balancing decision.

3.2 Assumptions

The simulation layer makes the following assumptions:
1. The protocol and application code for each of the hosts on a single processor is moved to the user space and bundled with the local copy of the simulator using the Weaves framework. This gives us a single user level process with efficient state saving, rollback and migration capabilities. The analysis of the parallel simulation algorithm is done assuming this support.
2. We also assume the interconnection network of workstations in the system to be fault tolerant. All messages between the workstations are assumed to be strictly *in order*; i.e. the link layer doesn't reorder messages.
3. In the current system, there is a centralized processor that acts as a *master*. Though this might be a possible single point of failure, we justify this and other assumptions in section 3.4.

4. We restrict our current system to topologies where the links between hosts are simple point-to-point links.

3.3 Design

We give a description of the design of the entire testbed. Figure 3.1 shows the various layers in the system. Virtual hosts, with different applications running on them form the uppermost layer in the system. The lowermost layer is a simulation engine that simulates a given network topology. The intermediate *personality layer* maps the network device functionality onto the simulation layer.

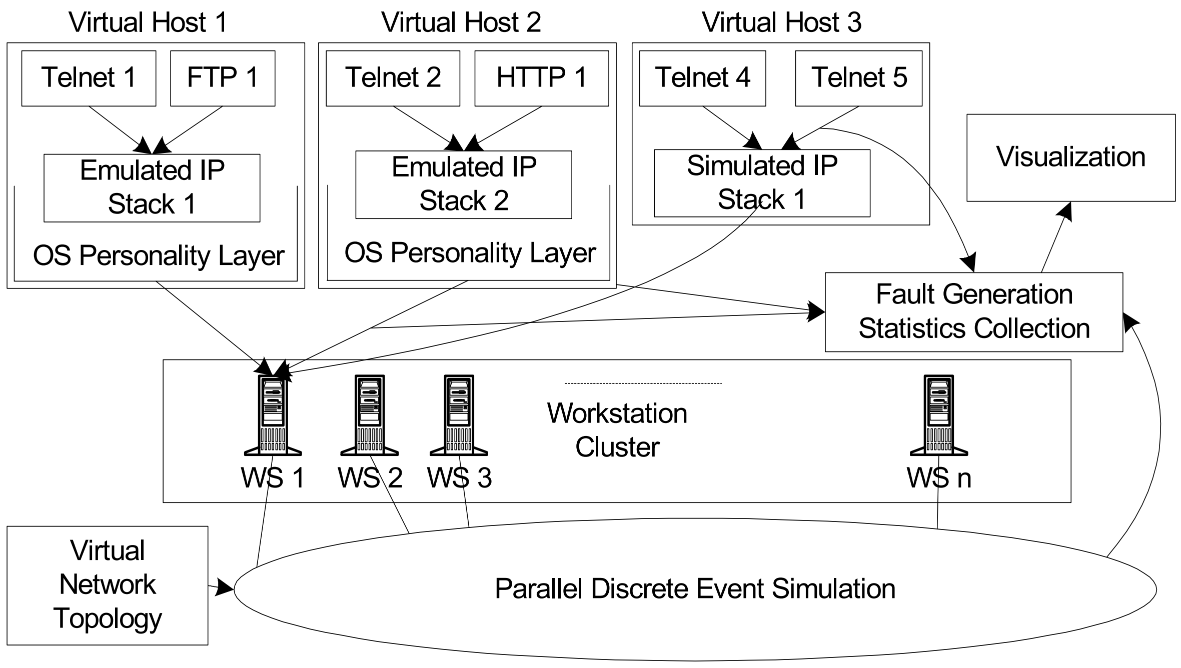

Figure 3-1 ONE Block Diagram

3.3.1 Parallel simulation algorithm

Each processor (workstation) in the system simulates a number of hosts in a given network topology. Each host is logically a distinct entity. From the perspective of the simulation layer, all virtual hosts on each processor form a single logical entity. The simulation layer is unaffected in functionality by the number of hosts each processor simulates. Thus the simulation layer is a set of *logical processes* LP0, LP1, LP2…, one per physical processor (workstation). Figure 3.2 shows the simulation layer in detail. The topology is represented by a graph with the hosts as vertices and the edges between these vertices are the communication links between the hosts. In the simulation algorithm we use the term **logical process** to refer to a **physical processor** with all the hosts and associated applications linked to it.

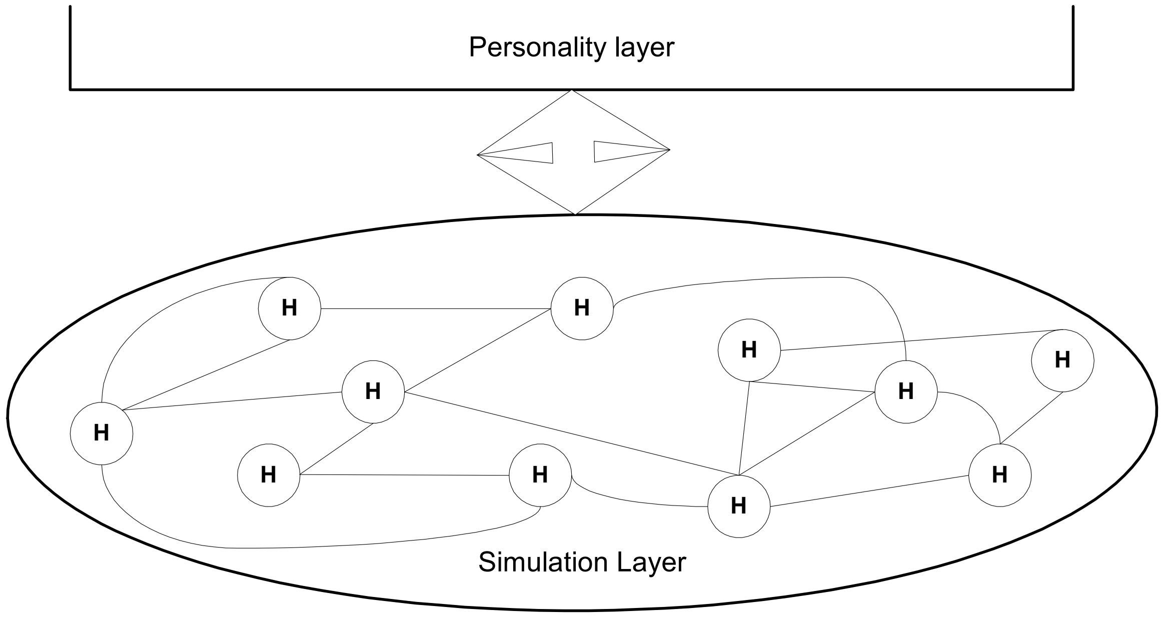

Figure 3-2 The simulation layer

All the interactions between the host and its applications with the network interface translate to events to the simulation layer. We classify these events into three categories. *Local events* are events that are associated with a virtual host that exists on the same processor. They are termed local, as any messages between the two hosts are not physically transmitted over the wire. *External events* are events that are associated with a host simulated on a different processor. We call these events external, as they need to be physically transmitted. *Timer Events* are events associated with the simulation time like querying the current simulation time, a timeout event etc.

For example, consider 3 hosts *a*, *b* and c. Hosts *a* and *b* are simulated on physical processor p1 and host *c* on processor p2. A message *m* from *a* to *b* is a local event where a message from *a* to *c* is an external event.

3.3.1.1 Conservative Approach

PDES approaches are built around the idea of imposing a global clock among all the Logical Processes. We introduce some more terms for clarity in explaining the design. We use the term processor to refer to the **physical processor (workstation) and LP_i to refer to the logical process associated with processor i**. **Virtual time** of a processor refers to simulation time and gives the progress of the processor in time. A **channel** is defined as a logical connection between two Logical Processes. The physical connection corresponding to a channel could be a shared medium or point-to-point medium. **Global Virtual Time** (GVT) refers to the time stamp of the smallest safe event in the entire system. No LP can receive a message scheduled for a time less than the GVT. All the LPs have input and output buffers associated with every other LP in the system.

With a zero look ahead available at each of the Logical Processors (processors), a conservative algorithm cannot achieve good performance. In a strictly conservative algorithm, the only speedup is from events scheduled for the same simulation time. Additionally, if we assume that the hosts in a network generate traffic (events) with arbitrary periodicity, the traffic in the entire system is the superposition of these periodic traffic sources. The net effect is self-similar traffic patterns that can only be viewed globally.

3.3.1.1.1 Strictly Conservative

A conservative algorithm without any lookahead is described in Algorithm 3-1. The algorithm is similar to the one used in [Cilk-97] in the way it calculates safe time and GVT. In the current system a centralized *Master* processor coordinates the remaining slave processors. The simulation proceeds in lock steps. In each lockstep each LP goes through three phases until global simulation time (`gst`) has reached the end of simulation (`final_time`):

- In phase 1 it calculates `safetime`, which is the maximum of the `gst` and the minimum of all messages on the input channels, `local_min`.
- In phase 2 it computes all events with time stamp less than or equal to the `safetime`.
- In phase 3 each processor sends the minimum of its event list's smallest event (`eptr_time`) and all messages in its output buffer (`out_time`) and receives new `gst` from the master processor.

Algorithm 3-1 Slave processor in lock step parallel simulation

```
//At the slave processors, do until end of simulation time.
while (gst<=final_time)
{
   Check the receive buffers for any messages
   min_max[i] = last message on each input channel i.
   local_min = MIN(local_min, min_max[i])
   (for all input channels i)
   safe_time = MAX(local_min, gst);
   local_processing(safe_time);
   // process all events with time stamps less than equal
   to the safe_time
   eptr = heap_min(event_heap);
   // minimum time stamp event in the event heap
   out_time = MIN(of messages on all output channels)
   local_safe = MIN(out_time, eptr_time)
   send local_safe to master processor
   receive the new gst from master
}
```

The master processor receives the local estimates of the new safe time and broadcasts to all the slave processors the minimum of all these safe times. (Algorithm 3-2)

Algorithm 3-2 Master processors in lock step parallel simulation

```
// at the  master processor, keep doing this till end of
simulation
 while(TRUE)
{
   min=SIM_INFTY; // initialize to infinity
   Receive messages (local_safe(i)) from all the slave
   processors
   min = MIN(all local_safe times)
   Send messages (min) to all the slave processors
   If (min >= final_time) return;
   // check for end of simulation time
}
```

For the system under study, results from the conservative approach are not encouraging. In the lock step parallel simulation algorithm described above, the only parallelism gained is from events scheduled for the same virtual time in the entire system. As each LP cannot *apriori* calculate when the next external message would arrive, the entire system moves at the rate of one virtual time instant per lock step which is the timestamp of the smallest event in the entire system.

3.3.1.1.2 Conservative with lookahead

We now explore the possibility of a *lookahead*. In principle lookahead provides each processor with a time window. When an LP has sufficient lookahead and can estimate the arrival time of the next external message from another LP in the system, every LP processes a window of events in every lock step of the algorithm. The YAWNS protocol [Nicol] and PDNS [Riley] are built on this idea of look ahead. The size of this time window depends on the lookahead available at the LP that translates to the external message inter arrival time.

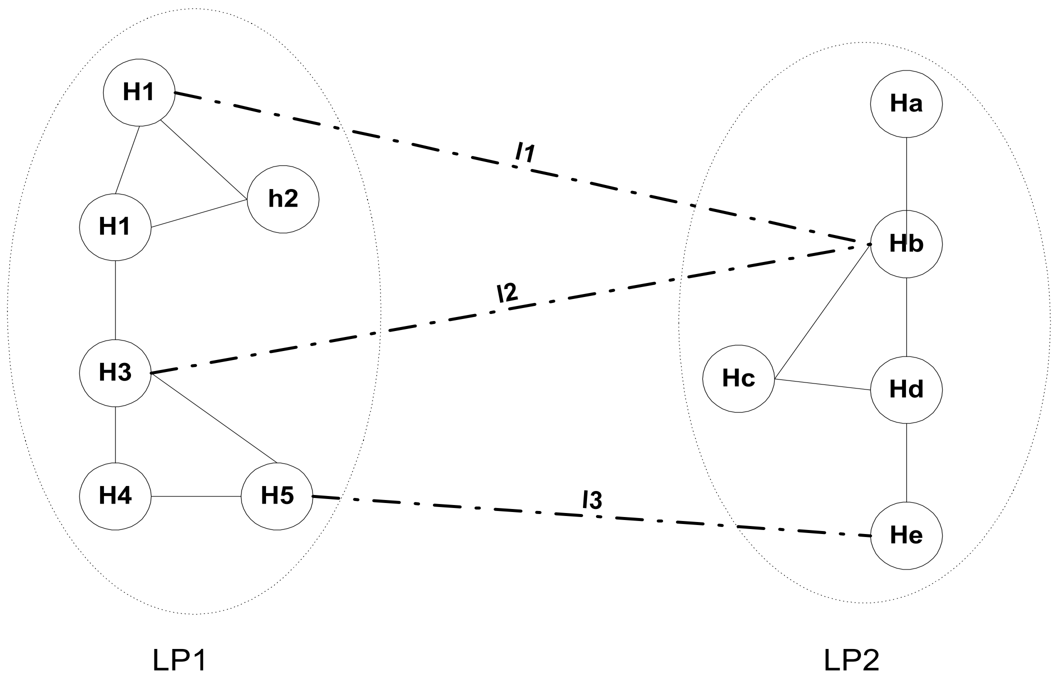

Figure 3-3 Lookahead in a parallel network simulation

Consider two LPs LP_1 and LP_2. (Refer to Figure 3-3). Each LP simulates a set of hosts. LP1 can receive messages from LP2 on virtual links `l1`, `l2` and `l3`. LP1 keeps track of the timestamps of last messages on each of these links, `last(l1)`, `last(l2)` and `last(l3)`. The transmission delays on these virtual links are given by `d(l1)`, `d(l2)` and `d(l3)`. The next message on `l1, next(l1)` cannot be before `last(l1)` + `d(l1)`. So in every lockstep LP1 can process events with time stamp less than the `MIN(next(l1), next(l2), next(l3))`. This window depends on the transmission characteristics of the virtual links in the topology. When the number of virtual links increases, the last messages on each of these links arriving at an LP are spaced away from one another; the lookahead window size decreases, reducing the algorithm to strictly conservative. Performance gains could be achieved if the mapping of the hosts to the processors is done so as to reduce the number of virtual links between the processors. With our goals in mind and the dependency of *look ahead* on the mapping from the hosts to processors, make the conservative approach unsuitable for the current system. We discuss more on our rationale in not choosing this approach in the next section.

3.3.1.2 Optimistic Approach

In the optimistic approach a Logical Processor (LP) is allowed to proceed ahead in virtual time, until it receives a message that would affect its simulated past. A message that causes the LP to restore its state and go backwards in time at least that far as this message is termed a *straggler* and the LP is said to have *rolled back*. Time Warp [Jefferson] is the first optimistic approach and the term *time warp* is used for optimistic approach and vice versa.

Optimistic approaches have the advantage of exploiting the parallelism in the system and do not suffer from the disadvantages in conservative approach like blocking and processing of null events. But optimistic approaches need efficient mechanisms for state saving and rollback to achieve gain in speedup and efficiency. With encouraging results from 'Weaves and Tapestries' [Srinidhi1], which could be used for state saving and rollback, we study the behavior of an optimistic algorithm in our current system. This section provides the necessary basis for the algorithm we propose.

3.3.1.2.1 Time Window

Pure time warp or unlimited optimism might perform very poorly in many cases especially when the event distribution among the LPs is non-uniform. The larger the difference in virtual time between any two LPs the greater is the possibility of a rollback when the slow moving LP communicates with the faster LP. We try to set a bound on this asynchrony that is inevitable between the LPs in the system, by the concept of a time window. This idea is similar to the Bounded Time Warp (BTW) [Turner] and the Moving Time Window (MTW) [Sokol]. Unbounded optimism also has another disadvantage. The system needs to save state for all virtual time instants less than the GVT. If the window size is unbounded, the amount of state that needs to be saved is also unbounded as it represents the state associated with virtual time difference between the slowest and fastest moving LPs in the system. With a bounded time window, we restrict the difference in virtual times between any two LPs to the length of a time window, thereby setting an upper bound on the amount of state that needs to be saved. Given a time window of size W, each processor computes all events with timestamp in the virtual time window [GST, GST+W]. All the processors synchronize at the end of its time window, i.e. when there are no more events to be processed in the current window. Each LP enters the SYNC state (Figure 3-4) at the end of its window, but this is based solely on local information. In this state, there are no more local events that an LP can process, but processing of an external event could generate more local events.

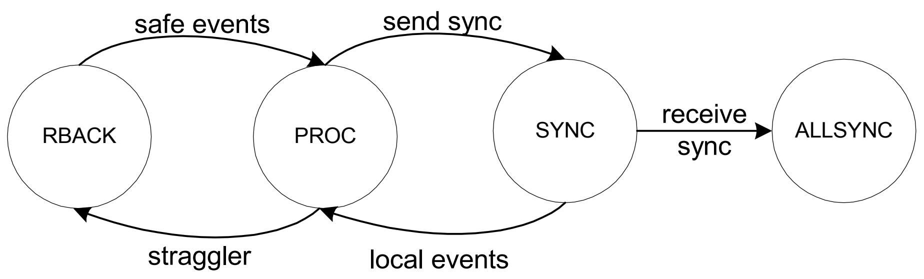

Figure 3-4 Simple State Diagram of a slave processor

Time window algorithms can be considered as both synchronous and asynchronous. They are asynchronous in that the processors do not agree on a particular virtual time to simulate at any instant, but they synchronize at the end of the time window making them synchronous.

3.3.1.2.2 Barrier Synchronization

At the end of a time window all the processors synchronize. All the processors block till they all reach the same step in the algorithm. This is termed as barrier synchronization. Barrier synchronization has the disadvantage of blocking a processor that has arrived at the *barrier* before the other processors. Let W be the size of the time window and GST the Global Simulation Time calculated at the end of the previous barrier synchronization. Each processor after processing the safe events within the time window [GST, W] communicates with the master processor and enters the state SYNC as depicted in Figure 3-4. On receiving external messages, each processor changes back to the PROC state and continues processing safe events and goes to the SYNC state at the end of the time window. From the PROC state on receiving a straggler message, the processor rolls back to a previous safe state and continues. The master processor on receiving 'sync' messages from all the slave processors responds back with a 'sync' message and this completes one cycle in simulation. At the end of this synchronization, all processors in the system are at the same virtual time, and the processors set the GST to the end of the time window and the cycle continues.

3.3.1.2.3 Rollbacks (cascaded, wildfire and echoing)

The degree of rollback in the system is dependent on the amount of parallelism in the application. If the amount of parallelism is limited, a significant amount of rollback is inevitable [Fujimoto]. Fujimoto describes the experiences by researchers in this area as "this usually points to a correctable weakness in the implementation rather than a fundamental flaw in the algorithm". The efficiency of any parallel simulation algorithm is dependent on the cost of rollbacks. In the ONE, there is very little state associated with the simulation layer. State restoration involves cleaning orphan messages from the event heap – messages with the sources undergoing state restoration. But state saving and rollback are complex when the simulation layer is linked with a large number of virtual hosts and their associated applications. When a processor receives a straggler message, it is rolled back to a safe state in the past and continues processing events. The amount of virtual time by which the processor goes backward in virtual time is termed as *rollback length*. After a processor has returned to a safe state, all the messages sent by this processor to other processors need to be invalidated. Messages sent to invalidate the messages sent by a processor as a result of incorrect computation are termed *anti-messages*. The time window optimism bounds the amount of state we need to save and the rollback length. The rollback length cannot be worse than the size of the time window.

Rollbacks may induce further rollbacks in the system making the system unstable. These are referred to as *cascaded rollbacks*. In unbounded optimism (traditional time warp) cascaded rollbacks can propagate to any length backwards and result in trashing of the Logical Processes. A good rollback model is given in [Lubachevsky], which discuss a number of cases that might lead to inefficiency in the system when there is a rollback. Cascaded rollbacks manifest in one of the below discussed forms. We address each of these conditions in our system. Occurrence of cascaded rollbacks is dependent on the cost of processing an anti-message versus the cost of processing an event.

When the number of rollbacks and the number of processors involved in the rollback increases in time, it is termed as *gushing cascade*. An *echo* is a cascade where the processors involved in the rollback does not increase but the size of each rollback tends to increase. Echo is possible in systems that allow pre-emption of events. In our current system we assume there is no pre-emption of events.

In traditional time warp, a processor on receiving straggler message rolls back to a previous safe state and sends *anti messages* to all the processors it has sent messages to. Some straggler messages may results in more than one rollback for a processor and anti-messages may result in further rollbacks and anti-messages. In order to reduce cascaded rollbacks in the system in its various forms, a processor on receiving a straggler message broadcasts a rollback message to every other processor in the system. The broadcast of a rollback to all processors in the system has the disadvantage of unnecessary messages to processors that are not affected by this rollback. When the processors simulate a partitioned network (Figure 3-5) there is no communication between hosts on some processors (LP1 and LP3, LP2 and LP3). In such scenarios the only overhead processing the broadcast of a rollback from LP1 involves the cost of handling an anti-message at the processor that is not affected by a rollback.

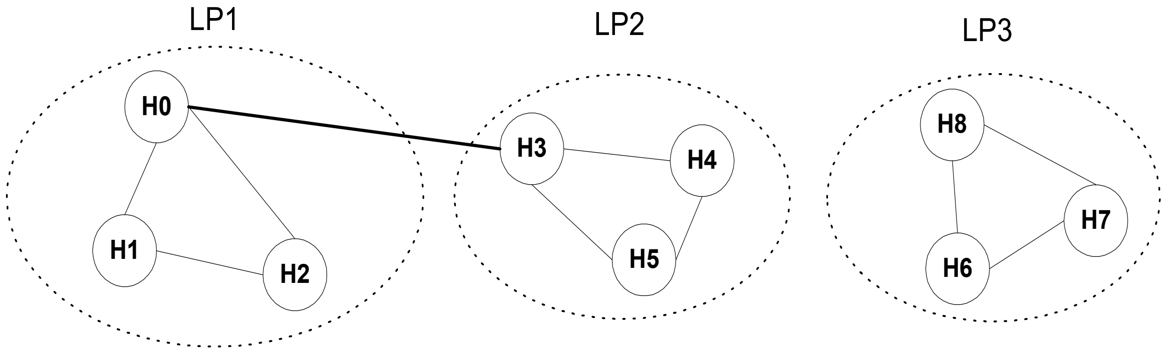

Figure 3-5 Processors simulating a partitioned network

A mechanism in which each processor sends anti-messages immediately after a rollback is termed aggressive cancellation. In lazy cancellation, a processor delays the sending of an anti-message to see if re-executing of the computation generates the same set of messages. Lazy cancellation might result in a faster execution [Fujimoto], but this is completely dependent on the correctness of messages with partial or incorrect input. We follow aggressive cancellation, as the possibility of the generation of same sequence of messages on re-executing the computation is very low. Our choice of aggressive cancellation is further strengthened by the broadcast mechanism supported in the system.

Our rollback mechanism could be summarized as follows:

1. On receiving a rollback broadcast message, each processor changes its state to a safe state (rollback mechanism) and does not propagate this message further.
2. If a processor receives a rollback broadcast to a time greater than its virtual time, it considers the broadcast as an anti-message. All events in response to the incorrect computations are removed from the event list.

3.3.1.2.4 Adaptation

The length of the time window determines the degree of optimism for an LP. As the length of the time window increases from a small value close to zero to a very high value, the simulation algorithm changes from a conservative to an optimistic algorithm. With a time window of size W, where W can range from a unit time to the entire simulation time, the choice of W determines the degree of optimism by any processor in the system. With increasing length of the time window, the number of rollbacks within the current time window increases. Though no claims (including linear behavior) can be made about the pattern of rollbacks, the probability of rollback increases with the size of the time window.

For analysis we can consider the number of rollbacks in a time window as a non-decreasing function with respect to the length of the time window. As demonstrated in the Figure 3-6, an increase in the length of the time window decreases the lost opportunity cost. By exploiting parallelism in the system, larger time windows tend to increase efficiency by reducing the total cost. But a large time window increases the probability of rollbacks and hence incurs an increased rollback cost. An optimal value for the time window is one that lies in region close to the intersection of the two cost curves. A parallel simulation algorithm can choose the size of the time window based on the estimated costs of rollback and synchronization, which are primarily dependent on implementation and the execution environment.

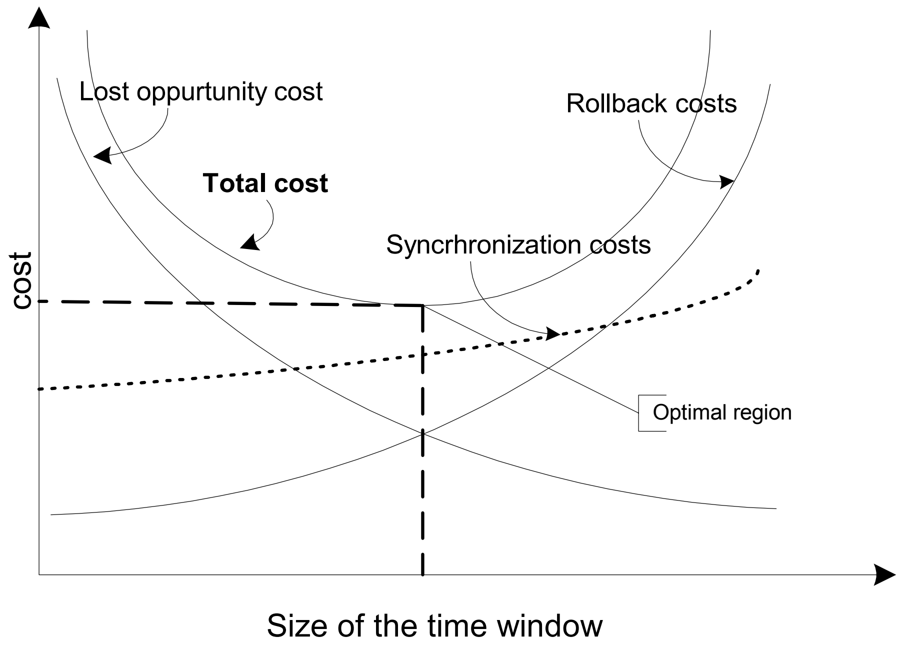

Figure 3-6 Optimal window size and costs

The argument for the synchronization cost not being a constant line in the figure is that as the size of the time window increase, the asynchrony between the processors increases forcing some processors to resend the synchronization message when it receives an external message.

In the previous section we mentioned the existence of a optimal time window. But the term optimal was undefined. We define optimal window as the time window for which the total cost of all the rollbacks within the time window and synchronization is compensated by the gain in speedup achieved by this degree of optimism. [Fujimoto] mentions that such time windows in optimistic mechanism cannot discriminate incorrect computations from correct ones and this might impede the performance. This is a drawback in any time window based approach.

As explained in [Lubachevsky], when the rollbacks cause further rollbacks and these in turn cause further rollbacks, theoretically the number of incorrect computations might explode and cause the system to spend time computing only events generated as a result of rollback and never move ahead in time. But in practice and from previous results, we observe that the rollbacks in the system only dampen the rate of increase in virtual time.

Between a small time window that allows very few events to be processed, forcing the processors to synchronize frequently and thereby reducing the number of rollbacks and an infinitely large window leading to unbounded optimism, we expect that there exists a time window whose length is close to the optimal point from Figure 3-6. If the parallel simulation algorithm is to gain sufficient speedup and exploit any parallelism in the system, the algorithm should operate with a time window that is optimal. A metric that captures the current state of the entire system and aids in choosing the so defined optimal time window size is required. Rollbacks in the system are a measure of the causality in the system and relative rates of progress of each processor in virtual time. A metric related to the occurrence of rollbacks is a good choice.

Other choices include:

1. The event density at the individual processors is another metric that can be used for determining the optimal window size. But the event density does not indicate the nature of causality between any two processors and the goodness of the window estimate is solely based on any heuristics to extend the event density to interactions between the processors.
2. The density of external messages is another choice. This metric illustrates the causality between every pair of processors. However, this metric fails to capture the relative ordering of the rollback messages if any among the external messages.
3. The length of rollback – the amount of time the processor was taken back in time during a rollback - is a metric that captures the degree of causality in the system. However the length of the rollback does not directly translate to the rollbacks or the event density during the interval.
4. The frequency of occurrence of rollbacks or the count of the number of rollbacks in a time window is another metric based on which an optimal window calculation can be made.

No single metric of an LP, which is based on local information can completely and accurate capture the state of the entire system. So the metric chosen should be the processor's best metric to track the status of the global system state. In our proposed algorithm we use the last discussed metric – number of rollbacks in the last time window. The reasons are two fold: the use of this metric has not been studied and this is the best estimate of global system state that can be made by a local. The frequency of rollbacks was only studied as an objective function to be minimized in algorithms like Filtered Rollback [Lubachevsky]. Our objective is different in that we would like the parallel simulation algorithm to operate with a window size that is optimal and this directly need not translate to a time window with lower number of rollbacks.

After our discussion on the existence of an optimal time window, we now explore the possibility of the algorithm adapting to the event causality in the system as it advances ahead in time. The optimal size of the window for some portions of the time line could be small thereby allowing the processors to proceed in a conservative manner and then increase in size allowing the processors to proceed in a completely optimistic manner, where there is not much causality between them. The causality in the system is dependent on the communication in the upper layers of the ONE. As described earlier, traffic in the network that translates to events in the simulation layer, is a superposition of several different traffic sources and sinks each operating with a different periodicity. This gives rise to a varying event rate and varying causality among the logical processors. The size of the time window should be an adaptive quantity that reacts to the changes in the event rate and causality in the system. Any algorithm that attempts to dynamically vary the size of the time window cannot be predictive and it can only be reactive. We assume that the rollbacks in system are an indication of the causality in the entire system and these rollbacks induce one another, so a rollback indicates the possibility of another rollback in the immediate future.

Our adaptive time window algorithm is reactive, i.e. its length shrinks or grows based on rollbacks at the end of previous synchronization. This is general algorithm to which all the other adaptive algorithms can be reduced. Many adaptive algorithms [Tripathi, Chiloa etc] estimate the next arrival of a message on the input channel and the processor blocks for an amount of time calculated based on this arrival rate. This could be viewed as that LP's estimate of the time window for the next cycle.

We further justify our choice of this algorithm for the ONE versus a conservative algorithm with lookahead. From the discussion on lookahead, each processor can calculate a certain amount of lookahead and process all events within the conservative window. In topology distributions where the lookahead available is large, which is possible when the number of virtual links between the processors is small, our proposed algorithm adapts to the reduced causality between the processors and proceeds optimistically and cannot perform worse than a conservative algorithm with lookahead. Further, our adaptive algorithm makes no assumptions about topology and physical characteristics of the underlying network.

The parallel simulation algorithm we propose is a reactive, in that it takes into consideration that rollbacks are an indication of the asynchrony among the processors in the system and estimates the degree of optimism for the next cycle in the simulation. The degree of optimism is based on an optimal time window for which the system estimates that the total cost translates to an appropriate gain in speedup. The estimate for the next time window is based on a simple exponential smoothed average (refer below) of the number of rollbacks in the previous cycles.

<u>Exponential smoothing</u> is a method for forecasting time series with no trend (upward or downward movement that characterizes a time series over a period of time, i.e. long run growth or decline) [Masliah]. Exponential smoothing generates new estimates by adjusting the forecast up or down based on the magnitude of the previous forecast error. The forecast error is given by

$$e_T = y_T - b_0(T-1)$$

e_T is the difference between the new observation T and the estimate for it based on the data through period T-1. Simple exponential smoothing generates an estimate $b_0(T)$ by modifying the old estimate b0(T-1) by a fraction of the forecast error e_T, such that

$$b_0(T) = b_0(T-1) + \alpha[y_T - b_0(T-1)]$$

The fraction α is called the smoothing constant. For simplicity, and to follow the standard notations, we shall define $S_T=b_0(T)$. S_T, called the smoothed estimate or smoothed statistic, is equal to

$$S_T = \alpha y_T + (1-\alpha) S_{T-1}$$

One way to compute the initial estimate, S_0, is take the average of the first several observations

This allows the system to track changes in the causality between the processors to a reasonable effect, though it is reactive. The time window cannot be calculated by a proportional calculation of the number of rollbacks tolerable and the number of rollbacks observed, as there is no simple relation between the number of rollbacks and the length of the time window. So we sample the number of rollbacks observed at regular intervals, at every *unit window* (a constant defined, arbitrarily say 1000units). Interpolating based on the rate of the increase or decrease in the number of rollbacks per unit window is incorrect, as this doesn't capture the effects of synchronization at the end of each unit time window. At the end of each time window, each processor calculates the number of rollbacks observed locally at the end of the first unit time window, end of twice the unit time window, thrice the unit time window etc. It estimates the size of the next time window as the largest optimal time window i.e. the time window for which it has observed an acceptable number of rollbacks. These estimates are smoothed averages of the observations from all the previous cycles as described in sampling algorithm.

3.3.1.2.5 Sampling algorithm

As discussed earlier, the sampling algorithm used to calculate the amount of optimism that bounds the LP in the next lockstep can determine the degree to which the algorithm adapts to causality in the system. The adaptive time window algorithm uses a smoothed average of its earlier observations to calculate the next time window.
Based on its implementation the algorithm could be an incremental, extrapolation, or exponential.

Consider the following scenarios:

- A sudden burst of events, followed by silence. In such a scenario, no sampling algorithm can have enough history to arrive at a good estimate. A pilot run can be used to gather the initial information. But for traffic traces, which have periods of silence and periods of traffic, this cannot be applied, as a pilot run becomes expensive and the information gathered by the pilot run becomes invalid and cannot be applied to the entire length of simulation run.
- A silence followed by a burst of events. If the sampling algorithm is overly conservative and follows an incremental approach, it would take several synchronization phases to choose a larger time window and complete the silence.

When referring to rollbacks in the system, the above two scenarios can be seen as periods of high number of rollbacks and low number of rollbacks. A simulation of network traffic is a collation of several periods of type 1 and type 2. Our sampling algorithm adapts to conditions in the system by sampling the number of rollbacks at periodic intervals for the length of time window. A simulation run with time window length X* UNITWINDOW has samples collected at X points, each spaced at UNITWINDOW units apart. The algorithm maintains samples for all possible window sizes. By the end of X*UNITWINDOWS, there is one sample each of time windows of size UNITWINDOW, 2*UNITWINDOW…(X-1)*UNITWINDOW, X*UNITWINDOW. Note that in a time window of length X*UNITWINDOW, there are no X number of samples for length UNITWINDOW; because all these samples do not account for synchronization. At the end of a rollback all the samples collected for the rollback length are discarded. When there is no enough history for an estimate, the algorithm extrapolates based on the rate of increase in the number of rollbacks in the previous synchronization phase. This allows the algorithm to increase exponentially in situations where there were no rollbacks or very few rollbacks in the previous cycle and no sufficient history to use the smoothed average. This increases the risk of running into a period of high causality with a large value for the time window. The algorithm assumes the capability of the system to checkpoint and rollback state and rollbacks are inevitable in the system. Moreover the gain in picking up the idle periods faster should offset this rollback cost.

Large values of the smoothing constant (α) correspond to quickly damping out the effects of older observations, while small values of α puts stronger weight on older observations. A larger value of gain makes the system reactive to any sudden changes in the causality among the processors, but the choice of α may be application specific. An appropriate value for α can be found by computing the squared errors between the estimates and the observed values. As our performance results show, moderate values for the gain were suitable for tracking down changes in the degree of causality and to estimate an optimal window size for the next cycle.

Finding an optimal window size for every cycle in the simulation that ensures optimality over the entire simulation run is a complex function. With only limited, local information and past history, the best a processor can estimate is a time window that is locally optimal. The problem of splitting the entire simulation run into time windows, such that the number of rollbacks in the entire simulation time is optimal is NP-Hard. Even an offline algorithm to calculate the optimal time window for one cycle in the simulation run cannot guarantee global optimality. As we describe an *oracle* in Chapter 4, we use a greedy algorithm to find a time window that is either a first or a best fit in the entire solution space.

3.3.1.3 The adaptive time window algorithm

With all the necessary rationale explained, we describe our parallel simulation algorithm. Figure 3-7 demonstrates the algorithm in detail.

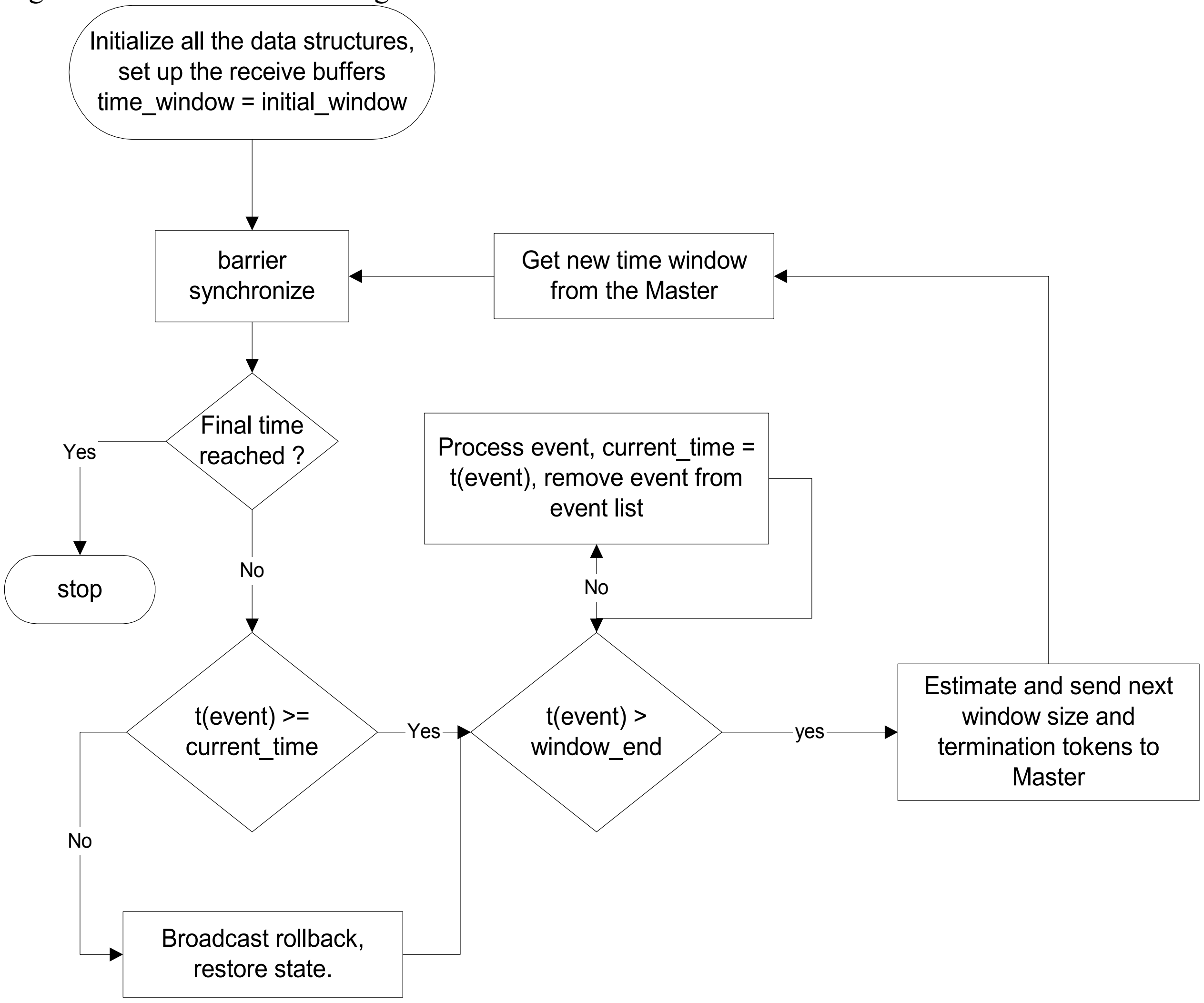
Initialize all the data structures, set up the receive buffers time_window = initial_window
barrier synchronize
Get new time window from the Master
Final time reached ?
Yes
stop
No
Process event, current_time = t(event), remove event from event list
No
t(event) >= current_time
Yes
t(event) > window_end
yes
Estimate and send next window size and termination tokens to Master
No
Broadcast rollback, restore state.

Figure 3-7 Algorithm for the slave processors

3.3.2 The simulation engine

From our analysis and performance of our parallel simulation algorithm and the requirements of the system under study, we built the simulation engine for ONE. The simulation layer runs the adaptive time window algorithm discussed. Given the topology of the underlying network, the simulation layer maintains mapping of the hosts to processors and the host connectivity information. The personality layer maps the functions to transmit a message to the appropriate functions of the simulation layer. In UNIX based operating systems, this involves changing the function pointers. For example in Linux OS the function pointer `hard_start_xmit` is changed to the simulator layer's transmit function.

Registration of devices: To differentiate among the multiple devices connected to a host, the unique components of the network topology to the simulation layer are devices. Every host registers the device with the simulation layer during the initialization phase. Implementation details of the simulation engine are discussed in Chapter4.

3.4 Justifying our assumptions

The testbed is an interconnection of workstations also referred to as Network of Workstations (NOW) connected by a high-speed system area network – Myrinet. Our choice was based on the performance of Myrinet over the Gigabits Ethernet and cost. Our assumption that our system is fault tolerant is justified by some of the features of Myrinet [Pham-99]:

- The hardware provides an end-to-end flow control mechanism for reliable delivery.
- Message losses are in general very rare making our assumption of a loss less network justifiable.

The centralized master processor coordinates between the other processors (slaves) during synchronization, adaptation and load balancing. This master processor is a possible single point of failure. The cost of communication using a completely distributed algorithm, where each processor (slave) communicates with every other processor in the system when it has reached the end of its time window is $O(n^2)$ Given a reasonably fault tolerant system, the message complexity cost of a centralized server implementation is significantly less than a distributed implementation. Our main focus is on building a parallel simulation layer and not on making the system fault tolerant. When any of the slave processors, which are the active elements concerned with simulation fails, the entire system has to be restarted. The master processor is only concerned with synchronization, adaptation and load balancing and has no state information associated with it. When a master processor fails, a new processor could be assigned the role of a master, it sends a control message to all the slaves. The slave processors respond to the new master processor and resend any information that was sent to the failed master and the system continues to operate.

3.5 Summary

In this chapter we have given the necessary background for understanding the overall design of ONE. We discussed possible approaches to solve the synchronization problem in PDES algorithms with emphasis on the system under study. We propose a new bounded time window algorithm that adapts to changing rollback conditions. We discuss the assumptions made and justify them in the context of this research. The next chapter presents a comprehensive performance evaluation of the proposed algorithm.

Chapter 4

4 Performance Evaluation

The primary goal of the current work is to analyze the different approaches to parallel simulation and arrive at a good algorithm for the simulation layer of ONE. The objective of performance evaluation is to support our proposed algorithm and the inadequacies in directly applying some of the traditional algorithms to our problem domain.

This chapter discusses the results of our proposed parallel simulation algorithm – the adaptive time window algorithm. We strengthen our choice of the algorithm by analyzing the results from traditional approaches. All analysis is performed using emulated traffic traces. We discuss the performance of a strictly conservative algorithm and a time window approach. We show the requirement for adaptation in the length of the time window and results of the adaptive time window algorithm. The results of the adaptive time window algorithm are compared with offline oracle algorithms for its behavior. In this chapter we also discuss the metrics we use in the adaptive time window algorithm. Implementation details of the various algorithms and the simulation engine are discussed. By proper choice of the values for the metrics, any parallel simulation algorithm can be analyzed for the target system.

4.1 Experimental setup

All the experiments were run on ***Anantham*** – an eighty-node Myrinet cluster. Each of the individual nodes is a workstation with an AMD Athlon 650 MHz processor and 256 MB of main memory. Figure 4.1 shows the physical connectivity of the workstations and switches of the cluster. The nodes are interconnected over a switching fabric based on seven 16 port Myrinet switches connected over a 2 level tree. This fabric does not run IP.

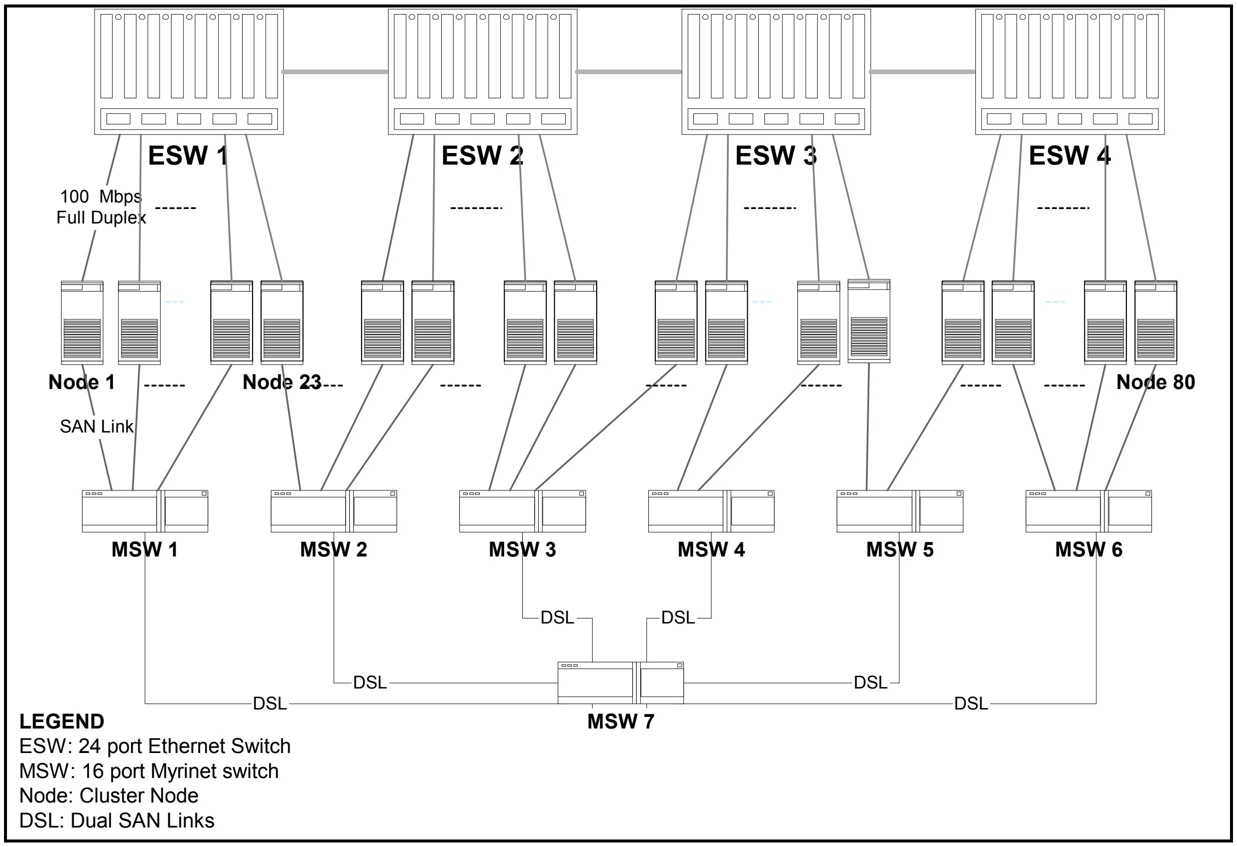

Figure 4-1 Anantham Cluster Connectivity

4.2 Input Traffic

None of the traditional parallel simulation algorithm or its variants could be chosen for the simulation layer of ONE. Restricted analysis of some of the approaches makes their behavior unanalyzed for the current experimental setup. Even a complete analysis of a parallel simulation algorithm cannot be used in extrapolating its behavior to an emulation environment. This is due to inadequacies in the event modeling used in the analysis of parallel simulation algorithms. In network emulation/direct code execution environments, generalized event models are complex and have a clear notion of self-similarity, whereas traditional parallel simulation algorithm analysis has relied on event models that have a statistical notion of periodicity and/or causality. Hence our analysis of parallel simulation algorithms for ONE uses real traffic traces to generate simulation events, maintaining the real-world self-similar behavior of network traffic.

4.2.1 Difficulties in Input traffic generation

Generating network traffic for analyzing a parallel simulation algorithm for an emulation environment is a complex task. Tools like *tcpanaly* and *tcpdump* can be used to capture and analyze TCP traffic dumps, which can then be used as event traces for network simulation. However, in the current case, traffic capture itself poses a major **serialization** problem. When a network traffic trace is captured at a single point in the network, the time stamps and events in the traffic trace are serialized due to the bit serial nature of the network interface used to perform the capture. This trace hence cannot be used to emulate multiple virtual hosts, since it has no concept of temporal parallelism. For collecting traffic traces to mimic N hosts, we actually need to collect traffic at N points within the network – true emulation requires inputs true to the real world. Additionally, the simulation layer imposes a global clock on all the participating nodes. Hence, the clocks on all these N points in the network should be synchronized. In a local or a wide area network the Network Timing Protocols like NTP can be used to achieve clock synchronization. Even in such a setup, the skew in the clock times might cause the time stamps on the traces to drift. With these difficulties, the possible solutions are to:

- Collect traffic traces at traffic aggregation points like routers and distribute the traffic among the hosts. This wouldn't capture the causality in the events completely, but this approach is feasible.
- Collect traffic at each individual host of a 'small-intranet' running in a lab environment.

4.2.2 Experimental Traffic Traces

With the difficulties that arise in generating input traffic to analyze parallel simulation algorithms for ONE and the possible solutions from previous section, we now describe the two traffic traces used to analyze our parallel simulation algorithm.

- Trace 1: Emulated traffic trace, generated by running an Ethernet spanning tree algorithm. This trace shows periods of high causality among all the Ethernet switches and periods of idle times. We refer to this trace as '**Ethernet Trace**'.
- Trace 2: Real network traffic traces from a small Intranet. Applications like File Transfer Protocol (ftp), Web browsing (http), X-Windows, Telnet are run on the workstations of the Anantham cluster. The traffic was captured using the *tcpdump* tool. All nodes of the cluster are time synchronized using the Network Time protocol. The traces are converted to logical host traces, by assigning logical ids to each host. Transmit and receive times of the packets are collected using the time stamps on the packets and packets are matched at the two ends of the pipe, by comparing the first few bytes of the protocol headers. We clean the traces for possible time drifts. Hosts start TCP based applications like ftp, telnet and web browsing to servers running on some hosts of the cluster. The browse times for

web pages, file length for ftp, words per minute for Remote shell etc are set to typical values. While the collected traces attempt to emulate real network traffic conditions, no guarantees can be made to their exact authenticity. The traffic trace is varied so as to capture the characteristics of typical network traffic. We refer to this trace as '**Web Trace**'.

When extending the results from a trace based analysis to the simulation engine we expect a better performance, as the number of rollbacks in our trace-based analysis is an upper bound on the number of rollbacks that occur in the real environment. Because of non-dependent transmits in a trace file analysis we experience far more rollbacks than in the target environment.

All event-handling costs are zero in the experiments. We only analyze the slow-down from our proposed algorithm. In this setup, we pay the cost of communication, without gaining anything from parallelism in computation. The goal is to minimize the slowdown. If event handling had non-zero computation costs, the speedup gain would be heavily application depend.

4.2.3 Trace Driven Simulation

All the experiments were run on the traces described above. We digress to discuss briefly about trace driven simulation. A simulation using a trace as its input is a trace-driven simulation. The trace is a record of events ordered by time on a real system. [Jain] gives a very good discussion on the advantages and disadvantages of a trace driven simulation. We discuss the factors that are of particular importance to our experiments. The following are the advantages in using a trace driven simulation for our experiments.

- Credibility: the results of trace driven simulation are more convincing, as the trace has more credibility than references generated randomly using an assumed distribution.
- Accurate Workload: The trace preserves the correlation and interference effects in the workload (traffic). There are no simplifications such as those needed in getting an analytical model of the workload.
- Less Randomness: The trace is a deterministic output. But repeated runs of the simulation might results in different output due to randomness in other parts of the model (in our case the underlying communication system). Overall output of trace driven model has less variance and is possible to obtain absolute results if other parts of the system are not random.
- Fair Comparison: The trace allows different alternatives (algorithms) to be compared under the same input stream. This is a fairer comparison than other simulation models in which input is generated from a random stream and is different for the various alternatives that are being simulated.

On the other hand trace driven simulations suffer from the following disadvantages.

- Complexity: trace driven model requires more detailed simulation of the system. This sometimes might overshadow the algorithm being modeled. In all our experiments, computations outside the scope of the simulation algorithm are made minimal.

- Finiteness: results based on a detailed trace of a few minutes of activity on a system may not be applicable to activities during the rest of the day.
- Representativeness: traces taken on one system may not be representative of the workload on other systems.
- Single point of validation: traces give only one point of validation each. So an algorithm best for one trace may not be the best for another. To validate results one should use a few different traces.
- Detail: main problem is the high level of detail. Traces are generally long sequences that have to be read from disk and computation for each record of the trace. To reduce the level of detail in the trace records, in the traces described minimal information like the time stamp, host information and message lengths were collected.

4.3 Experimental Results

4.3.1 Conservative Algorithm

We implemented the strictly conservative algorithm explained in Chapter 3 in our experimental environment. The algorithm was run with both trace1 and trace2. From the event times, the only parallelism the algorithm can exploit is by events scheduled at the same virtual time. From the event traces collected for both traces, the number of events scheduled for the same instant in time is very low, close to zero. This makes the system run at the rate of one time instant every lockstep and an increased block time for each of the processors. All implementations of the strictly conservative algorithm like [Cilk-97] which show speedup, are with event traces from simulated traffic and the granularity of the simulator is low. For the algorithm to perform well, the granularity of the simulator is reduced, thereby collating several events for the same simulation time. The system loses fidelity at a lower granularity. It is not likely to hold good when the event traces are from emulated traffic sources like the ones used in our experiments. Also when run at a granularity of nano seconds, the number of events along the entire system scheduled for the same time instant is very low.

4.3.2 Optimistic algorithm

An implementation of an optimistic parallel simulation algorithm requires the capability of saving state and restoring state when a rollback occurs. In a system that is a complete simulation based, optimistic algorithms like pure time warp can be implemented and studied. This is because the modeler has control and access to every module in the entire system. In the current system, this is not likely to be true. So there is no possibility of a user-level check pointing to support state saving. The Weaves-Tapestries model allows the check pointing of the system at any particular instant in time using a compiler-based framework [Srinidhi1]. In a system where the change in state between two particular instants of time is very high, the memory and computational power of the processors limit the amount of state that can be saved. With huge amount of processing required to save and restore state, the system could be thrashing between state saving and restoration. Although the parallel environment increases the amount of memory available, it is bounded. So an algorithm like the pure time warp, in which the amount of memory required is unbounded, is not feasible for a complex system like ONE. These make a pure optimistic algorithm for the simulation layer of the ONE an impractical approach. The closest practical solution to the optimistic algorithm in this case is to place a bound on the amount of optimism. This bound on the amount of optimism is directly dependent on the computational power and the memory available. These we consider under the assumption that the optimistic approach is a viable solution for the simulation layer of ONE. The efficiency of the state saving and restoration mechanisms directly affects the performance of the algorithm.

An alternative to checkpoint and restore is based on the concept of reverse compilation. Reverse compilation automatically generates the functional inverses of simulation code. The inverses are used to derive a backward path from some current state to a previous state in the past. The main problem with this approach is its assumption of the existence of an inverse function. Mathematically, many functions, including common network operations such as many-to-one hash functions are non-invertible.

As pure optimistic approach is impractical, we restrict the optimism at each LP by a time window. A time window of length W allows the processor to handle events scheduled for the time between GST and GST+W. As it was discussed in chapter 3, the length of the time window determines the performance of the parallel algorithm.

4.3.2.1 Static Time Window

Implementation with a simple restoration for state saving and restoration and placing arbitrary bounds on the amount of optimism and its performance on the Ethernet spanning tree trace was studied. The simulation was run with varying sizes for the time window. There is no complete justification for the size of the Window and is a choice of the modeler. We discuss the size of this time window in future sections.

Table 4-1 shows the results from running the Ethernet trace on 8 processors using a static time window algorithm. All values are in units of microseconds. As discussed earlier, the choice for the length of the time window in this case is arbitrary.

Time window (microseconds)	Simulation time (microseconds)	Execution time (microseconds)	# Rollbacks
2000	20000	4794.186	2000
5000	20000	2760.685	7566
9000	20000	2372.778	12745
15000	20000	2623.296	20314

Table 4-1 Static Time Window using Ethernet Trace on 8 processors.

From the table and the static time window algorithm, it can be inferred that for smaller simulation runs, a lower value of the time window could result in a lower execution time. But as the simulation time grows larger, smaller time windows incur more synchronization cost and result in higher execution times. As the number of rollbacks increases with the window size the choice of the time window becomes a complex function. From the above table it could be inferred that a time window with size between 5000 and 9000 could possibly be an optimal window length. But there is no known mechanism for choosing the length of static window *a priori*. This led us to an algorithm that is adaptive in its window length and based on a statistic that captures the effect of increased window length on the number of rollbacks.

4.3.3 Adaptive Time Window Algorithm

After studying the performance of both the conservative and the optimistic approaches, we studied the performance of a window-based approach. Results from the static time window implementation suggest a time window that adapts with the number of rollbacks in the system. In our adaptive window algorithm, the amount of optimism is varied with the causality in the entire system as recorded in the previous lock step. The number of rollbacks reflect causality in the system. A time window is computed at the end of very barrier synchronization phase, which takes into account the causality in the system and the costs associated with state recovery.

Here we briefly discuss some of the parameters that determine the parallel simulation algorithm. These parameters are both an implementation and a design issue. The performance of any specific parallel simulation algorithm can be studied for the target environment by determining appropriate values for the various parameters. The following parameter values are specific to the algorithm and should be determined to suit the algorithm. The parameters are:

1. *Granularity of the simulator*: The simulator granularity determines the precision of the time-based calculations in the simulation layer. For example, based on the input traffic trace, the granularity can be set to the nanoseconds or microseconds. The simulation layer granularity affects other parameters discussed below. All the parameter values are described in units of this granularity scale.

2. *Unit time window*: This is the smallest time window that the system runs during its entire run of simulation. Heuristics like the average inter event time and the average cost of rollback can be used to determine an initial value. For example in the web trace runs, we set a unit time window to 1000 units. The unit time window determines the performance of the system in worst-case situation. This is the lowest degree of optimism that each LP can operate. As most of the other parameter values directly depend on the unit time window, the choice alters the performance of the algorithm.
3. *Maximum time window*: This is the maximum length of the time window that the system operates. The value for this parameter can be chosen based on the amount of memory available. If events are scheduled for every time instant of the time line, each processor should be capable of saving the state of simulator and its associated 'Weaves and Tapestries'. But the higher the value of this parameter, the more each LP could exploit optimism and idle periods in the system.
4. *Synchronization to Rollback Ratio*: This value determines the affordable number of rollbacks of the system. This is the length of the simulation time window that could afford a single rollback. The rollback mechanism and its efficiency determine this parameter. Each processor's estimate for its affordable number of rollbacks at the end of synchronization phase is a calculation based this value and its previous samples. A very efficient rollback mechanism could set this to a very low value.

The results obtained by running the adaptive window algorithm on the Ethernet spanning tree trace gives a strong support for the appropriateness of an adaptive algorithm. From this trace, it is clearly shown that the adaptive algorithm very quickly picks up to a very large window size when there is idle time in the system. This is observed for the few time instances in the Ethernet spanning tree algorithm, when the nodes stay idle, basically to allow other nodes in the network to stabilize. In time intervals when the events are very closely spaced, the likelihood of rollback is high and this would effect the calculation of the time window for the next lockstep. This algorithm completely adheres to our initial idea that rollbacks in the system are indicative of causality among the processors. To speedup idle simulation time, the algorithm uses an exponential incremental scheme and it can increase to a maximum value. The bound on the maximum window size is required even in the adaptive algorithm as it is reactive. We compare the performance of the algorithm with another approach we introduce - *Oracle Computations*, in the next section.

Absolute Speedup S is defined as Ts/T(n), where Ts is the speed of sequential program and T(n) is the speed of parallel program with n processors. S lies between 0 and n. Absolute Speedup is a criterion for performance of parallel algorithm. Absolute efficiency is S/n and it should lie between 0 and 1. Relative speedup and efficiency use T(1) instead of Ts, i.e. the speed of the parallel algorithm when run on a single processor. Relative speedup and efficiency are larger than the absolute counterparts. Ideally the speedup should equal n the number of processors. However, due to communication costs, sequential bottlenecks and computational tasks required of a parallel application that are not necessary on a single processor, the speedup s is often less than n. As the number of processors increases, message passing costs and synchronization costs increase, the speedup falls. In general the size of the problem grows as the number of processors applied to the problem increases and the value of s usually decreases. As we operate only at zero event-handling cost, we restrict the analysis of our speedup curves to comparing the effect of synchronization costs on the total cost. Moreover a larger value for speedup does not guarantee a faster run time. This depends on the work assignment to each node, i.e. the load balancing on each node.

Another execution metric is in terms of *slowdown* of the best efficient simulation between the guest and host networks. Guest network is the network under study that is being simulated and host network is the network used to simulate; typically the experimental setup. The slowdown s = Tg/Th is the ratio of the simulation time on the host Th to the running time on the guest Tg. The goal is for the host to perform the simulation of the guest as quickly as possible without expending more than a constant factor more work.

4.3.4 Oracle computations

For a comparative analysis of the adaptive time window algorithm, we introduce the idea of an Oracle. The Oracle is an offline algorithm that tries to find an optimal window size for each cycle in the simulation. As explained in Chapter 3, finding globally optimal window sizes for every cycle in the simulation is an NP-Hard problem. So the oracle attempts to find a time window that is a local optimal solution. In selecting a local optimal solution, the oracle computations can choose a best – fit or a first fit algorithm.
In a best-fit algorithm the oracle runs the simulation with all window sizes starting from a MIN window size to a MAX window size. At the end of the simulation run, the best window size is determined. The best window size is chosen as

1. The time window that has an affordable number of rollbacks and
2. All time windows larger than this resulted in a higher number of rollbacks than the system could tolerate.

The first condition is because the window chosen should be at least as conservative as to not exceed the tolerable number of rollbacks. The second condition is due to the oracle's best-fit approach. When a larger time window causes the same number of rollbacks then the oracle chooses the larger time window, as the cost of rollbacks is already paid.

In the first fit algorithm, the oracle runs the simulation, by increasing the length of the time window gradually and it chooses the first time window beyond which the number of rollbacks in the system is beyond the tolerable number. It is possible that the first fit and the best fit choose the same window size, this could happen, when increasing optimism beyond a particular window size always results in rollbacks that are not acceptable for that time window. Also from Figure 4-2, it can be observed that the best-fit algorithm chooses a larger time window than the first fit as the number of rollbacks are still acceptable for this large time window, than the one chosen by the first fit.

From the oracle computations on the Ethernet trace, the first and best fits choose window sizes almost similar in length. The window size as seen from Figure4-2, constantly changes between a low value around 1000 and a value around 14000. This can be attributed to the causality in the trace. As the events among the processors increase, the oracle computations chose a very low time window.

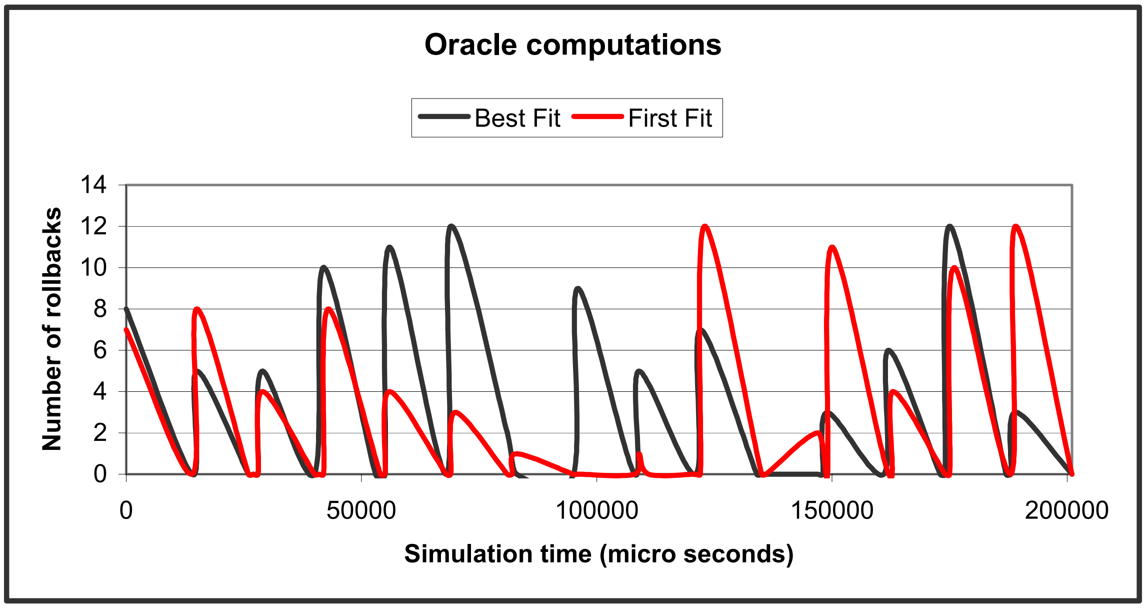

Figure 4-2 First and Best Fit for Ethernet Trace, 4 processors

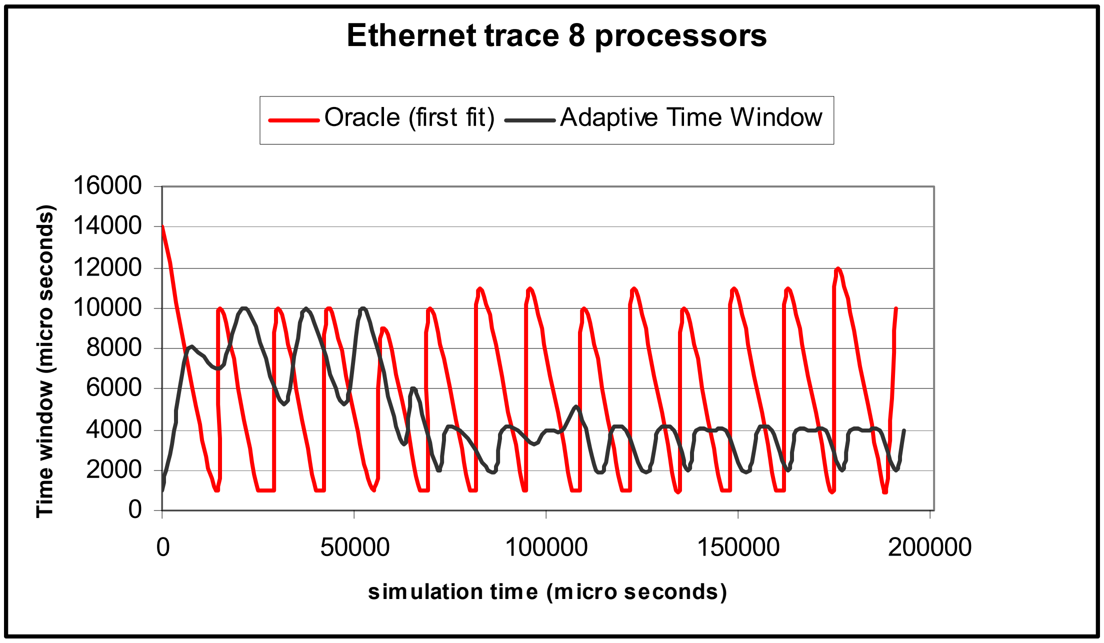

Figure 4-3 Plot showing length of time window for a partial run of Ethernet trace on 8 processors

Figure 4-3 shows the results from a simulation run with the adaptive time window algorithm and the oracle computations (first fit) algorithms. From the figure it can be inferred that the initial estimates of the adaptive time window algorithm result in a larger time window compared to the first fit algorithm. As explained in the sampling algorithm, the adaptive time window algorithm during the initial stages behaves aggressively in probing the optimal time window. As it gains sufficient history, it behaves in a more conservative fashion and estimates the time window, comparable to the first fit algorithm. From the figure it can be noted that the first fit algorithm, with a complete information about the systems behavior for all sizes of the time window, varies its optimal time window between a very close to 1000 (the UNITWINDOW) and close to 10000. The adaptive algorithm, on observing unacceptable number of rollbacks for a larger time window, runs in a very conservative manner so as to reduce the rollbacks. As simulation continues, the adaptive time window algorithm's estimate for the time window is damped after initially choosing a large value for the time window that caused a large number of rollbacks. This can be observed from Figure 4-4.

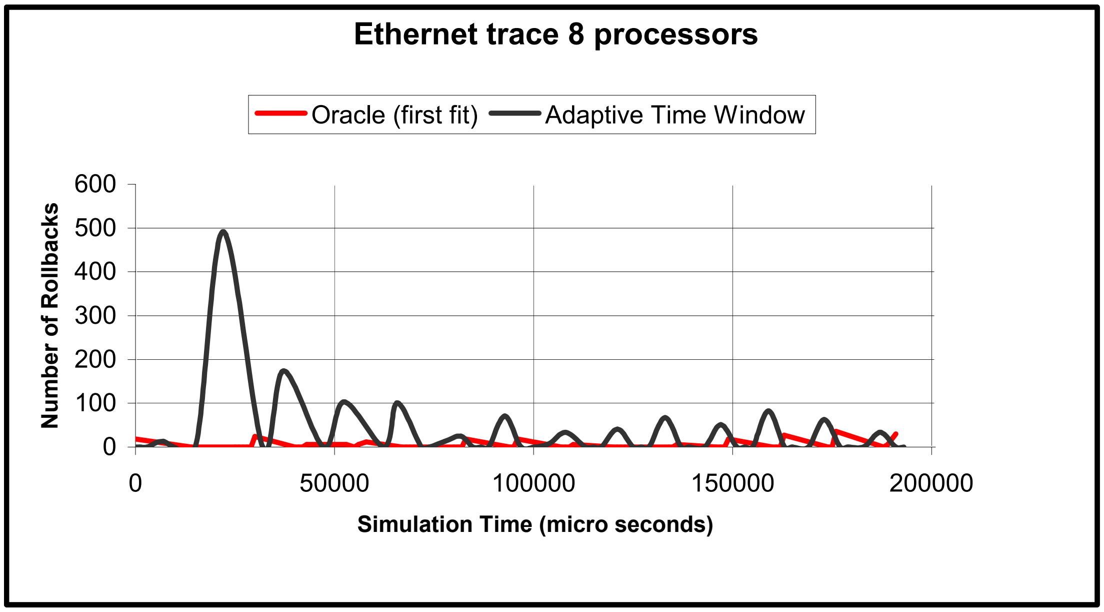

Figure 4-4 Plot showing number of rollbacks for a partial run of Ethernet trace for the Oracle and the Adaptive Time window algorithms

Referring to Figure 4-5 and Figure 4-6, the adaptive algorithm exploits the idle periods in the simulation run. In comparison to the Oracle computation, the adaptive window only grows to the Maximum Window Size in an exponential manner. As the Oracle has more state information, it shoots up to the Max Window Size. Referring to Figure4-5: When the degree of causality in the system increases, the adaptive time window is affected by the change causing a large number of rollbacks. In comparison the oracle chooses a smaller time window resulting in a fewer number or rollbacks. The Adaptive time window algorithm can react to the changes in degree of causality in the system only at the end of its current synchronization phase. This justifies the difference in the behaviors of the first fit and the adaptive time window algorithms.

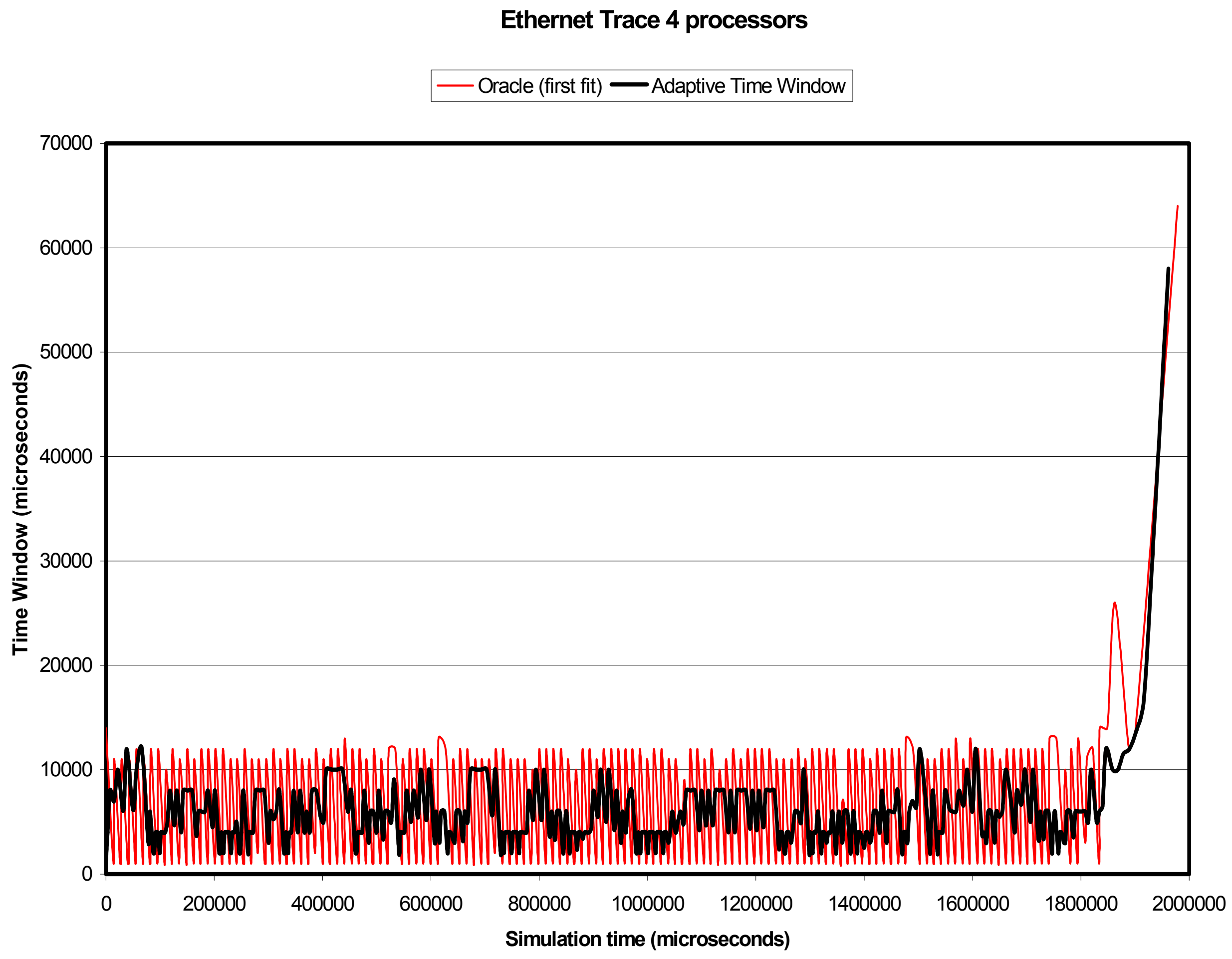

Figure 4-5 Plot showing the length of time window for Ethernet Trace on 4 processors.

The damping factor α determines the rate of decrease in the length of the time window as a result of these increased rollbacks. In certain traces, like the Web trace, the adaptive window algorithm shows to be more conservative than the oracle. A possible reason could be the choice of the damping factor α, which has no relevance in the Oracle computation. This might lead to a possible choice of a static time window algorithm, avoiding the cost of adaptation. Static time window is easier to implement and a good balance of conservative and optimistic approaches. As Table 4-1 shows, static time window algorithm could result in a high number of rollbacks and possible trashing. This reasoning against the static time window algorithms comes from our observation that rollbacks tend to cause further rollbacks. So limiting the degree of optimism that resulted in more than affordable number of rollbacks in the previous cycle is preferred to a constant degree of optimism. The possible instance when the adaptive algorithm cannot outperform the static time window is when the rollbacks, i.e. causality in the system varied very fast and it did not extend to the next time window. In such an instance, the adaptive time window algorithm reduces its time window and incurs a more lost opportunity cost than the static time window algorithm.

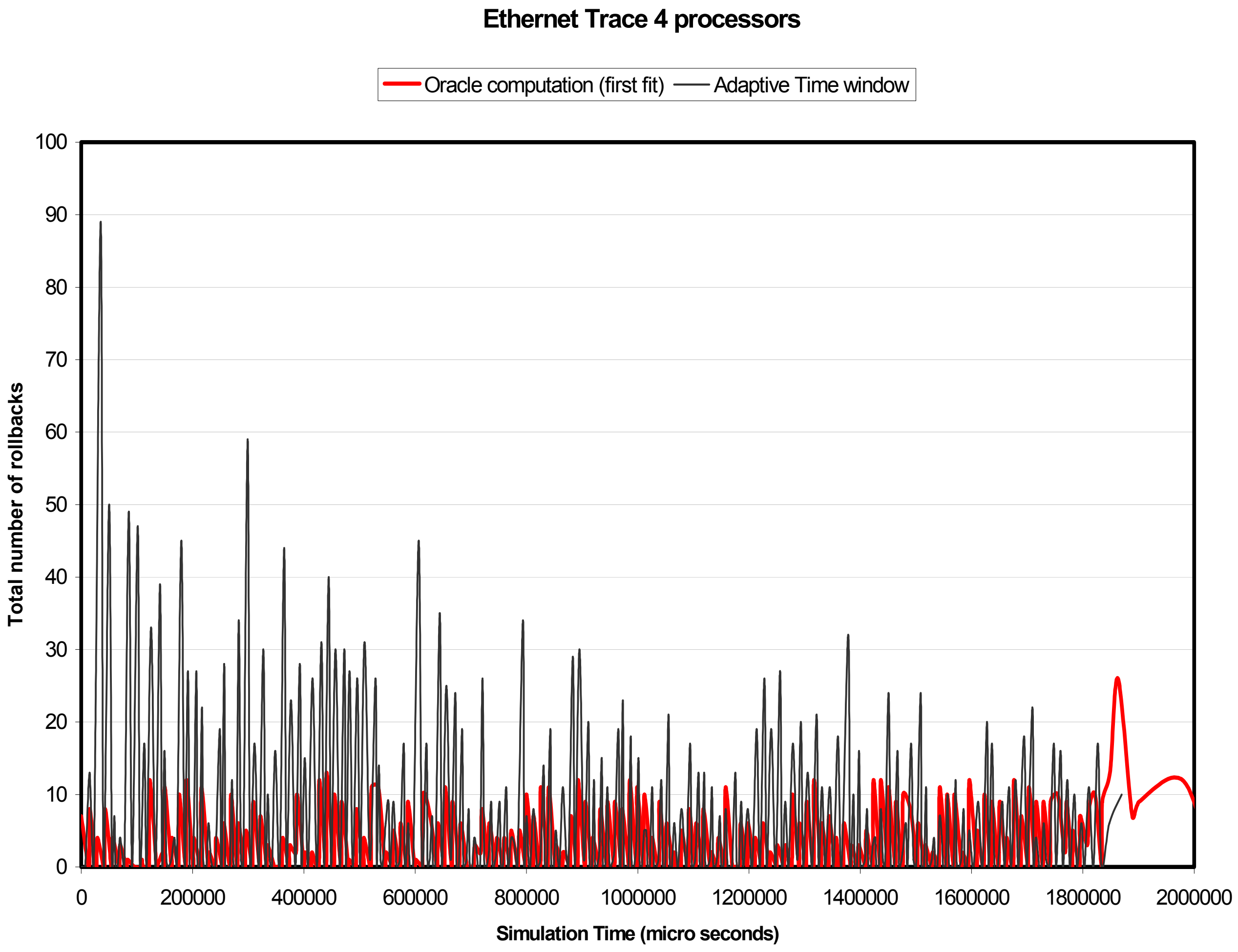

Figure 4-6 Plot showing the number of rollbacks for Ethernet Trace on 4 processors

It can be inferred from Figure 4-6 that during the initial stages of the simulation run, the adaptive time window algorithm does not have sufficient history for extrapolation. In this scenario, the adaptive time window algorithm tries to probe for the optimal time window in a exponential manner and as a result of this aggressiveness results in a far more rollbacks than the oracle. But as the adaptive time window algorithm builds history, it estimates window sizes comparable to the oracle computation and this reduces the number of rollbacks. This can be inferred from the trend of the rollbacks. Based on the definition of a tolerable number of rollbacks, the algorithm (Figure 4-7) stays at this value for a large period of time. During such runs, the window size is almost constant and this is the same window size probed by the oracle algorithm also. Figure 4-8 shows the comparative plot of the number of rollbacks for such a simulation run.

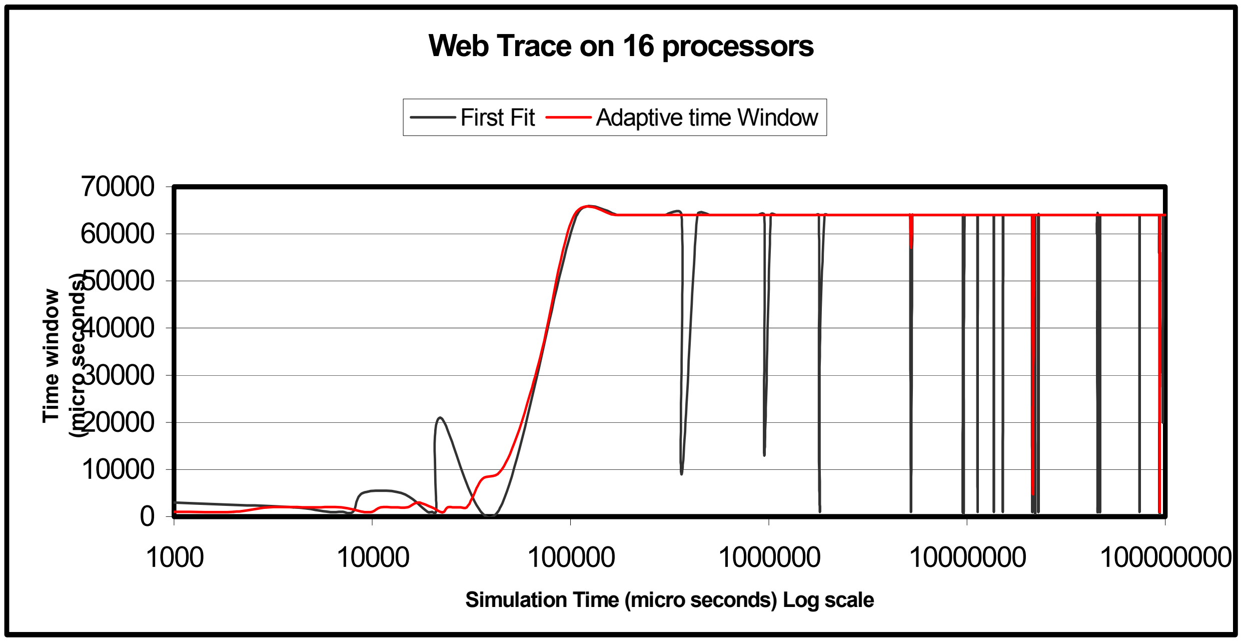

Figure 4-7 Simulation runs of the First fit algorithm and the adaptive time window algorithm for Web Trace run on 16 processors. Simulation time plotted on a logarithmic scale

The adaptive time window algorithm can be tuned by changing the estimating policy. When the sampling algorithm lacks sufficient history, it extrapolates based on the increase in rollbacks in the past. This can be replaced by a more conservative approach, say by incrementing by a unit time window. The damping factor α can be varied to change the algorithm's response to recent rollbacks in the system. The effect of changing α can be studying the total number of rollbacks for the entire simulation run and the execution times. Table summarizes the details of the simulation runs with different values for the damping constant α.

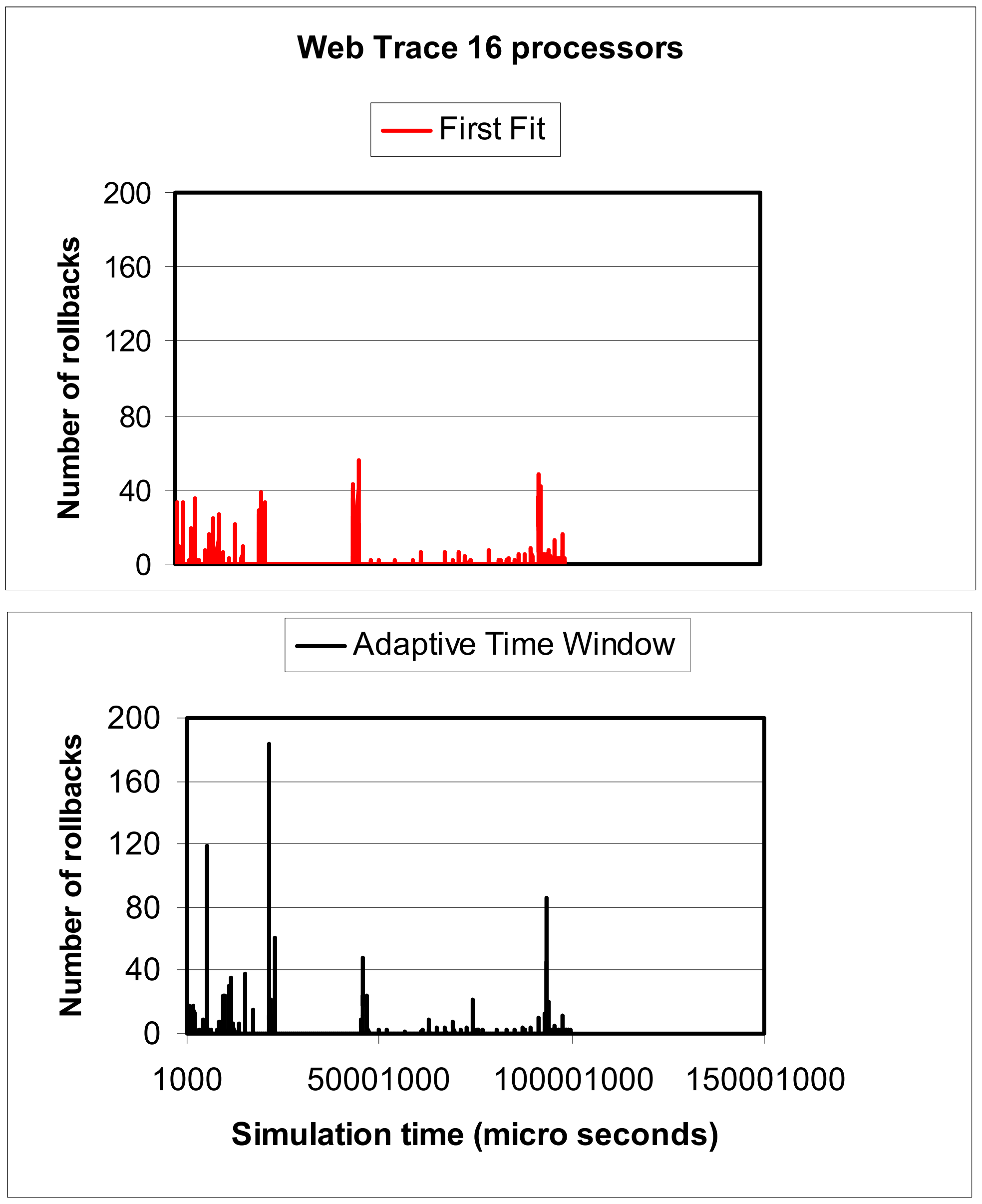

Figure 4-8 Comparative plots showing the number of rollbacks for Web Trace on 8 processors. The upper plot is for the Oracle (first fit) shown in red and the lower plot is for the Adaptive Time Window algorithm.

Gain α	Execution Time (microseconds)	Total Number of Rollbacks
4 processors		
0.6	773284	4581
0.7	767923	4623
0.8	779411	4813
0.9	762395	4635
8 processors		
0.6	852848	4769
0.7	1060405	4551
0.8	943636	5565
0.9	993309	5282

Table 4-2 Execution Time on the Anantham cluster for the Ethernet Trace. Results gathered from a simulation run for 4000 milliseconds.

Gain α	Execution Time (microseconds)	Total Number of Rollbacks
8 processors		
0.6	2660607	3406
0.7	3168438	3716
0.8	2656656	2927
0.9	3509547	5687
16 processors		
0.6	3226999	1373
0.7	3189524	1442
0.8	3482858	1612
0.9	4090474	2097

Table 4-3 Execution Time on the Anantham cluster for Web Trace. Results gathered for a simulation time of 100000 milliseconds.

Processors	Execution Time (microseconds)	Total Number of Rollbacks
Web Trace		
4	744778	1812
8	1187306	1938
16	2629307	1988
Ethernet Trace		
4	690921	922
8	619049	859
16	786475	768
32	1546231	635

Table 4-4 Execution Time on the Anantham cluster for the Web and Ethernet traces from the oracle computations (first fit). Web trace results are from a simulation run of 100000 milliseconds and Ethernet trace from 4000 milliseconds.

4.3.5 Other performance metrics

We also studied several other performance metrics apart from the length of the time window and the number of rollbacks. They are cost metrics – lost opportunity cost and synchronization cost.

4.3.5.1 Lost Opportunity Cost

This refers to the cost associated with loss of optimism. This is the cost incurred by each LP when it was unable to choose an optimal value for the current time window. None of the related work from Chapter 2 discusses the experimental details of this metric. We define the lost opportunity cost in terms of real system time. Though this is limited by the granularity of the system clock, for comparative purposes it still holds good. For a particular cycle in the simulation run, the oracle would not take longer than adaptive algorithm to run. This could be determined by the system clock differences at the start and end of the time window. The lost opportunity cost for both the algorithms cannot be compared at each synchronization phase. The reasons for this are: As the time window lengths differ, the synchronization points in both the algorithms are shifted by the difference in the lengths of the time window. Sampling cannot be used, as the statistics are available only at the end of each synchronization phase. The difference in the run times of the adaptive and the oracle algorithms as a fraction of the time taken for the oracle algorithm is a good indicator of the lost opportunity cost. As the adaptive time window algorithm only tries to operate close to the optimal algorithm, the smaller the difference in the time taken for the simulation run, the lower is the lost opportunity cost. Tables 4-2, 4-3 and 4-4 tabularize the various simulation runs and the lost opportunity cost.

Processors	Synchronization/total time ratio
Web Trace	
4	17.33
8	21.52
16	24.75
Ethernet Trace	
4	19.64
8	22.63
16	25.42
32	23.45

Table 4-5 Ratio of synchronization time to the total time for simulation.

4.3.5.2 Synchronization Cost

With no event handling cost, the cost of synchronizing at the end of every time window is high. The underlying hardware and the synchronization algorithm affects the synchronization cost. The synchronization algorithm to decide the time instant when all LPs have reached the end of the current time window and there are no messages in transit differentiates several parallel simulation algorithms. Referring to the state diagram Fig 3-4, the synchronization cost in our algorithm is the best possible, as in our approach each processor only coordinates with the master processor. Table 4-5 shows the ratio of synchronization to total cost. The synchronization cost is affected by two factors: the underlying hardware and the distribution of events among the processors. The cost associated with the physical transfer of message is almost a constant. But from the definition of synchronization cost, it refers to the total time spent by each processor at the end of each synchronization phase waiting for the other processors to synchronize. Depending on the event distribution, some processors incur more synchronization cost than others do. The costs shown in Table 4-5 are the synchronization costs recorded at the master processor. It refers to the time instant when the first slave processor reaches the end of the time window and the master processor decides that all the slave processors have reached the end of the time window.

4.4 Implementation Issues

In this section we discuss some of the implementation details. This includes a brief discussion of the libraries and application uses for the implementation of the algorithms and trace collection.

4.4.1 Message Passing Interface (MPI)

All the algorithms were implemented for running on the Anantham cluster using the Message Passing Interface (MPI) libraries for communication among the processors. MPI is a message-passing application programmer interface, which includes point-to-point message passing and collective operations. MPI provides support for both the SPMD and MPMD modes of parallel programming. MPI runs over the GM (General Messaging) layer, which interfaces MPI to the underlying Myrinet interface. Some simulations on the cluster run a modified version of the GM drivers to improve throughput when messages sent are short. Simulation runs are made by way of submitting a job to the job scheduler, which retrieves jobs from the queue, executes them and posts the results to an output file. MPI is a standard for writing portable message passing C code. The cluster runs MPICH - a freely available implementation of MPI. MPI provides many features intended to improve performance on scalable parallel computers with specialized inter processor communication hardware. We give a brief description of the some of the inter-processor communication mechanisms in MPI.

a. ***Non-Blocking Communication***: Performance in parallel systems can be improved by overlapping communication with computation. As event-handling cost is close to zero, the only costs that show up in our performance numbers is from communication cost. In non-blocking communication, *post-send* initiates a send operation, but does not complete it. The *post-send* will return before the message is copied out of the send buffer. A separate *complete-send* is needed to complete the communication that is to verify that the data is copied out of the send buffer. With suitable hardware, the transfer of data can occur concurrently with computations at the sender processor after the send was initiated and before the operations was completed. Similarly a non-blocking *post-receive* initiates a post receive operation, but does not complete it. The call will return before a message is stored in the receive buffer. A separate *complete-receive* is needed to complete the receive operation and verify that data has been received into the receive buffer. In our implementation of non-blocking communication, each processor maintains its own pool of receive buffers and it initiates a pool of *post-receive* initiations with every logical channel it is connected i.e. for every other processor in the system from which it receives a message. When messages are sent, a non blocking send is initiated and as receive operation are already posted at the receiving end always, the message is expected to be in the receive buffer in the order of microseconds. Only rollback messages are checked for immediate completion at the sending processor.
b. ***Buffer management***: Each processor maintains its own set of buffers for send and receive operations. Each processor maintains a pool of free send and receive buffers allocated at initialization phase. During the simulation run it only allocates new buffers on demand. This avoids the LPs from trashing between calls for memory allocation and garbage collection.
c. ***Barrier Synchronization***: The processors in a communication group can be synchronized by use of MPI_Barrier. This blocks the caller processor until all the group members have called it. The call returns at any processor only after all the group members have entered the call.
d. ***Rendezvous***: For sending larger messages MPI uses the rendezvous semantics. The message is split automatically and the transfer starts after both the sender and

receiver have agreed for the message transfer. The message envelope is sent initially and after this the sender sends the full message.

e. ***Probing***: MPI_Probe allows the process to probe for a particular message type, from a particular processor etc. This probe could be done in a blocking manner, where the processor blocks for such a message or returns immediately with the status indicating the receipt of such a message.

Apart from these MPI provides other basic calls required for implementing communication among processors – grouping, rank, split, join etc.

4.4.2 Utilities used for Trace Generation:

For collecting the Web trace, we used two tools, Wget and Tcpdump. Wget is a freely available network utility that can be used to retrieve files from the web using HyperText Transfer Protocol (HTTP) and File Transfer Protocol (FTP). It works non-interactively thereby enabling it work in the background. This is used recursively to make mirror sites. When all the nodes were running web applications file transfer, web browsing and remote login, automated by the scripts, TCPDUMP was run on all the processors. TCPDUMP is a utility that provides a system independent interface for user level packet capture. This uses *Libpcap* for low-level network monitoring. Using tcpdump, network statistic collection, security monitoring, network debugging and trace collection can be carried out. We ran the tcpdump with option to read only the link layer header, i.e. only the first few bytes of the packet. Each packet is time stamped using the time stamp option of tcpdump. All the nodes of the cluster are time synchronized by the network timing protocol (NTP).

4.4.3 Messages in Transit

The logical channels between each LP needs be flushed at the end of each cycle. This is to ensure that there are no messages in transit, which could arrive after the system has calculated the time window for the next phase and GST set to the current end of the window. As all state is cleaned at the end of the window, messages in transit cause fatal rollbacks. The efficiency of detecting the messages in transit and avoiding fatal rollbacks is important implementation issue. Algorithms like token passing [Niranjan] could be used. But this requires an explicit message from each processor to every other processor in the system, requiring $O(n^2)$ messages. In our implementation, each LP maintains a count of the messages transmitted and received on each of its logical channels. At the end of its current time window, along with synchronization tag the LP sends its channel statistics to the master processor. The master processor checks for both ends of each logical channel by matching the transmit statistics with the receive statistics. The master processor decides the end of current cycle only when it finds that there are no messages in transit.

4.5 Chapter Summary

In this chapter we try to justify our choice of the parallel algorithm for the simulation layer of ONE. We discuss the difficulties in generating traces required for analyzing an algorithm for an emulation environment. We describe the traces that are used for our analysis. We introduce the oracle computations – heuristics for the NP-hard problem of globally optimal window sizes. Later in the section we compare the performance of the adaptive time window algorithm with the oracle computations. We also present additional performance metrics such as synchronization and lost opportunity costs. We discussed some of the implementation issues and the environment used to run our simulations.

Chapter 5

5 Conclusions

In this thesis, we presented the design, implementation and evaluation of a new parallel simulation algorithm for the ONE – The Adaptive Time Window Algorithm. The algorithm is optimistic, with the ability to rollback to a previous safe state and adaptive to the changes in degree of causality in the system. The algorithm was implemented and its performance evaluated for emulated traffic traces. This research also introduces the concept of an oracle that captures the upper bound on performance of any parallel simulation algorithm. The oracle uses an offline algorithm to determine a series of locally optimal time window sizes given an affordable number of rollbacks. We present a comparison of our adaptive time window algorithm against the oracle computation. Our evaluation shows that the performance of the adaptive time window algorithm is comparable to the upper bound derived by the oracle.

5.1 Future work

The current work provides all necessary constructs and framework to aid further work in extending the adaptive time window algorithm. Future improvements and modifications include the following:

- The effect of varying the damping factor (α) based on the deviation of the collected statistic (number of rollbacks per unit time window) was not studied. This is similar to the constant β based on of the constant α in TCP-Congestion control. There is no study done based on the deviation this idea. This could result in a better estimate for the time window for the next cycle of the simulation.
- The parameters discussed in Section 4.3.3 should be adaptive. Rather than using constants for the Unit time window and maximum window, these parameters could be studied for adaptation. The effect of varying values for these parameters with the event density and causality in the system needs to be studied.
- The constant cost of rollback independent on the length of the rollback needs to be analyzed with more specific implementation information. The current work was based on the assumption of a near constant cost associated with state restoration irrespective of the rollback length. This would depend on the performance evaluation of the 'Weaves and Tapestries'.
- The new adaptive time window algorithm hasn't been analyzed for other simulation domains. From the results and basics provided by the current system, the future appears promising. If the parameters of the algorithm are tuned close enough to capture the characteristics in the input traffic that arise in simulated environments, the adaptive time window algorithm could be a proper choice for that simulation domain.
- All through out the performance evaluation, we evaluated with a zero event handling cost. This provides the worst-case behavior analysis. The event handling cost is dependent on the implementation issues of the upper layers of ONE. A random

distribution of event handling times is not a realistic approach. The algorithm needs to be analyzed for non-zero event handling times.

- The fault tolerant system design discussed in Section 3.4 needs to be implemented.

5.2 Conclusions

This current work establishes the need for an algorithm that strikes a balance between an optimistic and a conservative approach for parallel simulation. With no bounds on optimism the amount of state to be saved is unbounded. So a parallel simulation engine aiming at good performance for a system like the ONE needs a bounded optimistic algorithm. Due to our inability to predict the degree of optimism that would result in optimal performance, we designed an adaptive algorithm that reacts to changing degree of causality in the system. This algorithm was designed for ONE and with the constraints of the target system in mind. A simulation engine for ONE based on the adaptive time window algorithm would make it possible for ONE to be the only distributed environment supporting Direct Code Execution.

6 References

[Carson] M. Carson, “Application and Protocol Testing through Network Emulation”. http://www.antd.nist.gov/itg/nistnet/slides/index.html (Current Dec. 2000)

[Chandy] K.M. Chandy and J. Misra, “Asynchronous Distributed Simulation via a Sequence of Parallel Computations”, CACM., vol. 24, no. 11, Apr. 1981, pp.198-206.

[Chu] Chu-Cheow Lim, Yoke-Hean Low et al: “An Empirical Comparison of Runtime Systems for Conservative Parallel Simulation”, 2nd Workshop on Runtime Systems for Parallel Programming, Orlando, Florida USA, March, 1998.

[Cilk-90] W. Cai et al., “An Empirical Comparison of Runtime Systems for Conservative Parallel Simulation”, 2nd Workshop on Runtime Systems for Parallel Programming, Orlando, Florida, USA, Mar. 1990.

[Cilk-97] W. Cai, E. Letertre, S.J. Turner, “Dag Consistent Parallel Simulation: a Predictable and Robust Conservative Algorithm”, Proc. 11th Workshop on PADS, 97, pp. 178-181.

[Cleary] J.G. Cleary and J.J. Tsai, “Conservative Parallel Simulation of ATM Networks”. Research Report, Dept. of Computer Science, University of Waikato, Oct. 1995.

[Damani] O.P. Damani, Y.M. Wang and V.K. Garg, “Optimistic Distributed Simulation Based on transitive Dependency Tracking”, Proc.11th Workshop on PADS, 1997, pp. 90-97.

[Danzig] J.S. P.B. Danzig et al. “Evaluation of TCP Vegas: Emulation and Experiment”, ACM SIGCOMM, Cambridge, MA, USA, 95, pp. 185-195.

[Ernst] T. Ernst, “Notes about network simulators”. http://www.inrialpes.fr/planete/people/ernst/Documents/simulator.html (Current Jan 2000)

[Estrin] D. Estrin et al. “Improving Simulation for Network Research”, tech. report 99-702, Computer Science Department, Univ. of Southern California, 1999.

[Ferscha-94] A. Ferscha and G.Chiola, “Self-Adaptive Logical Processes: the Probabilistic Distributed Simulation Protocol”, Proceedings of the 27th Annual Simulation Symposium, 1994, pp. 78-88.

[Ferscha-95] A. Ferscha and J. Luthi, “Estimating Rollback Overhead for Optimism Control in Time Warp”, Proc. of the 28th Annual Simulation Symposium, 1995.

[Fujimoto] R.M. Fujimoto, “Parallel Discrete Event Simulation”, CACM, vol. 33, no. 10, Oct. 1990, pp 30-53.

[Jain] Raj Jain, The Art of Computer Systems Performance Analysis - Techniques for Experimental Design, Measurement, Simulation and Modeling, John Wiley & Sons, Inc.

[Jefferson] D.R.Jefferson, “Virtual Time”, ACM Transactions on Programming Languages and Systems, 7(3), 1985, pp. 404-425.

[Kalyan] K. Perumalla, “Techniques for efficient Parallel Simulation and their Application to Large-scale Telecommunication Network Models”, Georgia Institute of Technology, PhD Dissertation.

[Keshav] X.W.Huang, R.Sharma and S.Keshav, “The ENTRAPID Protocol Development Environment”, Proc. IEEE INFOCOM, Mar. 1999.

[Kevin] K. Fall, “Network Emulation with the NS Simulator”. http://www.isi.edu/nsnam/ns/ns-emulation.html (Current Dec. 2000)

[Lubachevsky] B. Lubachevsky, A. Shwartz and A. Weiss, “An Analysis of Rollback-Based Simulation”, tech. report TR 91-37, Institute for Systems Research, Univ. of Mayland, College Park, MD.

[Madisetti] V.K. Madisetti, J. Walrand and D. Messerschmitt, “WOLF: A rollback algorithm for optimistic distributed simulation systems”, Proc. Winter Simulation Conferences, Dec. 1988, pp. 296-305.

[Masliah] M.R. Masliah, Using the Exponential Smoothing Approach to Time Series Forecasting on 6 DOF Tracking Data.
http://gypsy.rose.utoronto.ca/people/moman/Smoothing/smoothing.html (Current Jan. 2000)

[Nicol-88] D.M. Nicol, “Parallel Discrete-Event Simulation Of FCFS Stochastic Queuing Networks”, Proc. ACM/SIGPLAN PPEALS, 1988, pp. 124-137.

[Nicol-96] D. Nicol and P. Heidelberger, “Parallel Execution for Serial Simulators”, ACM TOMACS, vol. 6, no. 3, July 1996, pp. 210-242.

[Niranjan] M.Singhal, N. Shivratri, Advanced Concepts in Operating Systems, McGraw Hill Publishers.

[Odlyzko] Andrew Odlyzko, “Internet Growth: Myth and Reality, Use and Abuse”, Information Impacts, November 2000.

[Opnet] “OPNET Module: Parallel Simulation”. http://www.opnet.com/products/modules/ps.html (Current Jan. 2001)

[Paxson] V. Paxson and S. Floyd, “Why We Dont Know How To Simulate The Internet”, Proc. Winter Simulation Conference, Atlanta, GA, 1997.

[Peterson-88] N.C. Hutchinson and L.L. Peterson, “Design of the x-kernel”, Proc. ACM SIGCOMM, Aug. 88, pp. 65-75.

[Peterson-96] L.S. Brakmo and L.L. Peterson, “Experiences with Network Simulation”, Proc. ACM SIGMETRICS, May 96, pp. 80-90.

[Pham-98] C.D. Pham, H. Brunst, S. Fdida, “Conservative simulation of Load - Balanced Routing in a Large ATM Network Model”, Proc. 12th workshop on PADS, May 1998, pp. 142-149.

[Pham-99] C.D. Pham, “High Performance Clusters: A Promising Environment for Parallel Discrete Event Simulation”, Proc. PDPTA 1999, vol 3, June 1999, pp 1299-1304.

[Prakash] A. Prakash and R. Subramanian, “Filter: An algorithm for reducing cascaded rollbacks in optimistic distributed simulations”, Proc. 24th Annual Simulation Symposium, Apr. 1991, pp.123-132.

[Rajaei] H. Rajaei, R. Ayani, and L.-E. Thorelli, “The local time warp approach to parallel simulation”, Proc. Workshop on PADS, 1993, pp. 119-126.

[Rego] E. Mascarenhas, F. Knop, V. Rego, “Minimum cost adaptive synchronization: experiments with the PARASOL system”. http://citeseer.nj.nec.com/73341.html (Current Apr. 2001)

[Reynolds-88] P.F. Reynolds Jr., “A Spectrum of Options for Parallel Simulation”, Proc. Winter Simulation Conference, 1988, pp. 325-332.

[Reynolds-98] S. Srinivasan and P.F. Reynolds, “Elastic Time”, ACM TOMACS, vol. 8, no. 2, Apr. 1998, pp. 103-139.

[Riley] G. Riley, “PDNS Parallel/Distributed NS”.
http://www.cc.gatech.edu/computing/compass/pdns/index.html (Current Apr. 2000)

[Rizzo] L. Rizzo, “Dummynet: a simple approach to the evaluation of network protocols”, ACM Computer Comm. Review, Vol. 27, no. 1, Jan. 97, pp. 31-41.

[ScramNet] L. Yujih, “Distributed Simulation Of A Large-Scale Radio Network”, OPNET Technologies Contributed Papers. http://www.opnet.com/products/modeler/biblio.html (Current Apr. 2001)

[Sokol] Lisa M.Sokol, Duek P.Briscoe, and Alexis P.Wieland, “MTW: A strategy for scheduling discrete simulation events for concurrent execution”, Proc. SCS Multiconference on Distributed Simulation, 19(3), 1988, 34-42.

[Steinman] J.S. Steinman, “Discrete event simulation and the event horizon part 2: Event list management”, Proc. 10th Workshop on PADS, 1996, pp. 170-178.

[Torrent] Ericsson IP Infrastructure (formerly Torrent Networks), “Virtual Net - Virtual Host Environment”, Personal Communication, 2000.

[Tripathi] D. Hamnes and A. Tripathi, “Investigations in Adaptive Distributed Simulation”, 8th Workshop on PADS, 1994, pp. 20-23.

[Turner] S.J. Turner and M.Q. Xu, “Performance Evaluation of the Bounded Time Warp Algorithm”, Proc. SCS Parallel and Distributed Simulation Conference, Jan. 1992, Newport Beach, CA, USA, pp 117-126.

[VINT] “Virtual InterNetwork Testbed (VINT): methods and system”, Project proposal, Information Sciences Institute, University of Southern California 96-ISI-05.

[Varadarajan1] S. Varadarajan and J. Mukherji, “Weaves and Tapestries A compiler driven framework”, May 2001. (Work in progress)

[Varadarajan2] S. Varadarajan and C. Knestrick, “User Level Network Stacks for Network Emulation”, May 2001. (Work in progress)

[Wilsey] A.C. Palaniswamy and P.A. Wilsey, “An Analytical Comparison of Periodic Checkpointing and Incremental State Saving”, Proc. Workshop on PADS, 1993, pp. 127-134.

[YAWNS] “The YAWNS Protocol, Parallel JteD: The YAWNS Protocol”. http://www.cs.dartmouth.edu/~nicol/ S3/yawns.html (Current Apr. 2001)

Vita

Surya Ravikiran Kodukula hails from Visakhapatnam, India. He completed his Bachelors Degree in Computer Science and Engineering from Andhra University, India. Later he joined the Graduate Program in Computer Science at Virginia Tech. He currently works with the Comsat Laboratories division of Viasat Inc., in Maryland.