Master's Thesis, Dept. of Electrical and Computer Engineering, University of Arizona, 1993.
Also PMRL Technical Report 93-11, Dept. of Electrical and Computer Engineering, Univ. of Arizona, May 1993.

PERFORMANCE EVALUATION OF N-PROCESSOR TIME WARP USING STOCHASTIC ACTIVITY NETWORKS

by

Bruce Daniel McLeod

A Thesis Submitted to the Faculty of the

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

In Partial Fulfillment of the Requirements
For the Degree of

MASTER OF SCIENCE
WITH A MAJOR IN ELECTRICAL ENGINEERING

In the Graduate College

THE UNIVERSITY OF ARIZONA

1 9 9 3

STATEMENT BY AUTHOR

APPROVAL BY THESIS DIRECTOR

This thesis has been approved on the date shown below:

__________________________ ________________

William H. Sanders Date

Assistant Professor of

Electrical and Computer Engineering

ACKNOWLEDGMENTS

There are numerous people I would like to thank for their help and support in this endeavor. I would like to thank my graduate committee members: Dr. Pamela Nielsen, Dr. Bernard Zeigler, and my advisor, Dr. William Sanders. I would like to further thank Pam for her assistance with the probability theory, and Bill for his continuous guidance and support since day one.

I would like to thank the whole PMRL crew for their support, especially Luai Malhis, who kept me from taking myself too seriously. I would also like to thank Abbie Warrick for answering my numerous math questions, even when under the facade of a friendly visit.

To the memory of my grandfather, Paul Cioffi,

who instilled in me a passion for science,

to my father, Bruce A. McLeod,

who taught me professionalism by example,

and to Al Bargoot,

who taught me the value of original thinking.

TABLE OF CONTENTS

LIST OF FIGURES 7

LIST OF TABLES 8

ABSTRACT 9

1. INTRODUCTION 10

2. THE TIME WARP PROTOCOL MODEL 14
- 2.1. Time Warp Protocol Review 14
- 2.2. Model Assumptions 17
- 2.3. Model description 19
  - 2.3.1. Virtual time representation 19
  - 2.3.2. Forward progress 23
  - 2.3.3. Rollback 25
  - 2.3.4. GVT advancement 27
  - 2.3.5. Model summary 31

3. SAN MODEL FOR TIME WARP 32
- 3.1. Forward progress 35
- 3.2. Rollback 37
- 3.3. GVT advancement 41
  - 3.3.1. Performance variables and solution 44

4. RESULTS 46
- 4.1. Results for Model I 47
- 4.2. Results for Model II 51
- 4.3. Results for Model III 52

5. CONCLUSIONS 55

Appendix A. PROJECT DOCUMENTATION 57
- A.1. Activity definitions 57
- A.2. Input gate definitions 61
- A.3. Output gate definitions 68

A.4. Performance variable definitions 72

REFERENCES 74

LIST OF FIGURES

2.1. Space-time representation of a simulation. 15
2.2. Residual time distribution. 20
2.3. Density function comparison with $\lambda = 1$. 23
2.4. Comparing virtual times. 24
2.5. Determining a new regulator. 27

3.1. Time Warp composed model. 32
3.2. *proc* SAN. 33
3.3. Definitions of *do_send* and *no_send* (without starred command). 36
3.4. Definition of *get_info* function. 38
3.5. First case probability of *how_many*. 40
3.6. Definition of *no_more*. 41

4.1. Speedup vs. N given different values of MAX. 47
4.2. Fraction of time blocked vs. N given different values of MAX. 48
4.3. Speedup vs. MAX given different sending probabilities, $N = 3$. 49
4.4. Speedup vs. N given different values of communication cost, $MAX = 3$. . 50
4.5. Channel utilization given different values of N and communication cost, $MAX = 3$. 50
4.6. Speedup vs. MAX given different values of β, the fixed rate rollback cost, $N = 3$. 51
4.7. Fraction of time redoing events, blocked, and rolling back for Model II, $N = 3$. 52
4.8. Speedup vs. MAX given different values of α, the rollback cost added for each rolled back event, $N = 3$. 53
4.9. Fraction of time redoing events, blocked, and rolling back for Model III, $N = 3$. 54

LIST OF TABLES

2.1. Values of $P_{ww}[i, j]$. 26

3.1. Definition of *limit*. 35

4.1. State Space Sizes for Model I, Model II, and Model III. 46

A.1. Activity Time Distributions 58
A.2. Coding of place *rolling* 58
A.3. Case Probabilities for Activities 58
A.4. Case Probabilities for Activities (continued) 59
A.5. Case Probabilities for Activities (continued) 60
A.6. Case Probabilities for Activities (continued) 62
A.7. Case Probabilities for Activities (continued) 63
A.8. Encoding of place *action* 64
A.9. Input Gate Predicates and Functions 64
A.10.Input Gate Predicates and Functions (continued) 65
A.11.Input Gate Predicates and Functions (continued) 66
A.12.Encoding of place *channel* 67
A.13.Output Gate Functions 69
A.14.Output Gate Functions (continued) 70
A.15.Output Gate Functions (continued) 71
A.16.Performance variables 72

ABSTRACT

The speedup obtainable with Time Warp parallel discrete-event simulation varies greatly with the characteristics of the simulation and the Time Warp implementation. Analytic studies have been done to determine the expected speedup of Time Warp, but these studies have been limited to bound analysis or analysis considering a few overheads with the others assumed negligible. The models used in these studies have often been constructed directly in terms of a Markov process, making the construction process difficult. This thesis uses stochastic activity networks to construct a model of Time Warp at a higher level, enabling the construction of a more detailed underlying Markov process. The result is an analytic model for N-processor Time Warp that considers the combined effects of limited optimism, cascaded rollback, communication cost, and rollback cost. Given these effects, solutions for performance variables such as speedup, fraction of time blocked, and channel utilization are obtained.

CHAPTER 1

INTRODUCTION

Performance modeling, simulation, and analysis are critical parts of the design process of today's complex systems. Analytical modeling offers the advantage of an exact answer, but solution techniques become exceedingly complicated for complex systems. Simulation is becoming increasingly popular for modeling such systems, but simulation too has problems with scalability. Simulations of telecommunication networks, VLSI circuits, and other complex systems can take weeks or even months.

Multiprocessing can be used to speed up simulation execution times. *Parallel discrete-event simulation* (PDES) [1] or *distributed simulation* involves splitting up a single simulation onto two or more processors. Discrete-event simulation involves the scheduling and execution of a sequence of events in time order. The ordering of events must be preserved to ensure correctness, making the parallelization of the algorithm difficult. Asynchronous algorithms for distributed simulation allow each processor to maintain a local clock. This implies that synchronization must be used to preserve the correct ordering of events. Using *optimistic synchronization*, each process is allowed to progress in an unrestricted manner. If a causality error occurs, the speculative progress is undone, a process called *rollback*. The most popular of the optimistic protocols is known as Time Warp [2, 3].

Making a simulation N times faster with an N processor multiprocessor is an attractive idea, but the protocols used for synchronization introduce overhead that limits the achievable *speedup*. Speedup is defined as the ratio of execution time on a single processor to execution time on a multiprocessor. It would be useful to designers and users of simulation to predict the performance of Time Warp and other synchronization protocols. This thesis will analyze the performance of Time Warp simulation for a certain class of simulations.

Several analytical studies have been done to evaluate the performance of the Time Warp protocol. The first analytical studies were by Lavenberg, Muntz, and Samadi [4] and Mitra and Mitrani [5]. They both studied the two processor case with minimal attention to overhead. Felderman and Kleinrock have extensively studied the two processor case. In [6, 7], they analyze models with no overhead. In [8], they analyze a model which considers message queuing and another model that considers state-saving and rollback cost.

Several studies have looked at Time Warp performance given an arbitrary number (N) of processors. Many of these studies determine bounds on the performance of Time Warp. In particular, Felderman and Kleinrock [9] consider the improvement of asynchronous over synchronous operation for N processors. In [10], they derive bounds on N processor performance, based on the number of processes at GVT (Global Virtual Time). Lubachevsky and Weiss look at the dynamics of rollback in [11], including wildfire cascading and echo. They also propose a new method of rollback called “filtered rollback.” Lin and Lazowska use critical path analysis [12, 13] to determine bounds on N processor performance. They give conditions when Time Warp is conservative optimal, and when

Time Warp outperforms Chandy-Misra (a conservative approach). Eager, Zahorjan, and Lazowska [14] derive bounds based on the average parallelism of the simulation. Nicol [15] finds bounds on N processor performance with constant state-saving and rollback costs.

Other studies have attempted to obtain exact values rather than bounds on the performance of Time Warp. Dickens and Reynolds [16] analyze performance using a conservative windowing algorithm with N processors and no overhead. Madisetti, Walrand, and Messerschmitt [17, 18] analyze a model for the two processor case, and then look at the N processor case when using a new synchronization method called concurrent resynchronization, which resynchronizes whole clusters of processes at a time. Madisetti and Hardaker look at similar models in [19] with more emphasis placed on weakly-coupled systems. Gupta, Akyildiz, and Fujimoto [20] study N processor performance with no overhead. This work is extended to consider limited memory [21] and multiple processes per processor [22]. Lin and Lazowska have models to predict the state-saving overhead and optimal checkpoint interval in [23].

All of these studies have built Markov processes at the state level. Their analyses were therefore limited by the complexity of this state level representation. This thesis presents an analytic model for determining the performance of N-processor Time Warp using stochastic activity networks (SANs) [24], a stochastic extension to Petri nets. By defining the model at a higher level, we were able to build a more exact model, considering multiple sources of overhead in a single model. The model is a self-initiating model with N processes interacting probabilistically. By expressing the virtual time of each process as a number of events, rather than a number of clock ticks, we are able to consider continuously distributed simulation time. The density function of virtual time is first derived, along with a set of probabilities related to comparisons of virtual times. These probabilities are incorporated into a SAN model that considers the combined effects of limited optimism, cascaded rollback, communication cost, and rollback cost. *UltraSAN* [25], a SAN-based modeling package, is used to specify the model, generate the state space, and solve for selected performance variables.

The organization of this thesis is as follows. Chapter 2 will review the Time Warp protocol and discuss the model that was constructed. Chapter 3 will discuss SANs and the SAN representation of this model. Chapter 4 will give the results of our analysis, and Chapter 5 will give conclusions and directions for continued research.

CHAPTER 2

THE TIME WARP PROTOCOL MODEL

This chapter will first briefly review the workings of the Time Warp protocol, as described in [3]. This reference contains a more complete description. The assumptions that will be used in the model will then be given, followed by a detailed description of the model.

2.1 Time Warp Protocol Review

To use the Time Warp protocol, the physical system to be modeled must be divided into individual tasks. Each task is then executed as a *logical process* (LP). Each LP contains a partition of the state variables of the simulation. The LP's are then divided among the available processors. As in normal event-driven simulation, each logical process repeatedly advances its local clock to the time of the earliest scheduled event in its event list, and then processes the event. This may involve changing state variables, scheduling new events, and/or updating the simulation statistics. In the simple case, all needed state information is local, and any events to be scheduled are for the same LP. If interaction is needed with another LP, it is accomplished with the use of messages, which are passed

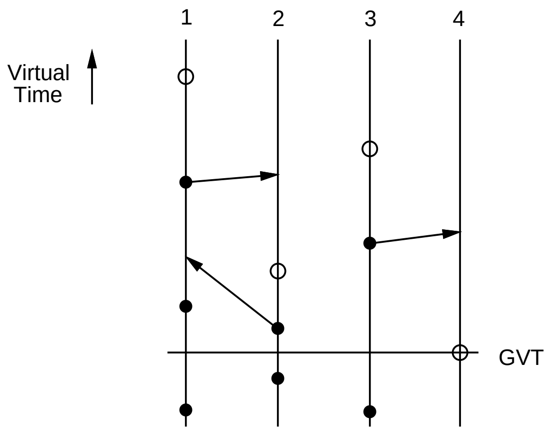

Figure 2.1: Space-time representation of a simulation.

between the processes. The communication delay, which is the real time between when a message is sent and received, varies for different architectures and implementations.

To ensure correctness of the simulation, two causality constraints must be obeyed:

- Events scheduled within each process must be executed in order.
- The time a message is received must be after the time the message is sent.

Note that the words “time” and “order” refer to simulation time, not real time. The second causality constraint is enforced by putting a *timestamp* on each message, which represents the simulation time at which the destination process should process the message. Figure 2.1 gives a space-time representation of the progress of a simulation, with each vertical line representing an LP. Positions along the lines represent points in simulation time or *virtual time*, as it is called in distributed simulation. The filled dots represent events that have been processed. The empty dots represent events that are currently being processed, and the arrows represent messages between processes.

Each process maintains its own virtual clock, whose time is called the *local virtual time* (LVT). The Time Warp protocol allows each process to continue processing the events in its event queue under the assumption that no message will arrive with a timestamp earlier than LVT. If such a message, called a *straggler*, does arrive, the receiving process must rollback, undoing the effects of the erroneous computation, and then continue forward again. A rollback involves restoring the state of the process to its state before the causality error occurred. In addition, anti-messages must be sent to processes that received messages that should not have been sent. Anti-messages cancel the effects of their positive counterpart, and may, in turn, cause other rollbacks. This effect is called *cascaded rollback*. For a process to be able to restore its state, it must periodically save it, a process called *state-saving* or *checkpointing*.

The point in time marked GVT in Figure 2.1 is the *global virtual time*. This time is the smallest virtual time of any process or outstanding message in the system. Process 4 is known as the *GVT regulator*, since it is the process with its LVT equal to GVT. Given continuously distributed virtual time, only one process or message can be the GVT regulator. Events that occurred before GVT cannot be rolled back, and therefore can be safely *committed*. *Fossil collection* is the process of reclaiming memory from saved states and messages with times before GVT. The frequency and method of GVT calculation vary among Time Warp implementations. If GVT is determined more often, events will be committed earlier, which frees up memory, but the calculation relies upon global knowledge, and therefore requires a polling algorithm to be executed. This type of algorithm takes up network bandwidth and processor time, and can therefore hurt performance.

Several different performance parameters are used to measure the effectiveness of the Time Warp protocol. The must common parameter is speedup, the ratio of execution time on a single processor to execution time on a multiprocessor, ideally equaling the number of processors being used. Speedup is limited by the overheads discussed, and also by the parallelism available in the simulation.

2.2 Model Assumptions

The goal of this thesis is to build an analytic model to estimate speedup, and other performance measures, taking in effect the limitations induced by the various overheads just discussed. In doing this we make the following assumptions:

1. The model is *self-initiating*. In a self-initiating model, each process continuously schedules its own events. Messages are for synchronization only and carry no work.[1] The sender therefore uses its own LVT as the timestamp of the message. The Ising spin model [26] is an example of a simulation for which this model would apply.

2. All delays in the model are exponentially distributed. The virtual time increments of each process are independent and identically distributed exponential random variables with rate λ. The time to process an event is exponentially distributed with rate γ. The time to rollback is also exponentially distributed. Its rate will be discussed later in this chapter. The communication channel is modeled as a queue,

[1]In a *message-initiating* model, messages represent events and therefore do carry work.

with the time to retrieve a message from the channel exponentially distributed with rate μ.

3. After processing an event, a message is sent with probability p. All processes have equal chance (including the sender) of receiving the message.

4. State-saving takes negligible time, and state is saved after every event. This implies that rollbacks can stop one event before the time of the straggler causing the rollback.

5. Each process is allowed to be, at most, MAX events ahead of the GVT regulator. This is a way of limiting the optimism of each process. A Time Warp variation known as *Moving Time Window* [27] limits the optimism of each process, but the limit is in terms of simulation time and not the number of events.

6. Aggressive cancellation and message preemption are used. With aggressive cancellation, anti-messages are sent immediately after a causality error is detected. With message preemption, incoming messages preempt the event being processed if rollback is necessary. This is not common among implementations but, according to Lin and Lazowska [13], is necessary for the correctness of the Time Warp algorithm, given that incorrect computation could cause an infinite loop.

7. GVT calculation is initiated every time the GVT regulator advances.

8. There is assumed to be one process per processor. Given this assumption, the words process and processor will be used interchangeably.

The above assumptions relate to both the characteristics of the executing simulation and the Time Warp implementation. As will be discussed in the next section, the model reflects many of the dynamics of the class of simulations and implementations that have been described. The model is not meant to predict the expected speedup of a particular Time Warp implementation, but rather to provide insight into the relationships of various parameters of the simulation and implementation, and how they relate to speedup. By considering performance across ranges of relative timings for the components of the system, an understanding of these relationships can be obtained.

2.3 Model description

As in simulation, there are two notions of time to be considered: *real time* and *virtual time*. The time it takes a processor to process a simulation event is real time. The timestamp increment associated with each new event is virtual time. Virtual time may go forward or backward, but real time always progresses. The model looks at the dynamics of virtual time as real time progresses. This section will first discuss the representation of virtual time, and then discuss the components of the model related to forward progress, rollback, and GVT advancement.

2.3.1 Virtual time representation

In order to study the dynamics of virtual time, we must find a convenient way of expressing the virtual time of each process. Different studies have done this in different ways. If the virtual time of each process is expressed as a number of clock ticks, comparison

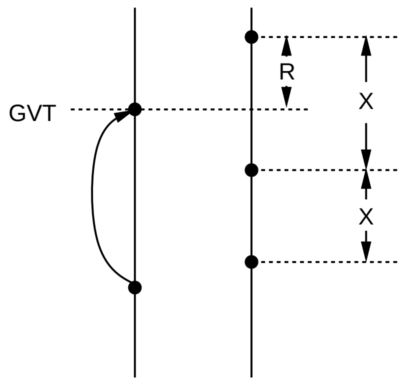

Figure 2.2: Residual time distribution.

of two virtual times is as simple as comparing their values, but this forces virtual time to be discrete in nature, even though simulation time is typically continuously distributed. We consider simulation time to be continuously distributed, with the timestamp increment (virtual time increment) of each event exponentially distributed. As in [20], we accomplish this by representing the virtual time of each process by the number of events, k, that have been processed, but not committed. This notation implies that virtual time is a random variable and therefore must be considered probabilistically.

In order to consider virtual time, we must first determine its density function as a function of k. Inter-event distances are exponentially distributed, so virtual time is mostly a sum of exponentially distributed random variables, but special consideration must be given to the first interval above GVT. Figure 2.2 shows that the time between GVT and the first event of each process, R, is not a full inter-event interval, and is therefore not exponentially distributed. The density function of this *residual time* must be found. Using this density function, the density function of virtual time can be derived.

Residual time density function

We must first derive the density function of the residual time. Bertsekas and Gallager [28] consider the residual time of an arriving customer to an M/G/1 queue with arrival rate equal to λ, and X_i equal to the service time of the ith customer. They show that as $i \to \infty$, the mean residual time equals $\frac{1}{2}\lambda\overline{X^2}$, which simplifies to $1/\lambda$ for exponential service times. Ross [29] considers the residual time in the context of alternating renewal processes. Given two renewal process and a jump of size t by the first process, he looks at the residual time until the next renewal of the second process at time t. He shows that as $t \to \infty$, the residual time converges in distribution to the inter-renewal distribution. Nicol [15] does a similar proof, considering a jump of a certain number of hops, j, by the first process. He shows that as $j \to \infty$, the residual time converges in distribution to the inter-renewal distribution. He uses the result to obtain bounds on the performance of Time Warp.

The advance of one process with respect to another is typically small, and therefore the limiting behavior is not an adequate measure. We must derive the exact density function of the residual time, which we denote R. The timestamp increment of each process, X, is exponentially distributed with rate λ (as per assumption 2 in Section 2.2), which has a density function given by:

$$f_X(t) \;=\; \lambda e^{-\lambda t}, \qquad t \geq 0.$$

If the increment of the non-regulator is known, say x, the residual time of the non-regulator relative to the new GVT is distributed as a truncated exponential with the time axis reversed, with density function:

$$f_{R|X}(t|x) = \frac{1}{K}\lambda e^{-\lambda(x-t)}, \qquad 0 \le t \le x.$$

K is a normalization constant, and is given by:

$$K = \int_0^x \lambda e^{-\lambda t} dt = 1 - e^{-\lambda x}, \qquad x \ge 0,$$

which gives:

$$f_{R|X}(t|x) = (1 - e^{-\lambda x})^{-1} \lambda e^{-\lambda(x-t)}, \qquad 0 \le t \le x.$$

The bivariate density of R and X is then:

$$\begin{aligned} f_{R,X}(t,x) &= f_{R|X}(t|x) f_X(x) \\ &= \lambda^2 (1 - e^{-\lambda x})^{-1} e^{-\lambda(2x-t)}, \qquad 0 \le t \le x. \end{aligned}$$

Integrating out x gives:

$$\begin{aligned} f_R(t) &= \int_{-\infty}^{\infty} f_{R,X}(t,x) dx \\ &= \lambda^2 e^{\lambda t} \int_t^{\infty} e^{-2\lambda x}(1 - e^{-\lambda x})^{-1} dx, \qquad t \ge 0. \end{aligned}$$

The difference between this density function and an exponential density function can be seen in Figure 2.3. The mean of R was found numerically with IMSL [30] to be approximately $0.645/\lambda$ as opposed to $1/\lambda$ for an exponential random variable.

Virtual time density function

The virtual time of an event k steps ahead of the regulator is the sum of $k-1$ exponentially distributed random variables and one residual random variable. We denote this random

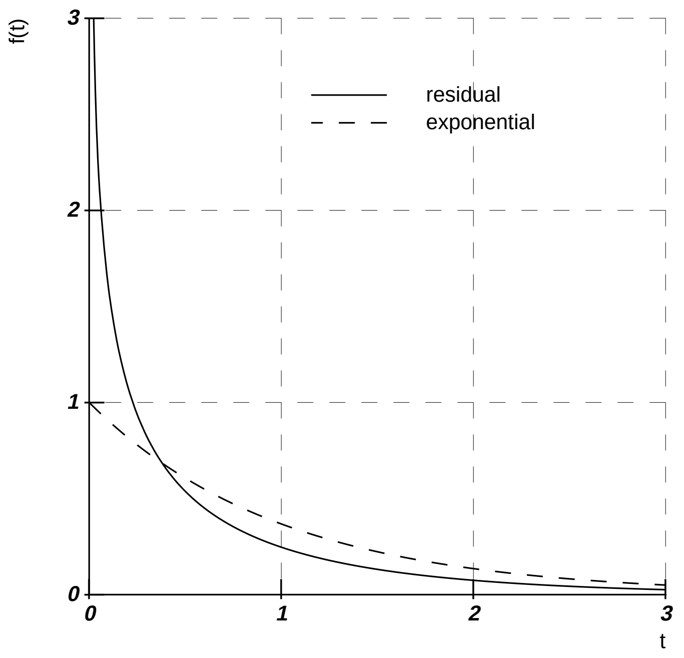

Figure 2.3: Density function comparison with $\lambda = 1$.

variable as W_k. Therefore, the density function is given by the convolution of an Erlang-$(k-1)$ density function $(f_{E_{k-1}}(t))$ and a residual time density function. For k equal to one, this is just $f_R(t)$. For $k \geq 2$:

$$\begin{aligned} f_{W_k}(t) &= \int_{-\infty}^{\infty} f_R(t-\tau) f_{E_{k-1}}(\tau) d\tau \\ &= \int_0^t \lambda^2 e^{\lambda(t-\tau)} \left(\int_{t-\tau}^{\infty} e^{-2\lambda x}(1-e^{-\lambda x})^{-1} dx \right) \frac{\lambda^{k-1}\tau^{k-2}}{(k-2)!} e^{-\lambda\tau} d\tau, \qquad t \geq 0 \\ &= \frac{\lambda^{k+1}}{(k-2)!} e^{\lambda t} \int_0^t \tau^{k-2} e^{-2\lambda\tau} \left(\int_{t-\tau}^{\infty} e^{-2\lambda x}(1-e^{-\lambda x})^{-1} dx \right) d\tau, \quad t \geq 0. \end{aligned}$$

2.3.2 Forward progress

Each of the N processes in the simulation progresses forward by continually processing events. As per assumption 2 of Section 2.2, the time to process one event is exponentially distributed with rate γ. Figure 2.4 shows an example of a three process simulation.

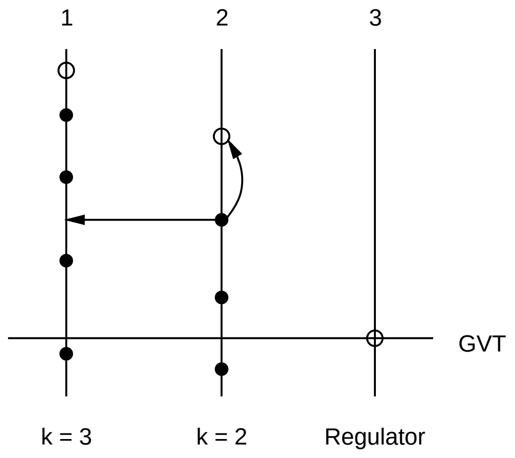

Figure 2.4: Comparing virtual times.

Process 3 is the GVT regulator. For process 1, k is equal to three. Process 2 has just finished processing an event and has incremented its value of k to two.

After processing each event, a process sends a *simulation message* with probability p. Simulation messages represent messages of the simulation, not messages related to the workings of the protocol. A message is sent by putting it onto a common channel. Each process (including the sender) then has an equal chance of receiving the message. Figure 2.4 shows process 1 receiving a message that was sent by process 2. Since this is a self-initiating model, the timestamp of the message is the virtual time of the sender, given as the sender's value of k before advancing. The message in Figure 2.4 would have a timestamp of two. This would cause a rollback, which is discussed in the next section.

Each process is limited from being more than MAX events ahead of the regulator. Once this limit is reached, a process stops processing. It cannot resume processing until it is rolled back or GVT advances. Section 4 will discuss whether or not this technique of limited optimism improves performance.

2.3.3 Rollback

When a process receives a message, it must determine probabilistically how many events must be rolled back, given the virtual time of the message. The rollback must then be performed.

Probability of rollback

To calculate how many events are rolled back, two virtual times must be compared probabilistically. The probability that a message with timestamp i rolls back an event with timestamp j is:

$$\begin{aligned} P_{ww}[i,j] &= P[W_i < W_j] \\ &= \int_{-\infty}^{\infty} f_{W_j}(t_2) \left(\int_{-\infty}^{t_2} f_{W_i}(t_1) dt_1 \right) dt_2. \end{aligned}$$

The ww subscript implies that two virtual time distributions are being considered.

Table 2.1 shows values for $P_{ww}[i,j]$ for several values of i and j. The bottom half is not filled in, but can be calculated given $P_{ww}[i,j] = 1 - P_{ww}[j,i]$. There is no closed form solution for these probabilities, so numerical integration was used (IMSL [30]). Each probability was calculated to an accuracy of three digits. In order to show the effect of the difference between W_k and an E_k random variable on P_{ww}, values are shown using both the correct residual distribution (res) and using an exponentially distributed random variable as the residual (exp). The latter case simplifies to a comparison of two Erlang distributions, which is given in [20] as:

$$P_{ee}[i,j] = 1 - \sum_{k=1}^{i} \begin{pmatrix} j+k-2 \\ k-1 \end{pmatrix} \left(\frac{1}{2}\right)^{j+k-1}.$$

Table 2.1: Values of $P_{ww}[i, j]$.

		j							
i		1	2	3	4	5	6	7	8
1	res	0.5000	0.7933	0.9050	0.9533	0.9755	0.9864	0.9918	0.9949
	exp	0.5000	0.7500	0.8750	0.9375	0.9688	0.9844	0.9922	0.9961
2	res		0.5000	0.7017	0.8269	0.9008	0.9434	0.9674	0.9812
	exp		0.5000	0.6875	0.8125	0.8906	0.9375	0.9648	0.9805
3	res			0.5000	0.6621	0.7810	0.8620	0.9145	0.9478
	exp			0.5000	0.6562	0.7734	0.8555	0.9102	0.9453
4	res				0.5000	0.6389	0.7503	0.8323	0.8901
	exp				0.5000	0.6367	0.7461	0.8281	0.8867
5	res					0.5000	0.6235	0.7280	0.8093
	exp					0.5000	0.6230	0.7256	0.8062
6	res						0.5000	0.6125	0.7112
	exp						0.5000	0.6128	0.7095
7	res							0.5000	0.6044
	exp							0.5000	0.6047
8	res								0.5000
	exp								0.5000

These values differ by as much as 5.4%. The effect of this difference on speedup will be shown in Chapter 4.

Cost of rollback

When a process is rolled back, it must restore its state and send anti-messages. For each event that is rolled back, an anti-message is sent with probability p, since that is the probability that a message was sent the first time the event was processed. This is done by putting a message on the channel with a timestamp equal to the value of k of the event that is being rolled back. Anti-messages are not distinguished from messages on the channel; they both have the effect of possibly rolling back the process that receives them.

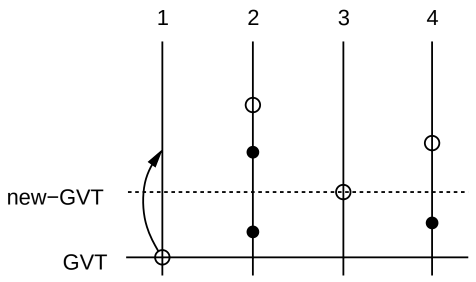

Figure 2.5: Determining a new regulator.

The cost of rollback is the time that the process spends processing the rollback. When the rollback is completed, the process can resume forward progress. The rollback time consists of the time to restore state and the time to process rolled back events. This model assumes that the time needed for a rollback is exponentially distributed with a mean of $\frac{1}{\beta} + \frac{num}{\alpha}$. This mean consists of two components: $1/\beta$ represents the average time to restore state, and $1/\alpha$ represents a time to be added for every event that is being rolled back (num), representing the average time to process a rolled back event. Therefore, the rollback cost has a constant component and a component proportional to the rollback distance.

2.3.4 GVT advancement

When the regulator advances, a new regulator must be chosen, and each process must commit those events with timestamps less than the new GVT. Figure 2.5 shows an example of this procedure. When process 1, which was the regulator, advanced, process 3 became the new regulator, since it was processing the earlist event in virtual time. Processes 2 and 4 each committed one event, since GVT advanced beyond one event at each process. In this model, when the regulator advances, choosing a regulator and the committing of events are done probabilistically.

Determining a new regulator

The state of all processes, other than the regulator, is expressed by a vector V, where V_k is the number of processes that are k events ahead of the GVT regulator. Rather than choosing a new regulator explicitly, the value of k of the new regulator is first chosen probabilistically, given V. The new regulator is then chosen uniformly from those processes that are k steps ahead of the old regulator.

Consider first the case where a new process becomes the regulator. Let $\mathcal{N}(|)$ be the number of processed, uncommitted events at a process j. The probability that a particular new process with k processed, uncommitted events will be the next regulator is the probability that the virtual time of that process is less than the virtual times of all other processes in the model, including E_1 for the old regulator's advance, and $W_{\mathcal{N}(\rangle)+\infty}$ for each other process i. The probability that some process with k processed, uncommitted events is chosen is the sum of these probabilities over all processes j such that $\mathcal{N}(|) = \|$. This probability is:

$$\begin{aligned} P_{Reg(k)} &= \sum_{j \ni \mathcal{N}(|)=\|} P\left[W_{k+1} < E_1 \; and \right. \\ &\qquad \left. W_{k+1} < W_{\mathcal{N}(\rangle)+\infty} \; \forall \; i \in [1, 2, \ldots, N], \; i \neq j, i \neq \mathit{regulator}\right]. \end{aligned}$$

Because the virtual times of each process with the same number of processed, uncommitted events are stochastically identical, the probability within the sum is the same for each value of j. This implies that the sum can be replaced by a factor of V_k. We will also change the order by which the virtual times will be compared; virtual times will be compared in ascending order of number of events ahead of the regulator. This gives:

$$\begin{aligned}P_{Reg(k)} &= V_k\ P\left[W_{k+1} < E_1\ and\right. \\ &\qquad \left.(W_{k+1} < W_{m+1}\ \forall\ i \in [1,2,\ldots,V'_m])\ \ \forall\ m \in [0,1,\ldots,MAX-1]\right].\end{aligned}$$

V' is equal to V with V_k decremented, which accounts for the $i \neq j$ condition given in the previous equation. The probabilities for the events in the previous equation can now be explicitly expressed in terms of the density functions calculated earlier. Additionally, the probability that $W_{k+1} < W_{m+1}$ is the same for each value of i, so the corresponding term can just be raised to the power of V'_m. This gives:

$$\begin{aligned}P_{Reg(k)} &= V_k \int_{-\infty}^{\infty} f_{W_{k+1}}(t_2) \left(\int_{t_2}^{\infty} f_{E_1}(t_1)dt_1\right) \prod_{m=0}^{MAX-1} \left[\left(\int_{t_2}^{\infty} f_{W_{m+1}}(t_1)dt_1\right)^{V'_m}\right] dt_2 \\ &= V_k \int_{0}^{\infty} e^{-\lambda t_2} f_{W_{k+1}}(t_2) \prod_{m=0}^{MAX-1} \left[\left(\int_{t_2}^{\infty} f_{W_{m+1}}(t_1)dt_1\right)^{V'_m}\right] dt_2.\end{aligned}$$

The probability that the old regulator is still the regulator is the probability that one exponentially distributed interval (the regulator's timestamp increment) is less than the virtual times of all other processes. This probability is:

$$\begin{aligned}P_{Reg(same)} &= P\left[E_1 < W_{\mathcal{N}()) + \infty}\ \forall\ i \in [1,2,\ldots,N], i \neq regulator\right] \\ &= \int_{-\infty}^{\infty} f_{E_1}(t_2) \prod_{m=0}^{MAX-1} \left[\left(\int_{t_2}^{\infty} f_{W_{m+1}}(t_1)dt_1\right)^{V_m}\right] dt_2 \\ &= \int_{0}^{\infty} \lambda e^{-\lambda t_2} \prod_{m=0}^{MAX-1} \left[\left(\int_{t_2}^{\infty} f_{W_{m+1}}(t_1)dt_1\right)^{V_m}\right] dt_2.\end{aligned}$$

Therefore, the regulator either determines that it is still the regulator, or it determines the value of k of the new regulator. Either way, in the model, a broadcast *GVT message* is sent with this information. Each process then uses this information to determine how many events should be committed. If there is a new regulator, a process with the correct value of k (selected uniformly) assumes the role of regulator.

Committing events

If a process receives a GVT message, it must probabilistically determine how many events should be committed. If the old regulator is still the regulator, then GVT has advanced by one exponentially distributed increment. Let us first determine the probability that one exponentially distributed increment (E_1) is less than W_j:

$$\begin{aligned}
P_{ew}[j] &= P[E_1 < W_j] \\
&= \int_{-\infty}^{\infty} f_{W_j}(t_2) \left(\int_{-\infty}^{t_2} f_{E_1}(t_1) dt_1 \right) dt_2 \\
&= \int_{-\infty}^{\infty} f_{W_j}(t_2) \left(\int_{0}^{t_2} \lambda e^{-\lambda t_1} dt_1 \right) dt_2 \\
&= 1 - \int_{0}^{\infty} e^{-\lambda t_2} f_{W_j}(t_2) dt_2 .
\end{aligned}$$

The *ew* subscript implies that an exponential distribution is being compared to a virtual time distribution. The probability that an event with timestamp j ($1 \le j \le k$) is committed, given that the old regulator is still the regulator, is $P[E_1 > W_j \mid E_1 < W_{k+1}]$, where k is the number of processed, uncommitted events of this process. It is known that $E_1 < W_{k+1}$, because this process was not chosen as the new regulator. This probability equates to $(P_{ew}[k+1] - P_{ew}[j])/P_{ew}[k+1]$.

If a new regulator is chosen and previously had i processed, uncommitted events, then GVT has advanced by W_{i+1}. The probability that an event with timestamp j

$(1 \leq j \leq k)$ is committed is: $P[W_{i+1} > W_j \mid W_{i+1} < W_{k+1}]$, where k is the number of processed, uncommitted events of this process. This equates to $(P_{ww}[i+1, k+1] - P_{ww}[i+1, j])/P_{ww}[i+1, k+1]$.

2.3.5 Model summary

The operation of the model can be summarized as follows. Each of the N processes repeatedly processes events, each time sending a message with probability p. A process is blocked if it is MAX events ahead of the regulator. If a simulation message arrives, a process determines how many events must be rolled back and sends anti-messages. It spends some time doing the rollback, and then resumes forward progress. If the GVT regulator advances, it first determines if it is to stay the regulator, or, if a new regulator exists, it determines the value of k of the new regulator. A GVT message is then sent to each process, so they may commit the appropriate number of events.

CHAPTER 3

SAN MODEL FOR TIME WARP

Rather than building a Markov model directly, the model presented in the previous chapter is constructed as a *stochastic activity network*. Stochastic activity networks (SANs) [24, 31] are a stochastic extension to Petri Nets that can be used to obtain analytical (as well as simulation) results concerning the performance of many systems. *Composed models* are constructed using a hierarchical structure of SANs. Figure 3.1represents the composed model of a Time Warp system. The *proc* node is a SAN, and the *Rep* node indicates that the *proc* SAN is to be replicated.

Figure 3.2 shows the *proc* SAN, which represents one processor of a simulation synchronized with the Time Warp protocol. SANs consist of five types of components: *timed*

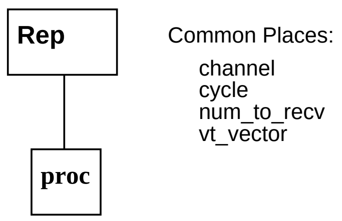

Figure 3.1: Time Warp composed model.

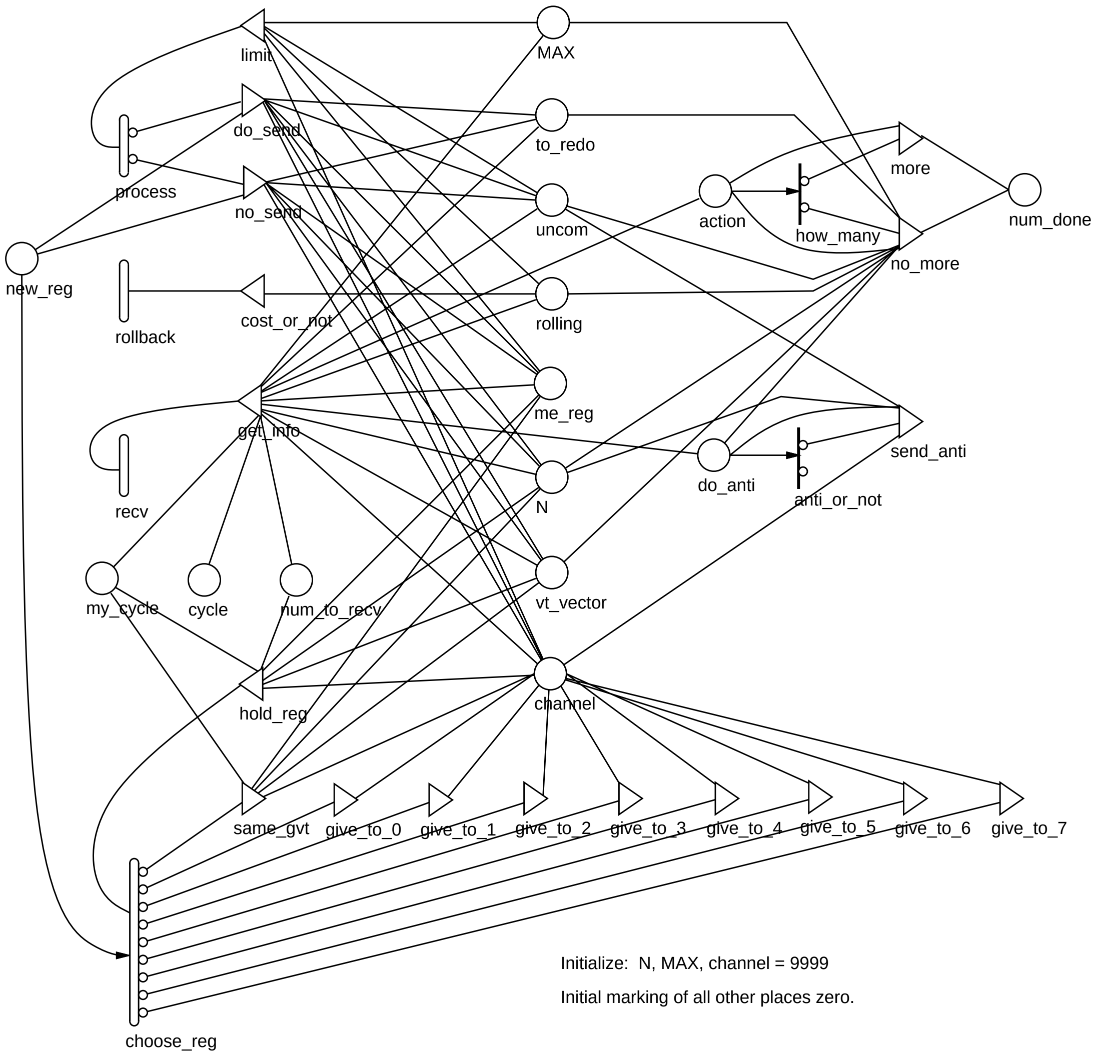

Figure 3.2: *proc* SAN.

activities (ovals), *instantaneous activities* (vertical lines), *places* (circles), *input gates* (triangles with their point connected to an activity), and *output gates* (triangles with their backside connected to an activity). The execution of SANs is discussed in detail in [24].

A composed model is used to replicate submodels or join them together with other submodels. Replication reduces the state space size of the resulting Markov process by exploiting symmetries in the model [31]. Replica submodels can interact with one another through the use of *common places.* The list of common places in Figure 3.1 represents the places that are common among all the replicas. As an example, the place *channel* represents the communication channel by which processors communicate. By making this place common, all replicas are sending and receiving on the same channel. All subsequent discussion related to the model will refer to the SAN in Figure 3.2.

Figure 3.2 and all other model figures were generated using *UltraSAN* [25], a stochastic activity network modeling package. The package allows the attributes of all the components to be defined using C code. The only extension to standard C is the *MARK* macro, which returns the marking of the specified place. For example, *MARK(channel)* returns the number of tokens in place *channel.*

This chapter will describe how the SAN model in Figure 3.2 implements the model discussed in the previous chapter, including forward progress, rollback, and GVT advancement. The performance variables will also be discussed.

Table 3.1: Definition of *limit*.

Gate	Enabling Predicate	Function
limit	$(MARK(me\_reg)\ ==\ 1\ \Vert$ $MARK(uncom)\ <\ MARK(MAX)\ -\ 1)$ $\&\&\ MARK(rolling)\ ==\ 0$ $\&\&\ MARK(channel)\ !=\ 9999$	identity

3.1 Forward progress

The activity *process* in the model represents the time it takes for a processor to execute a simulation event. This time is exponentially distributed with rate γ, and represents real time. There is no activity in the model to represent virtual time. Instead, as discussed in the previous chapter, the virtual time of each process is represented by its number of processed, uncommitted events, which is given as the number of tokens in place *uncom*. It should be noted that place *uncom* is not a common place, and therefore each copy of the replicated SAN maintains its own value of this variable.

The input gate *limit* gives the enabling condition for activity *process*. Its definition is given in table 3.1. The enabling predicate of the gate evaluates to one or zero, determining if the connected activity (*process*) is enabled or not. The marking of *me_reg* is one if this process is the GVT regulator, or zero if it is not. The marking of *uncom* represents the number of processed, uncommitted events of this process. The marking of *MAX* represents the maximum number of events that a process can be ahead of the regulator. The marking of *rolling* is greater than zero if the process is currently rolling back, and place *channel* represents the communication channel. Therefore, *process* continues to be enabled as long as the process is the regulator or has not exceeded the maximum

```
Decrement to_redo if > 0
if the channel is occupied
    Increment to_redo
else
    Put message on the channel        *
    if I am the regulator
        Set new_reg = 1
    else
        Increment uncom, adjust vt_vector
```

Figure 3.3: Definitions of *do_send* and *no_send* (without starred command).

allowable advance, the process is not rolling back, and 9,999 is not on the channel, which is an initialization condition. The function is specified as *identity*, which means the gate does not change the marking of the network when the activity completes. In this case, the marking change is handled by the output gates of *process* (i.e. *do_send* and *no_send*). For ease of explanation and conservation of space, all subsequent discussion will use pseudo-code rather than the C code of the model. The full code of the model can be found in Appendix A.

The case probabilities of activity *process* (the two small circles to the right of the activity) represent the probabilities of sending or not sending a message after processing an event (p and $1 - p$). They are fixed numbers. If the first case is chosen, output gate *do_send* is executed, else *no_send* is executed. Figure 3.3 shows the pseudo-code of the output gate *do_send*. Output gate *no_send* is identical except for the absence of the starred command. The place *vt_vector* is a coded vector of the number of processed, uncommitted events at each process. Its marking is:

$$\sum_{k=0}^{MAX-1} V_k(N+1)^k.$$

N is the number of processes in the model. $N+1$ acts as the base of the code. Each time a process modifies its own value of *uncom*, it adjusts *vt_vector*, by subtracting $(N+1)^{old\_k}$ and adding $(N+1)^{new\_k}$. It is a common place, so all processes know the status of all other processes in the model. The place *channel* is also coded, in order to allow different types of messages to be sent using the one place. Messages are given values in different ranges to represent different kinds of messages with different timestamps. This code also uses a function of the place N as its base.

The place *to_redo* represents the number of events that need to be redone, and will become important when considering the performance variables. Rather than queue up messages for the channel, if the channel is occupied when an event is completed, the event is redone. Finally, if this process is the GVT regulator, the place *new_reg* is set. The effect of setting *new_reg* will be discussed in the forthcoming section on GVT advancement.

3.2 Rollback

The timed activity *recv* represents the time to pull a message off the channel and has an activity time which is exponentially distributed with rate μ. The predicate of *get_info* is true (i.e. activity *recv* is enabled) when there is a simulation message or a GVT message on the channel. GVT messages will be discussed later in this chapter. The function of *get_info*, shown in Figure 3.4, is executed when *recv* completes. A marking of 9,999, the

```
if initialization code is on the channel
     Set $me\_reg$ = 1
     Clear the channel
     Initialize $vt\_vector$ to N - 1
else if simulation message on channel
     if message is from the regulator
         if I am not the regulator
             Set $rolling$
             $to\_redo$ = $to\_redo$ + $uncom$
             $do\_anti$ = $uncom$
             $uncom$ = 0, adjust $vt\_vector$
         else ignore it
         Clear the channel
     else if message is not from the regulator
         Remove the earliest message from the channel
         if I am not the regulator
             Put the message in $action$
         else ignore it
else if GVT message on channel
     if regulator is still available and $uncom$ = new value
         Set $me\_reg$ = 1
         Make regulator unavailable
         $uncom$ = 0, adjust $vt\_vector$
     else put the new value in $action$
     Toggle $my\_cycle$
     Decrement $num\_to\_recv$
     if $num\_to\_recv$ = 0
         Clear the channel
         Toggle $cycle$
```

Figure 3.4: Definition of *get_info* function.

initialization code, is put in place *channel* initially. The first process to receive this code becomes the first regulator and initializes *vt_vector*.

If a simulation message is received, one of several actions may be performed by the function of *get_info*. If the receiver is the regulator, the message is ignored. If the message is from the regulator, all progress is rolled back by setting *rolling* (as discussed in the next paragraph), adding the value of *uncom* to *to_redo*, setting *do_anti* to the value of *uncom*, and clearing *uncom*. The place *do_anti* represents a queue of anti-messages to be used to model cascaded rollback. Since the activity *anti_or_not* is instantaneous, they are sent immediately. The case probabilities of this activity are p and $1 - p$. Therefore, anti-messages are sent with the same probability as the original messages.

When place *rolling* has tokens in it, the process is rolling back. The predicate of *limit* indicates that the process cannot resume processing while the marking of *rolling* is greater than zero. The way that *rolling* is set gives rise to three different models. Model I sets rolling to zero, or equivalently, does not set it at all. This implies no rollback cost. Models II and III use the rollback cost given in Chapter 2, which was $1/\beta + num/\alpha$. The reciprocal is assigned as the rate of activity *rollback*. Model II assumes $\alpha = \infty$, and therefore the rollback cost equals $1/\beta$. Model III incorporates both α and β. The predicate of the input gate *cost_or_not* enables this activity if the marking of *rolling* is greater than zero. The function merely clears *rolling*.

If the incoming message is not from the regulator, the receiver must determine how many of its own events are rolled back. The message is put into place *action*, and immediately is processed by *how_many*, an instantaneous activity. The first case probability of

```
if $action$ contains a simulation message
    if $uncom$ = $num\_done$ - 1
        return 0.0
    else if $num\_done$ = 0
        return $P_{ww}[message, uncom + 1]$
    else
        return $\frac{P_{ww}[message, uncom - num\_done + 1]}{P_{ww}[message, uncom - num\_done + 2]}$
else if $action$ contains a GVT message
    if $uncom$ = $num\_done$
        return 0.0
    else if regulator advanced and still is regulator
        return $\frac{P_{ew}[uncom+1] - P_{ew}[num\_done+1]}{P_{ew}[uncom+1] - P_{ew}[num\_done]}$
    else
        return $\frac{P_{ww}[new\_value, uncom+1] - P_{ww}[new\_value, num\_done+1]}{P_{ww}[new\_value, uncom+1] - P_{ww}[new\_value, num\_done]}$
```

Figure 3.5: First case probability of *how_many*.

how_many is given in Figure 3.5. The parameter values of the return statements represent the probabilities to be used, given the different conditions, as derived in the previous chapter. The second case probability is identical to the first with each probability the complement of the first. The place *num_done* is a counter of the number of events that will be rolled back. It starts at zero for each new rollback.

The probability that the first event (the event with the highest timestamp) is rolled back is the probability that the timestamp of the message is less than the timestamp of the event being processed, or P_{ww}*[message, uncom + 1]*, where *message* is the timestamp of the received message, and *uncom* is the number of uncommitted events in the process. If the first case is chosen, the output gate *more* merely increments *num_done* and restores

```
Restore action
if num_done > 0
     if action contains a simulation message
         uncom = uncom - num_done + 1, adjust vt_vector
         Set rolling
         do_anti = num_done - 1
         to_redo = to_redo + num_done - 1
     if action contains a GVT message
         uncom = uncom - num_done, adjust vt_vector
Clear num_done, action
```

Figure 3.6: Definition of *no_more*.

the value of *action*. The probability that the second event is rolled back (the marking of *num_done* is now 1) is the probability that the timestamp of the message is less than the second event, given that the first event was rolled back, or P_{ww}*[message, uncom] /* P_{ww}*[message, uncom + 1]*. Activity *how_many* continues to complete, each time evaluating the probability that the message rolls back the next event. The output gate *no_more* is executed when the second case is chosen, which is either done probabilistically or when all events have been rolled back. The definition of *no_more* is given in Figure 3.6. The gate adjusts the value of *uncom* and processes the rolled back events as described earlier.

3.3 GVT advancement

The output gates *do_send* and *no_send* show that when the regulator advances, a token is put into place *new_reg*. A token in place *new_reg* indicates that a new regulator must be chosen. Activity *choose_reg*, via input gate *hold_reg*, contends for the channel, given its predicate of: *MARK(channel) == 0*. Once the channel is clear, a new regulator is chosen by activity *choose_reg*. Activity *choose_reg* is exponentially distributed with rate μ, which in this case represents the time to put a message on the channel. The first case probability of *choose_reg* is $P_{Reg(same)}$, which is the probability that this process is still the regulator. If this case is chosen, output gate *same_gvt* puts a GVT message on the channel, indicating that the regulator has advanced one event. The other cases have probabilities of $P_{Reg(k)}$, where k varies from zero to seven. These are the probabilities that the new regulator has k processed, uncommitted events. The vector needed to determine these probabilities is the value presently in place *vt_vector*. Each of these cases has an output gate associated with it named *give_to_k*, which puts a GVT message on the channel indicating that the regulator role is available and the value of k of the new regulator.

Output gates *do_send* and *no_send* perform two other actions in addition to those described; they set *num_to_recv* to $N-1$ and toggle the value of *my_cycle*. These actions are needed to facilitate the broadcast operation needed for GVT messages. The place *cycle* is a common place that has a value of one or zero. Place *num_to_recv* is also common, and indicates the number of processes that have yet to receive the message in place *channel*. The predicate of *get_info* says that a process should only retrieve a GVT message if *MARK(cycle) == MARK(my_cycle)*. After reading the message, it toggles its own value of *my_cycle*, preventing the process from reading the message again, before some other process reads it for the first time. After reading the message, the process also decrements *num_to_recv*. If the new value of *num_to_recv* is zero, the channel is cleared and *cycle* is toggled, making all processes eligible for receiving once again.

Figure 3.4, the function of input gate *get_info*, shows what a process does when it reads a GVT message off the channel. The process first checks to see if the flag is set indicating that the role of regulator is still available. If so, and the value of k of the regulator matches its own value of *uncom*, the process clears the flag and sets *me_reg* to one. If either of these conditions is not true, then the GVT message is put into place *action*.

Figures 3.5 and 3.6 show that GVT messages are handled in a manner similar to that of simulation messages. If the old regulator is still the regulator, then GVT advanced by one exponentially distributed increment. As discussed in Chapter 2, the probability that the first (bottom) event is committed is the probability that one exponentially distributed increment is greater than one residual time, given that the exponentially distributed increment is less than that process's virtual time. This equates to: $(P_{ew}[uncom + 1] - P_{ew}[1]) / P_{ew}[uncom + 1]$. The probability that the second event is committed (the marking of *num_done* is now 1) is the probability that the virtual time of the new regulator is greater than the second event, given that the first event was committed. This activity continues to complete, each time evaluating the probability that the next event is committed. When the second case is eventually chosen, *uncom* is decremented by the number of events that have been committed, as shown in Figure 3.6.

If there is a new regulator with $k = new\_value$, then the first (bottom) event is committed with probability $(P_{ww}[new\_value, uncom + 1] - P_{ww}[new\_value, 1]) / P_{ww}[new\_value, uncom + 1]$. As before, activity *how_many* continues to complete, each time evaluating the probability that the next event is committed.

3.3.1 Performance variables and solution

As can be appreciated from the previous section, the stochastic activity network model constructed is a fairly detailed representation of the behavior of a distributed simulation, and hence can support a rich set of performance variables. Performance variables for SANs are defined in terms of markings of the network and numbers of times particular activities complete. After a composed model and one or more performance variables are defined, *UltraSAN* generates a reduced base model (Markov reward model) [31], which represents the behavior of the composed model and supports the specified performance variables.

The mean, variance, probability density function, and probability distribution function can then be computed for each performance variable, by numerically solving the resulting Markov reward model. For the model in this thesis, we use an iterative method known as successive overrelaxation (SOR). When this solution method is used, the interpretation of the resulting numeric value is the instantaneous value the variable takes on in steady-state. Furthermore, if the model is in a single recurrent non-null class of states with probability one in steady-state, the value can be interpreted as the fraction of time the variable takes on various values. In the distributed simulation model, the initialization condition gives rise to a Markov process with a single transient state, and a single class of recurrent non-null states. Thus the "fraction-of-time" interpretation holds for this model.

The most important performance variable is speedup. To determine speedup, we define a performance variable that takes the value of the number of processes that have activity *process* enabled and have no tokens in place *to_redo*. This implies that events are being processed, and the current event is not being redone. Each replica of the *proc* SAN adds one to the random variable if these conditions are true, and zero if they are false. The value of speedup therefore varies between zero and N.

The fraction of time each process is blocked is another important performance variable. A process is blocked when it is MAX events ahead of the regulator. Since each replica is identical in steady-state, they will each have the same state-occupancy blocking probabilities, and hence fraction of time blocked. This allows us to normalize the random variable to the range of zero to one. In terms of the SAN model, each of the replicas adds a value of $1/N$ to the random variable when their own marking of place *uncom* equals $MAX - 1$, and adds zero otherwise.

Using a similar argument, the fraction of time each process spends rolling back is determined by having each replica add a value of $1/N$ when their marking of place *rolling* is greater than zero. This equates to the fraction of time that each process is rolling back. Furthermore, the fraction of time a process spends redoing events can be determined by having each replica add a value of $1/N$ when activity *process* is enabled, and there are tokens in place *to_redo*. Finally, the fractions of time that the channel contains simulation messages and GVT messages can be determined in a similar manner, by looking at place *channel*.

Each of these performance variables will be determined in the next section for particular numbers of processes and values of MAX.

CHAPTER 4

RESULTS

This chapter will present results that were obtained with the model. Theoretically, the model presented can be used to analyze the performance of Time Warp for any number of processors and any value of MAX, but the state space grows as each of these values increases. The analysis is therefore limited by the memory constraints of the computer used to do the evaluation. Table 4.1 shows the state space sizes of the resulting Markov models for the three models that were used. Only those combinations of N and MAX that were analyzed are given, although other combinations are possible. Model II, which adds the rollback activity with constant rate, and Model III, which adds the rollback activity with rate proportional to the number of events rolled back, each add more detail to the model, and therefore have larger state spaces.

Table 4.1: State Space Sizes for Model I, Model II, and Model III.

	N(I)						*N(II)*	*N(III)*
MAX	2	3	4	5	6	7	3	3
1	41	93	177	301	473	701	93	93
2	77	279	765	1,755	3,557	6,581	791	1,151
3	213	1,355	6,295	23,873	77,741	224,405	5,399	11,225
4	629	6,789	49,291	280,093			30,877	91,283
5	1,567	27,535	300,557				137,507	
6	3,601	102,405						
7	7,977	368,991						

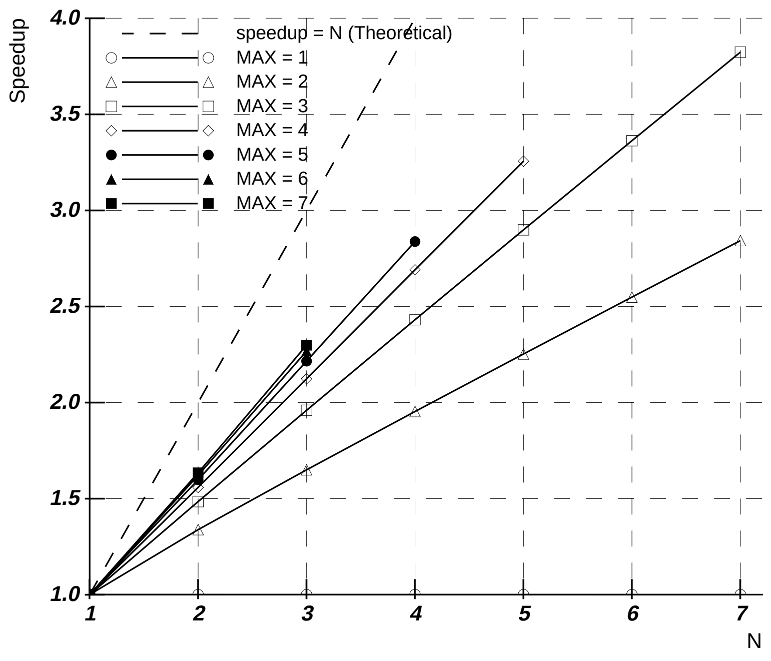

Figure 4.1: Speedup vs. N given different values of MAX.

4.1 Results for Model I

Figure 4.1 shows results obtained from Model I. Speedup vs. N is given for different values of MAX. The dashed line represents ideal performance (speedup $= N$). For this set of runs it was assumed that there is no rollback cost, p (probability of sending a message) equals 0.2, and γ (rate of processing events) equals 10. The communication cost, μ, is in the model, but the rate was set at 100,000 for this set of runs, so its impact is negligible. It can be seen that with no overhead, the speedup increases linearly with N, given a constant value of MAX. As MAX increases, the lines seem to approach a limiting slope. This slope is not equal to N because of the rollbacks initiated by simulation messages. Figure 4.2 shows the fraction of time each process is blocked for each value of MAX as

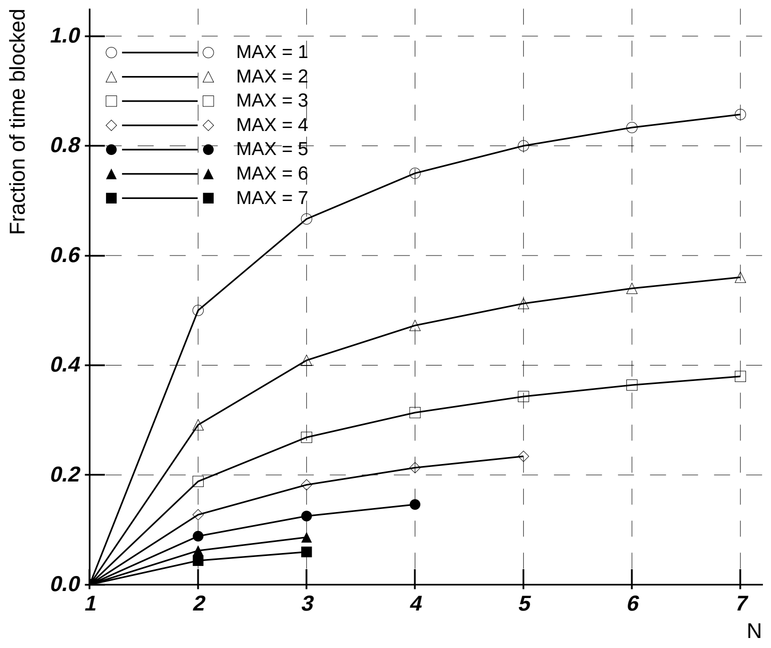

Figure 4.2: Fraction of time blocked vs. N given different values of MAX.

N is varied. With a MAX of one, each processor except the regulator is blocked, so the fraction of time blocked is $1 - 1/N$. As MAX increases, there is less and less chance of being blocked.

Figure 4.3 shows speedup vs. MAX for different values of p, when the number of processors is three. All other assumptions are the same as for Figure 4.1. As p increases, there are more rollbacks, and therefore speedup decreases. The top curve has p equal to zero. This implies no simulation messages, and therefore no rollback. In this case, progress is only limited by the assumed limited optimism. It is therefore believed that the top curve asymptotically approaches ideal performance (speedup = 3).

Figure 4.3 also shows the effect on speedup of the calculated residual distribution from Chapter 2. The solid lines use the correct distribution. The dotted lines use the

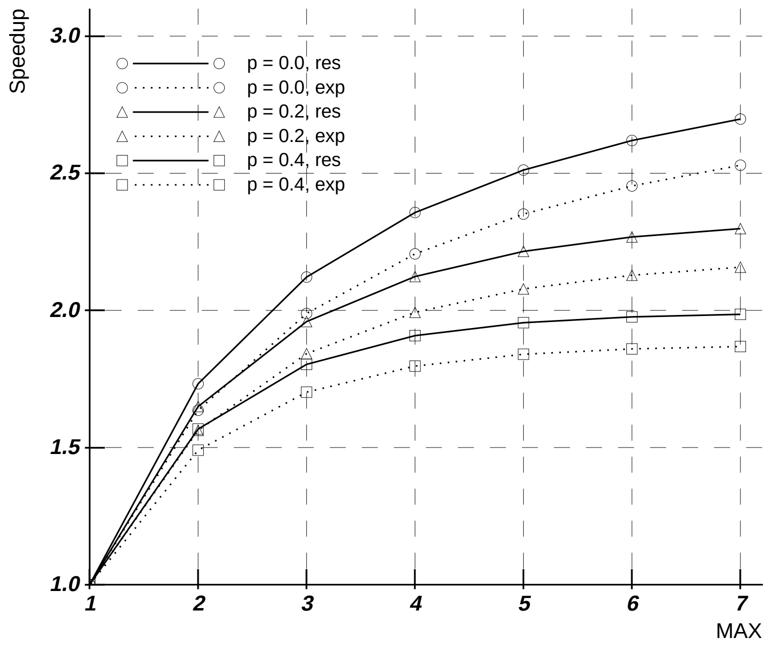

Figure 4.3: Speedup vs. MAX given different sending probabilities, $N = 3$.

exponential simplification. Since the simplification assumes a full inter-event interval just above GVT, fewer events will be committed after every GVT advance, yielding the smaller speedups.

Figure 4.4 shows speedup vs. N as a function of the communication cost, μ. A MAX of three is assumed. All other assumptions are the same. Once N increases beyond one, speedup is linearly related to N, but decreases significantly as communication cost increases. Figure 4.5 shows the fraction of time various message types are on the channel for the same set of runs. The bars are divided into simulation messages and GVT messages. As communication cost increases, the time spent on simulation messages decreases, while the time spent on GVT messages increases. It would seem that it is the increased time needed to determine a regulator that is affecting speedup.

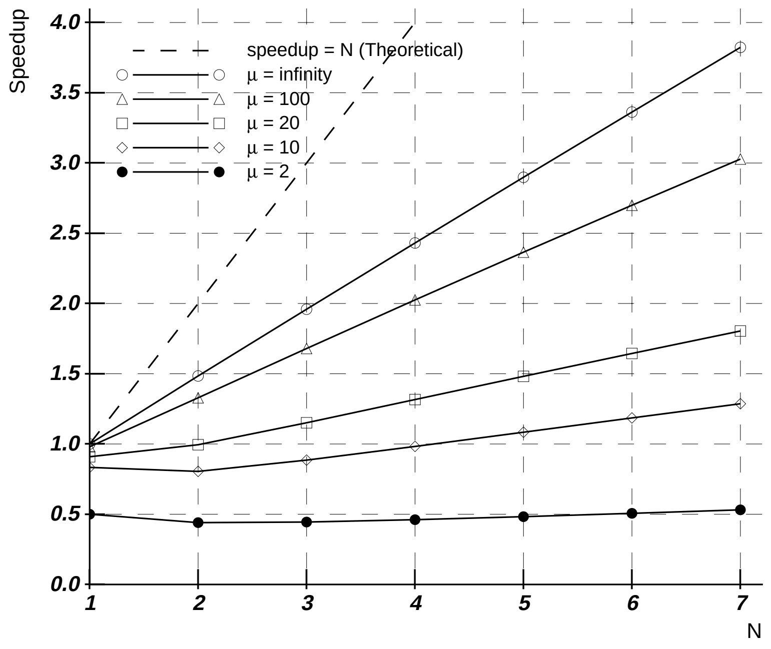

Figure 4.4: Speedup vs. N given different values of communication cost, $MAX = 3$.

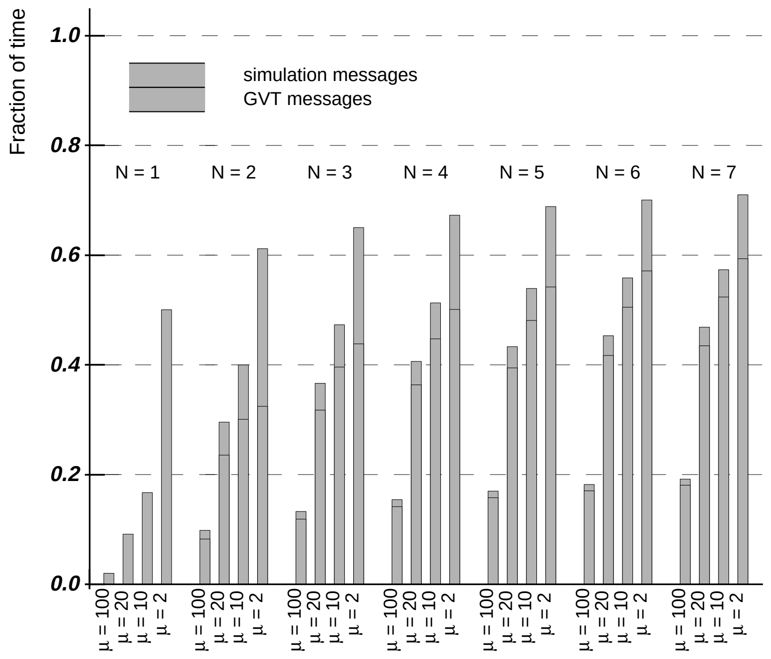

Figure 4.5: Channel utilization given different values of N and communication cost, $MAX = 3$.

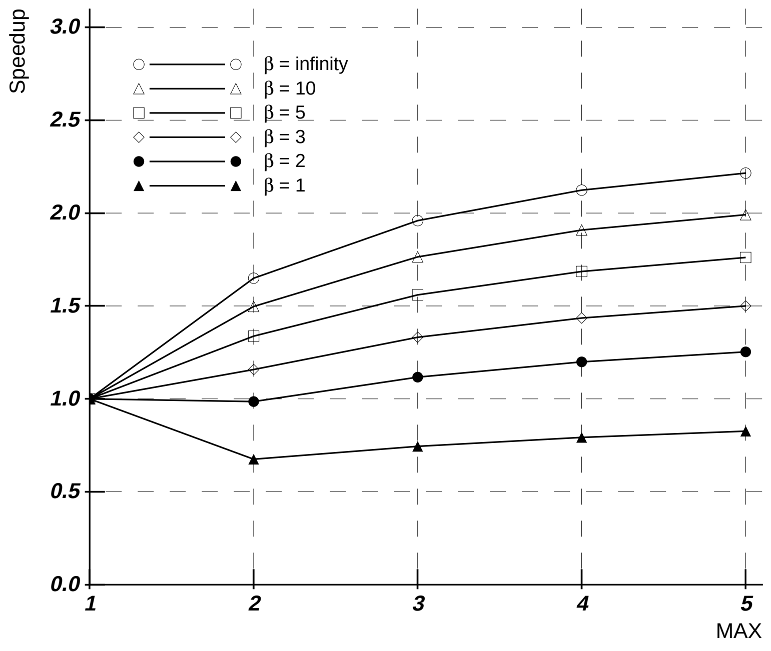

Figure 4.6: Speedup vs. MAX given different values of β, the fixed rate rollback cost, $N = 3$.

4.2 Results for Model II

Figure 4.6 shows speedup vs. MAX for different values of β. These runs were done with Model II. To reiterate, β is the part of rollback cost that does not depend on the number of events being rolled back. All other assumptions are the same, except μ (communication cost) was put back at 100,000. There is no model limitation against varying both parameters at the same time, but this would make the results less clear. As can be seen, beyond a MAX of one, speedup decreases as the rollback cost increases. There are no rollbacks when MAX equals one, and therefore that speedup does not change. An interesting point to note is that even for the curves with very high rollback cost, speedup always increases with more optimism. Figure 4.7 shows the fractions of

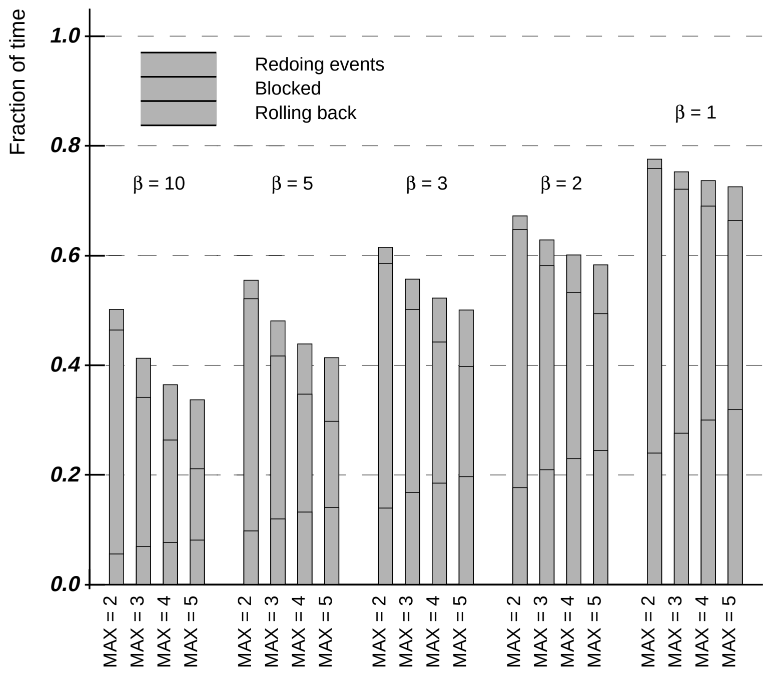

Figure 4.7: Fraction of time redoing events, blocked, and rolling back for Model II, $N = 3$.

time each processor is redoing events, blocked, and rolling back. The space between the bar and 1.0 represents the fraction of time each processor is doing useful work.

4.3 Results for Model III

Figure 4.8 shows speedup vs. MAX for different values of α, and β set at ten. These runs were done with Model III. To reiterate, α is the part of the rollback cost proportional to the number of events rolled back. As expected, lower values of α decrease speedup. It is interesting to note that the slopes do decrease and eventually go negative with decreasing α. The implication is that if there exists a component of the rollback cost that is proportional to rollback distance, there exists an optimal level of optimism, beyond which speedup will decrease. Figure 4.9 shows the fractions of time each processor is

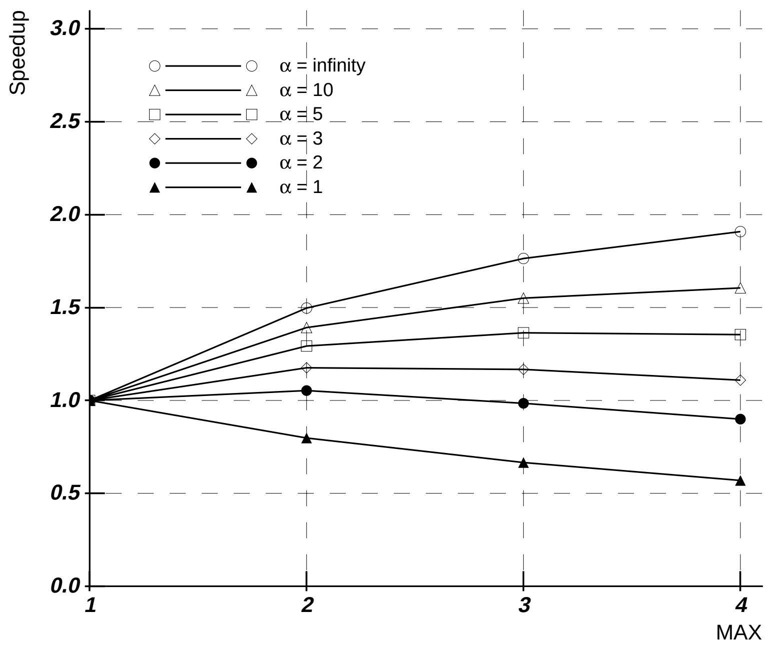

Figure 4.8: Speedup vs. MAX given different values of α, the rollback cost added for each rolled back event, $N = 3$.

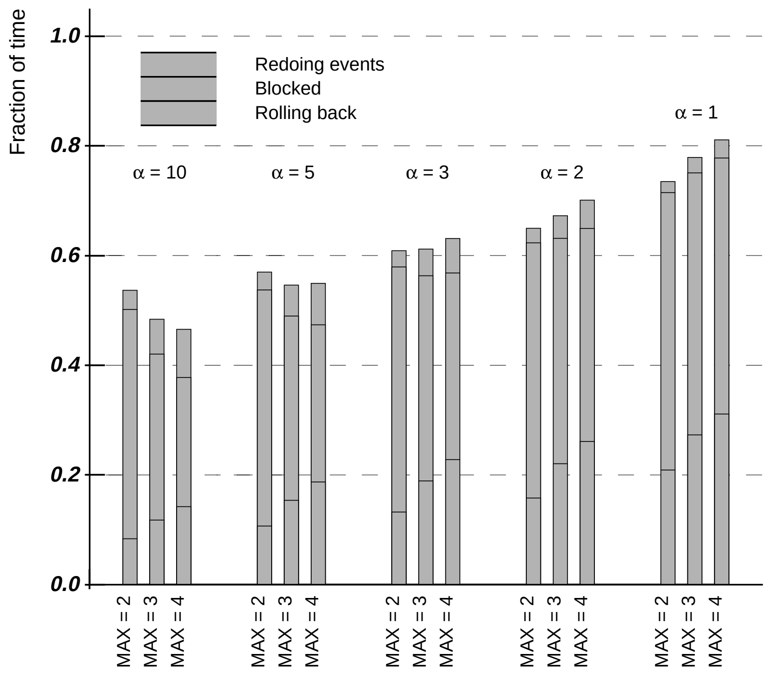

Figure 4.9: Fraction of time redoing events, blocked, and rolling back for Model III, $N = 3$.

redoing events, blocked, and rolling back. The space between the bar and 1.0 represents the fraction of time each processor is doing useful work.

CHAPTER 5

CONCLUSIONS

This thesis has presented a SAN model to analytically determine the performance of Time Warp, including speedup, fraction of time blocked, fraction of time rolling back, and channel utilization. By defining the model at the SAN level rather than the state level, we have been able to build a much more complex model than has previously been done. The model is a self-initiating model with N processes interacting probabilistically. By expressing the virtual time of each process as a number of events, rather than a number of clock ticks, we were able to consider continuously distributed simulation time. The distribution of the residual time of the first event above GVT was derived. Numerical integration was then used to calculate a set of probabilities related to comparisons of sums of exponentially distributed random variables and this residual distribution.

These probabilities were incorporated into a SAN model that considers the combined effects of limited optimism, cascaded rollback, communication cost, and rollback cost. Speedup was shown to increase linearly with N, given a fixed level of optimism and no overheads. As communication cost was varied, speedups decreased, but still increased linearly with N. Rollback cost was divided into a fixed cost, representing state restoration, and a cost proportional to rollback distance, representing event processing. As the fixed cost increased, speedup decreased, but it was shown that speedup always increases with added optimism. When the cost proportional to rollback distance was added, it was shown that speedup increases with added optimism only to a certain point, after which speedup decreases. This suggests the presence of an optimal level of optimism.

This model is extremely flexible and can be used to analyze several other factors critical to Time Warp performance. For example, the effect of the state-saving interval can be studied by adding a place to the model representing how many events need to be processed before saving state. The frequency of GVT calculation could be considered in a similar way. Unfortunately, these enhancements will add a significant number of states to the Markov process, making solution more difficult. Future research will attempt to simplify the model in order to reduce the state space without eliminating its flexibility.

APPENDIX A PROJECT DOCUMENTATION

This appendix will give the details of the SAN model for a Time Warp process. Tables are given with the C code that defines the attributes of each component in Figure 3.2.

A.1 Activity definitions

Table A.1 lists the activities of the model and their distributions. Activities *anti_or_-not* and *how_many* are instantaneous. All other activities are timed and exponentially distributed. Activity *choose_reg* represents the time to put a GVT message on the channel. Its rate is μ, which is 100,000 in this example. Activity *process* represents the time to process a simulation event. Its rate is γ, which is 10 in this example. Activity *recv* represents the time to pull a message off the channel. Its rate is μ, which is 100,000 in this example. Activity *rollback* represents the time to process a rollback. Its rate is dependent on the marking of place *rolling.* Table A.2 shows what the different markings of place *rolling* mean. Therefore, activity *rollback* has a rate of β if Model II is being considered, and a rate of $1/((MARK(rolling) - 1)/\alpha + 1/\beta)$ if Model III is being considered.

Table A.3 shows the case probabilities for activities *anti_or_not* and *process.* They both have values of p and $1 - p$, where p equals 0.2 in this example.

Tables A.4 and A.5 give the case probabilities for activity *choose_reg.* These probabilities are used to choose a new regulator, once the old regulator advances. The file "../../probs/prob_vec.h" contains values for $P_{Reg(k)}$ for all possible markings of place *vt_vector.* Place *vt_vector* is a coded vector of the virtual times of each process. Its

Table A.1: Activity Time Distributions

Activity	**Distribution**	**Rate**
anti_or_not	*instantaneous*	
how_many	*instantaneous*	
choose_reg	*exponential*	100000
process	*exponential*	10
recv	*exponential*	100000

rollback	*exponential*	*double alpha* = 3.0;
		double beta = 1.0;
		
		if (*MARK*(*rolling*) == 9998)
		    return(*beta*);
		else
		    return(1.0 / ((*MARK*(*rolling*) − 1) / *alpha*
		        + 1.0 / *beta*))

Table A.2: Coding of place *rolling*

Marking	**Meaning**
0	Model III, no rollback in progress
x	Model III, rollback of $x-1$ events in progress
9997	Model II, no rollback in progress
9998	Model II, rollback in progress
9999	Model I

Table A.3: Case Probabilities for Activities

Activity	**Case**	**Probability**
anti_or_not	1	0.2
	2	0.8
process	1	0.2
	2	0.8

Table A.4: Case Probabilities for Activities (continued)

Case	**Probability**
	choose_reg
1	*double result = 1.0;*
	#include "../../probs/prob_vec.h"
	if (MARK(MAX) >= 1 && prob_vec[0][MARK(vt_vector)] != ZERO)
	result -= prob_vec[0][MARK(vt_vector)];
	if (MARK(MAX) >= 2 && prob_vec[1][MARK(vt_vector)] != ZERO)
	result -= prob_vec[1][MARK(vt_vector)];
	if (MARK(MAX) >= 3 && prob_vec[2][MARK(vt_vector)] != ZERO)
	result -= prob_vec[2][MARK(vt_vector)];
	if (MARK(MAX) >= 4 && prob_vec[3][MARK(vt_vector)] != ZERO)
	result -= prob_vec[3][MARK(vt_vector)];
	if (MARK(MAX) >= 5 && prob_vec[4][MARK(vt_vector)] != ZERO)
	result -= prob_vec[4][MARK(vt_vector)];
	if (MARK(MAX) >= 6 && prob_vec[5][MARK(vt_vector)] != ZERO)
	result -= prob_vec[5][MARK(vt_vector)];
	if (MARK(MAX) >= 7 && prob_vec[6][MARK(vt_vector)] != ZERO)
	result -= prob_vec[6][MARK(vt_vector)];
	if (MARK(MAX) >= 8 && prob_vec[7][MARK(vt_vector)] != ZERO)
	result -= prob_vec[7][MARK(vt_vector)];
	return(result);
2	*#include "../../probs/prob_vec.h"*
	if (MARK(MAX) >= 1)
	return(prob_vec[0][MARK(vt_vector)]);
	else
	return(ZERO);
3	*#include "../../probs/prob_vec.h"*
	if (MARK(MAX) >= 2)
	return(prob_vec[1][MARK(vt_vector)]);
	else
	return(ZERO);

Table A.5: Case Probabilities for Activities (continued)

Case	**Probability**

4	#include “../../probs/prob_vec.h”
	
	if (MARK(MAX) >= 3)
	    return(prob_vec[2][MARK(vt_vector)]);
	else
	    return(ZERO);

5	#include “../../probs/prob_vec.h”
	
	if (MARK(MAX) >= 4)
	    return(prob_vec[3][MARK(vt_vector)]);
	else
	    return(ZERO);

6	#include “../../probs/prob_vec.h”
	
	if (MARK(MAX) >= 5)
	    return(prob_vec[4][MARK(vt_vector)]);
	else
	    return(ZERO);

7	#include “../../probs/prob_vec.h”
	
	if (MARK(MAX) >= 6)
	    return(prob_vec[5][MARK(vt_vector)]);
	else
	    return(ZERO);

8	#include “../../probs/prob_vec.h”
	
	if (MARK(MAX) >= 7)
	    return(prob_vec[6][MARK(vt_vector)]);
	else
	    return(ZERO);

9	#include “../../probs/prob_vec.h”
	
	if (MARK(MAX) >= 8)
	    return(prob_vec[7][MARK(vt_vector)]);
	else
	    return(ZERO);

marking is:

$$\sum_{k=0}^{MAX-1} V_k(N+1)^k,$$

where N is the number of processes in the model, and V_k is the number of processes with k processed, uncommitted events. The GVT regulator is not included in this value. Cases two through nine of activity *choose_reg* represent the probabilities that a process with zero through seven processed, uncommitted events, respectively, will be the next regulator. The probability associated with case one is one minus the sum of these probabilities, which equates to $P_{Reg(same)}$, which is the probability that this process is still the GVT regulator.

Tables A.6 and A.7 represent the case probabilities of activity *how_many.* They are used to determine how many events are rolled back and committed. They are dependent on the marking of place *action.* Table A.8 shows how place *action* is encoded to represent simulation and GVT messages that need to be processed.

The file "../../probs/prob_r.h" contains values of $P_{ww}[i,j]$ and $P_{ew}[j]$ for all i and j in $[0, MAX]$. These probabilities are used to calculate how many events are rolled back if a simulation message is in place *action*, or how many events are committed if a GVT message is in place *action.*

A.2 Input gate definitions

Tables A.9, A.10, and A.11 give the definitions of the input gates in the model. Input gate *limit* controls activity *process.* Activity *process* is enabled if:

Table A.6: Case Probabilities for Activities (continued)

Case	Probability
how_many	

1	#include "../../probs/prob_r.h"
	
	if (MARK(action) / 100 == 1) /* GVT message */
	{
	    if (MARK(uncom) == MARK(num_done))
	        return(ZERO);
	
	    else if (MARK(action) == 199) /* reg advanced, still reg */
	        return(
	            (prob_er[MARK(uncom) + 1] − prob_er[MARK(num_done) + 1]) /
	            (prob_er[MARK(uncom) + 1] − prob_er[MARK(num_done)]));
	
	    else
	    {
	        int reg_vt = MARK(action) − 100 + 1;
	
	        return(
	            (prob_rr[reg_vt][MARK(uncom) + 1] −
	                prob_rr[reg_vt][MARK(num_done) + 1]) /
	            (prob_rr[reg_vt][MARK(uncom) + 1] −
	                prob_rr[reg_vt][MARK(num_done)]));
	    }
	}
	
	else /* simmessage */
	{
	    int msg_vt = MARK(action);
	
	    if (MARK(uncom) == MARK(num_done) − 1)
	        return(ZERO);
	
	    else if (MARK(num_done) == 0) /* first time */
	        return(prob_rr[msg_vt][MARK(uncom) + 1]);
	
	    else
	        return(
	            prob_rr[msg_vt][MARK(uncom) − MARK(num_done) + 1] /
	            prob_rr[msg_vt][MARK(uncom) − MARK(num_done) + 2]);
	}

Table A.7: Case Probabilities for Activities (continued)

Case	**Probability**
	how_many

2	#include "../../probs/prob_r.h"
	
	if (MARK(action) / 100 == 1) /* GVT message */
	{
	   if (MARK(uncom) == MARK(num_done))
	      return(1.0);
	
	   else if (MARK(action) == 199) /* reg advanced, still reg */
	      return(1.0 −
	         (prob_er[MARK(uncom) + 1] − prob_er[MARK(num_done) + 1]) /
	         (prob_er[MARK(uncom) + 1] − prob_er[MARK(num_done)]));
	
	   else
	   {
	      int reg_vt = MARK(action) − 100 + 1;
	
	      return(1.0 −
	         (prob_rr[reg_vt][MARK(uncom) + 1] −
	            prob_rr[reg_vt][MARK(num_done) + 1]) /
	         (prob_rr[reg_vt][MARK(uncom) + 1] −
	            prob_rr[reg_vt][MARK(num_done)]));
	   }
	}
	
	else /* simmessage */
	{
	   int msg_vt = MARK(action);
	
	   if (MARK(uncom) == MARK(num_done) − 1)
	      return(1.0);
	
	   else if (MARK(num_done) == 0) /* first time */
	      return(1.0 − prob_rr[msg_vt][MARK(uncom) + 1]);
	
	   else
	      return(1.0 −
	         prob_rr[msg_vt][MARK(uncom) − MARK(num_done) + 1] /
	         prob_rr[msg_vt][MARK(uncom) − MARK(num_done) + 2]);
	}

Table A.8: Encoding of place *action*

Marking	**Meaning**
0	Idle
$00x$	Simulation message with timestamp x
$10x$	GVT message, new regulator VT = x, regulator not available
199	GVT message, regulator advanced, still regulator

Table A.9: Input Gate Predicates and Functions

limit	
Predicate	$(MARK(uncom) < MARK(MAX) - 1 \mid\mid MARK(me\_reg) == 1)$ && $(MARK(rolling) == 0 \mid\mid MARK(rolling) == 9999 \mid\mid$ $MARK(rolling) == 9997)$ && $MARK(channel) != 9999$
Function	*identity*
cost_or_not	
Predicate	$MARK(rolling) != 0$ && $MARK(rolling) != 9997$ && $MARK(rolling) != 9999$
Function	$if\ (MARK(rolling) == 9998)$ $MARK(rolling) = 9997;$ $else$ $MARK(rolling) = 0;$
hold_reg	
Predicate	$MARK(channel) == 0$
Function	$MARK(my\_cycle) = 1 - MARK(my\_cycle);$ $MARK(me\_reg) = 0;$ $MARK(num\_to\_recv) = MARK(N) - 1;$ $MARK(vt\_vector) += 1;$
get_info	
Predicate	$/*$ *read from the channel if its a simmessage,* *or GVT message while cycles match* $*/$ $MARK(channel) / 100 >= 3 \mid\mid$ $(MARK(channel) > 0$ && $MARK(cycle) == MARK(my\_cycle))$

Table A.10: Input Gate Predicates and Functions (continued)

get_info (continued)

Function

```
/* initialization condition for picking first regulator */
if (MARK(channel) == 9999)
{
   MARK(me_reg) = 1;
   MARK(channel) = 0;
   MARK(vt_vector) = MARK(N) - 1;
}

else if (MARK(channel) / 100 >= 3)  /* simmessage */
{
   /* message from the regulator */
   if (MARK(channel) == 300)
   {
      /* ignore it if you are the regulator, else roll everything back */
      if (MARK(me_reg) == 0)
      {
         if (MARK(rolling) != 9999 && MARK(MAX) != 1)
         {
            if (MARK(rolling) < 9997)
               MARK(rolling) = MARK(uncom) + 1;
            else
               MARK(rolling) = 9998;
         }

         MARK(to_redo) += MARK(uncom);
         MARK(vt_vector) += 1 -
            pow(1.0 * (MARK(N) + 1), 1.0 * MARK(uncom));
         MARK(do_anti) = MARK(uncom);
         MARK(uncom) = 0;
      }

      MARK(channel) = 0;     /* clear the channel */
   }

   else    /* peel off the first message */
   {
      int pos = 1;

      while ((MARK(channel) - 300) % (int) pow(1.0
            * MARK(N), 1.0 * pos) == 0)
         ++pos;
```

Table A.11: Input Gate Predicates and Functions (continued)

get_info (continued)

```
        /* ignore it if the regulator */
        if (MARK(me_reg) == 0)
            MARK(action) = pos;

        MARK(channel) -= pow(1.0 * MARK(N), 1.0 * (pos - 1));

        /* clear the channel, if last message */
        if (MARK(channel) == 300)
            MARK(channel) = 0;
    }
}

else    /* GVT message */
{
    /* if regulator available, and eligible, take it */
    if (MARK(channel) / 100 == 2 &&
        MARK(channel) % 100 == MARK(uncom))
    {
        MARK(me_reg) = 1;
        MARK(channel) -= 100;
        MARK(vt_vector) -=
            pow(1.0 * (MARK(N) + 1), 1.0 * MARK(uncom));
        MARK(uncom) = 0;
    }

    else
        MARK(action) = 100 + MARK(channel) % 100;

    MARK(my_cycle) = 1 - MARK(my_cycle);
    --MARK(num_to_recv);

    if (MARK(num_to_recv) == 0)
    {
        MARK(channel) = 0;
        MARK(cycle) = 1 - MARK(cycle);
    }
}
```

Table A.12: Encoding of place *channel*

Marking	**Meaning**
0	Idle
$10x$	GVT message, new regulator VT = x, regulator not available
199	GVT message, regulator advanced, still regulator
$20x$	GVT message, new regulator VT = x, regulator available
300	Simulation message from the regulator
$30x$	Simulation message, $x = \sum_{i=0}^{MAX-1} M(i)N^i$, where $M(i)$ = number of messages with timestamp i
9999	Initialization

- MAX has not been exceeded or this process is the regulator, and
- this process is not rolling back, and
- 9,999 is not on the channel, which is the initialization condition.

If these conditions are true, then this process is processing events. Input gate *cost_or_not* enables activity *rollback* if a rollback is in progress, as explained by table A.2. The function resets the rollback condition.

Input gate *hold_reg* enables activity *choose_reg* when the channel is free. Its function toggles place *my_cycle* and sets place *num_to_recv* for the broadcast of the GVT message. It also sets place *me_reg* to zero, although it might be set right back to one, if the process remains the regulator. Place *vt_vector* is incremented, since this process is no longer the regulator, and there is now one more process with virtual time equal to zero.

Table A.12 shows how place *channel* is encoded to represent simulation and GVT messages on the channel. As can be seen, multiple simulation messages can be on the channel at one time.

Input gate *get_info* controls activity *recv*. It pulls a message off the channel if there is a simulation message, or if there is a GVT message, and *cycle* matches *my_cycle*. The function is fairly complicated. If the process received the initialization condition (9,999), the process makes itself the GVT regulator, clears the channel, and initializes *vt_vector*. If the process received a simulation message from the regulator, all progress is rolled back. All other messages are put into place *action*, so that activity *how_many* can probabilistically determine how many events to roll back. If the message was a GVT message, the process first checks to see if the role of regulator is still available and if its value of *uncom* matches the value of the new regulator. If so, this process becomes the new regulator. If not, the message is put into place *action*, so that activity *how_many* can probabilistically determine how many events to commit.

A.3 Output gate definitions

Table A.13 shows the functions of output gates *do_send* and *no_send*, one of which is executed after processing each simulation event. Output gate *do_send* is executed if the first case probability of activity *process* is chosen, which indicates that a simulation message is to be sent. Output gate *no_send* is executed is no simulation message is to be sent.

Table A.14 shows the function of output gates *more* and *no_more*. If output gate *more* is executed, it means that activity *how_many* has determined that either another

Table A.13: Output Gate Functions

Gate	**Function**

do_send	if (MARK(to_redo) > 0)
	    -- MARK(to_redo);
	
	if (MARK(channel) > 0)
	    ++ MARK(to_redo);
	else
	{
	    if (MARK(me_reg) == 0) /* not the regulator */
	    {
	        ++ MARK(uncom);
	
	        MARK(vt_vector) +=
	            pow(1.0 * (MARK(N) + 1), 1.0 * MARK(uncom)) −
	            pow(1.0 * (MARK(N) + 1), 1.0 * (MARK(uncom) − 1));
	
	        MARK(channel) =
	            300 + pow(1.0 * MARK(N), 1.0 * (MARK(uncom) − 1));
	    }
	
	    else /* the regulator */
	    {
	        MARK(new_reg) = 1;
	        MARK(channel) = 300; /* message from the regulator */
	    }
	}

no_send	if (MARK(to_redo) > 0)
	    -- MARK(to_redo);
	
	if (MARK(channel) > 0)
	    ++ MARK(to_redo);
	else
	{
	    if (MARK(me_reg) == 0) /* not the regulator */
	    {
	        ++ MARK(uncom);
	
	        MARK(vt_vector) +=
	            pow(1.0 * (MARK(N) + 1), 1.0 * MARK(uncom)) −
	            pow(1.0 * (MARK(N) + 1), 1.0 * (MARK(uncom) − 1));
	    }
	
	    else /* the regulator */
	        MARK(new_reg) = 1;
	}

Table A.14: Output Gate Functions (continued)

Gate	**Function**

more	++MARK(action);
	++MARK(num_done);

no_more	++MARK(action);
	
	if (MARK(num_done) > 0)
	{
	    /* num_moved is one less for messages, because the first one is for
	       the event currently being processed */
	    int num_moved;
	    if (MARK(action) / 100 == 0)
	        num_moved = MARK(num_done) − 1;
	    else
	        num_moved = MARK(num_done);
	
	    MARK(vt_vector) += pow(1.0 * (MARK(N) + 1),
	        1.0 * (MARK(uncom) − num_moved)) −
	        pow(1.0 * (MARK(N) + 1), 1.0 * MARK(uncom));
	    MARK(uncom) −= num_moved;
	
	    if (MARK(action) / 100 == 0) /* simmessage */
	    {
	        MARK(do_anti) = num_moved;
	        MARK(to_redo) += num_moved;
	
	        /* only set rolling if rollback costs are being considered,
	            and was processing */
	        if (MARK(rolling) != 9999 && !(MARK(uncom) ==
	            MARK(MAX) − 1 && num_moved == 0))
	        {
	            if (MARK(rolling) < 9997)
	                MARK(rolling) = num_moved + 1;
	            else
	                MARK(rolling) = 9998;
	        }
	}
	
	MARK(num_done) = 0;
	MARK(action) = 0;

Table A.15: Output Gate Functions (continued)

Gate	**Function**
send_anti	$if\ (MARK(channel)\ ==\ 0)$ $/*\ first\ message\ on\ channel\ */$ $MARK(channel)\ =\ 300;$ $MARK(channel) += pow(1.0 * MARK(N),$ $1.0 * (MARK(uncom) + MARK(do\_anti)));$
same_gvt	$MARK(me\_reg)\ =\ 1;$ $MARK(vt\_vector) -= 1;$ $if\ (MARK(N)\ >\ 1)$ $MARK(channel)\ =\ 199;$ $else$ $MARK(my\_cycle)\ =\ 1\ -\ MARK(my\_cycle);$
give_to_0	$MARK(channel)\ =\ 200;$
give_to_1	$MARK(channel)\ =\ 201;$
give_to_2	$MARK(channel)\ =\ 202;$
give_to_3	$MARK(channel)\ =\ 203;$
give_to_4	$MARK(channel)\ =\ 204;$
give_to_5	$MARK(channel)\ =\ 205;$
give_to_6	$MARK(channel)\ =\ 206;$
give_to_7	$MARK(channel)\ =\ 207;$

event must be rolled back or committed, depending on what type of message is in place *action*. Place *num_done* represents the number of events to be rolled back or committed for this message. Place *action* is incremented, because it was decremented when activity *how_many* completed. If activity *no_more* is executed, it means that activity *how_many* is done processing this message, and no more events are to be rolled back or committed.

Table A.15 gives the functions of the rest of the output gates. Output gate *send_anti* is executed for every anti-message that is sent. After activity *choose_reg* completes, either output gate *same_gvt* or one of the *give_to_i* output gates is executed. Output gate *same_gvt* puts a message on the channel, indicating that the regulator has advanced, but

Table A.16: Performance variables

Speedup	
Predicate	$(MARK(me\_reg)\ ==\ 1 \Vert MARK(uncom)\ < MARK(MAX)\ -\ 1)\ \&\&\ (MARK(rolling)\ ==\ 0 \Vert MARK(rolling)\ ==\ 9997 \Vert MARK(rolling)\ ==\ 9999)\ \&\&\ MARK(to\_redo)\ == 0$
Function	1.0
Fraction of time redoing events	
Predicate	$(MARK(me\_reg)\ ==\ 1 \Vert MARK(uncom)\ <\ MARK(MAX)\ -\ 1)\ \&\&\ (MARK(rolling)\ ==\ 0 \Vert MARK(rolling)\ ==\ 9997 \Vert MARK(rolling)\ ==\ 9999)\ \&\&\ MARK(to\_redo)\ >\ 0$
Function	$1.0\ /\ MARK(N)$
Fraction of time blocked	
Predicate	$MARK(me\_reg)\ ==\ 0\ \&\&\ MARK(uncom)\ ==\ MARK(MAX)\ -\ 1\ \&\&\ MARK(channel)\ !=\ 9999$
Function	$1.0\ /\ MARK(N)$
Fraction of time rolling back	
Predicate	$MARK(rolling)\ !=\ 0\ \&\&\ MARK(rolling)\ !=\ 9997\ \&\&\ MARK(rolling)\ !=\ 9999$
Function	$1.0\ /\ MARK(N)$
Channel utilization – sim	
Predicate	$MARK(channel)\ /\ 100\ >=\ 3$
Function	$1.0\ /\ MARK(N)$
Channel utilization – GVT	
Predicate	$MARK(channel)\ /\ 100\ ==\ 1 \Vert MARK(channel)\ /\ 100\ ==\ 2$
Function	$1.0\ /\ MARK(N)$

is still the regulator. The *give_to_i* output gates put a message on the channel, indicating that the new regulator has i processed, uncommitted events.

A.4 Performance variable definitions

Table A.16 gives the performance variable definitions for the model. The first four are mutually exclusive. *Speedup* is defined as the fraction of time that each process is doing useful work (i.e. work that is not being redone), multiplied by the number of processes. When not doing useful work, each process is redoing events, blocked, or rolling back. *Fraction of time redoing events*, *Fraction of time blocked*, and *Fraction of time rolling back* are variables that show the amount of time spent in each of these states. *Channel utilization – sim* and *Channel utilization – GVT* show the fraction of the channel utilization that is due to each type of message.

REFERENCES

[1] R. M. Fujimoto, "Parallel discrete event simulation," *Communications of the ACM*, vol. 33, pp. 30–53, October 1990.

[2] D. Jefferson and H. Sowizral, "Fast concurrent simulation using the time warp mechanism," in *Proceedings of the 1985 SCS Multiconference on Distributed Simulation*, pp. 63–69, January 1985.

[3] D. R. Jefferson, "Virtual time," *ACM Transactions on Programming Languages and Systems*, vol. 7, pp. 404–425, July 1985.

[4] S. Lavenberg, R. Muntz, and B. Samadi, "Performance analysis of a rollback method for distributed simulation," in *PERFORMANCE '83* (A. K. Agrawala and S. K. Tripathi, eds.), pp. 117–132, North-Holland Publishing Company, 1983.

[5] D. Mitra and I. Mitrani, "Analysis and optimum performance of two message-passing parallel processors synchronized by rollback," in *PERFORMANCE '84* (E. Gelenbe, ed.), pp. 35–50, Elsevier Science Publishers B.V. (North-Holland), 1984.

[6] R. E. Felderman and L. Kleinrock, "Two processor Time Warp analysis: Some results on a unifying approach," in *Proceedings of the 1991 SCS Multiconference on Distributed Simulation*, pp. 3–10, 1991.

[7] L. Kleinrock, "On distributed systems performance," *Computer Networks and ISDN Systems*, vol. 20, pp. 209–215, 1990.

[8] R. E. Felderman and L. Kleinrock, "Two processor Time Warp analysis: Capturing the effects of message queueing and rollback / state saving costs," Tech. Rep. 920035, University of California, Los Angeles, Computer Science Department, June 1992. To appear in a special issue of the *International Journal of Electronics and Communications.*

[9] R. E. Felderman and L. Kleinrock, "An upper bound on the improvement of asynchronous versus synchronous distributed processing," in *Proceedings of the 1990 SCS Multiconference on Distributed Simulation*, pp. 131–136, 1990.

[10] R. E. Felderman and L. Kleinrock, "Bounds and approximations for self-initiating distributed simulation without lookahead," *ACM Transactions on Modeling and Computer Simulation*, vol. 1, pp. 386–406, October 1991.

[11] B. Lubachevsky, A. Weiss, and A. Shwartz, "An analysis of rollback-based simulation," *ACM Transactions on Modeling and Computer Simulation*, vol. 1, pp. 154–193, April 1991.

[12] Y.-B. Lin and E. D. Lazowska, "Optimality considerations of "Time Warp" parallel simulation," in *Proceedings of the 1990 SCS Multiconference on Distributed Simulation*, pp. 29–34, 1990.

[13] Y.-B. Lin and E. D. Lazowska, "A study of Time Warp rollback mechanisms," *ACM Transactions on Modeling and Computer Simulation*, vol. 1, pp. 51–72, January 1991.

[14] D. L. Eager, J. Zahorjan, and E. D. Lazowska, "Speedup versus efficiency in parallel systems," *IEEE Transactions on Computers*, vol. 38, pp. 408–423, March 1989.

[15] D. M. Nicol, "Performance bounds on parallel self-initiating discrete-event simulations," *ACM Transactions on Modeling and Computer Simulation*, vol. 1, pp. 24–50, January 1991.

[16] P. M. Dickens and P. F. Reynolds, Jr., "A performance model for parallel simulation," in *Proceedings of the 1991 Winter Simulation Conference* (B. L. Nelson, W. D. Kelton, and G. M. Clark, eds.), pp. 618–626, 1991.

[17] V. Madisetti, J. Walrand, and D. Messerschmitt, "Synchronization in message-passing computers - Models, algorithms and analysis," in *Proceedings of the 1990 SCS Multiconference on Distributed Simulation*, pp. 35–48, 1990.

[18] V. K. Madisetti, J. C. Walrand, and D. G. Messerschmitt, "Asynchronous algorithms for the parallel simulation of event-driven dynamical systems," *ACM Transactions on Modeling and Computer Simulation*, vol. 1, pp. 244–274, July 1991.

[19] V. K. Madisetti and D. A. Hardaker, "Synchronization mechanisms for distributed event-driven computation," *ACM Transactions on Modeling and Computer Simulation*, vol. 2, pp. 12–51, January 1992.

[20] A. Gupta, I. F. Akyildiz, and R. M. Fujimoto, "Performance analysis of Time Warp with multiple homogeneous processors," *IEEE Transactions on Software Engineering*, vol. 17, pp. 1–15, October 1991.

[21] I. F. Akyildiz, L. Chen, S. R. Das, R. M. Fujimoto, and R. F. Serfozo, "Performance analysis of "Time Warp" with limited memory," in *Performance Evaluation Review*, pp. 213–224, June 1992.

[22] I. F. Akyildiz, R. M. Fujimoto, and Y. Gong, "Performance analysis of Time Warp with multiple processes per processor." Submitted for publication.

[23] Y.-B. Lin and E. D. Lazowska, "Reducing the state saving overhead for Time Warp parallel simulation," Tech. Rep. 90-02-03, University of Washington, Seattle, WA, February 1990.

[24] J. F. Meyer, A. Movaghar, and W. H. Sanders, "Stochastic activity networks: Structure, behavior, and application," in *Proceedings of the International Conference on Timed Petri Nets*, (Torino, Italy), pp. 106–115, July 1985.

[25] J. A. Couvillion, R. Freire, R. Johnson, W. D. Obal II, M. A. Qureshi, M. Rai, W. H. Sanders, and J. E. Tvedt, "Performability modeling with UltraSAN," *IEEE Software*, pp. 69–80, 1991.

[26] B. D. Lubachevsky, "Efficient parallel simulations of asynchronous cellular arrays," *Complex Systems*, vol. 1, pp. 1099–1123, 1987.

[27] L. M. Sokol, D. P. Briscoe, and A. P. Wieland, "MTW: A strategy for scheduling discrete simulation events for concurrent execution," in *Proceedings of the 1988 SCS Multiconference on Distributed Simulation*, pp. 34–42, July 1988.

[28] D. Bertsekas and R. Gallager, *Data Networks.* Prentice-Hall, second ed., 1992.

[29] S. M. Ross, *Stochastic Processes.* John Wiley & Sons, 1983.

[30] *IMSL Math/Library User's Manual: FORTRAN subroutines for mathematical applications*, September 1991.

[31] W. H. Sanders and J. F. Meyer, "Reduced base model construction methods for stochastic activity networks," *IEEE Journal on Selected Areas in Communications*, vol. 9, pp. 25–36, January 1991. Special issue on Computer-Aided Modeling, Analysis, and Design of Communication Networks.