A Framework for Formalization and Characterization of Simulation Performance

BHAKTI SATYABUDHI STEPHAN ONGGO
B.Sc (Hons) in Computer Science, Sepuluh Nopember Institute of Technology (Surabaya, Indonesia)
M.Sc. in Operational Research, University of Lancaster (UK)

A THESIS SUBMITTED
FOR THE DEGREE OF DOCTOR OF PHILOSOPHY
DEPARTMENT OF COMPUTER SCIENCE
NATIONAL UNIVERSITY OF SINGAPORE

2004

DECLARATION

No portion of the work referred to in this thesis has been submitted in support of an application for another degree or qualification of this or any other university or other institution of learning.

Abstract

Researchers have lamented that the lack of adequate performance evaluation framework is one of the major obstacles hindering the widespread adoption of parallel discrete-event simulation [LIN93, LIU99, TEO99]. In this thesis, we propose a simulation performance characterization framework based on *time*, *space*, and *simulation event dependence*. The proposed performance characterization framework consists of two main parts.

First, we formalize simulation event orderings based on the partially ordered set (poset). This formalization provides a theoretical foundation for carrying out performance analysis of simulation. The formalization also facilitates the understanding of the relationship of different event orderings. In addition, it allows the performance of different event orderings to be evaluated independent of implementation overheads. We identify and formalize the event orderings of a number of simulation protocols including sequential simulator, Chandy-Misra-Bryant [CHAN79], Bounded-Lag [LUBA89], Time Warp [JEFF85], and Bounded Time Warp [TURN92].

Second, we characterize simulation performance based on the three simulation development process boundaries: physical system, simulation model, and simulator implementation. A *physical system* is viewed as a collection of interacting service centers. Events are ordered in a physical time order. At the *simulation model* layer, a physical system is modeled as a set of Logical Processes (LPs), and the interaction

among service centers in the physical system is modeled by exchanging events among LPs. Events at the physical system can be executed using different event orders at the simulation model layer to exploit event-level parallelism. At the *simulator* layer, LPs are mapped onto a number of Physical Processors. A simulator implements a synchronization algorithm (protocol) to maintain correct event ordering at runtime. The same event order at the simulator layer can be implemented using different protocols at the simulator layer. The layered approach provides a framework to study the factors affecting simulation performance from the physical system to its simulator implementation.

Event parallelism is defined as the number of events executed per unit time. Therefore, its measurement is influenced by the unit of time which complicates performance comparison across layers because the time units used in the three layers differ. Therefore, normalization is required to allow comparison across layers. The simulation memory requirement can be divided into three components: the data structure that implements queues at the physical system layer, the data structure that implements event lists at the simulation model layer, and the data structure that holds overhead events at the simulator layer. To compare and quantify the degree of event dependency of simulation event orderings, we propose a relation called *stricter* and a measure called *strictness*, respectively. Strictness is independent of time; hence it can be used to compare performance across layers without normalization.

To measure time, space and strictness at the three layers, we develop two measurement tools, namely, SPaDES/Java (Structured Parallel Discrete-Event Simulator) and TSSA (Time, Space and Strictness Analyzer). We conduct two sets of experiments using the

two measurement tools. First, we validate the proposed framework using an *open* system called Multistage Interconnection Network (MIN) and a *closed* system called PHOLD [FUJI90]. We measure the time and space performance at the three layers as well as the strictness of a number of event orderings used in the simulation. Second, we apply our framework to study the performance and scalability of Ethernet simulation.

Acknowledgements

I would like to express my sincere gratitude to my advisor A/P Teo Yong Meng for his guidance and support that help me through my graduate study. During his supervision, I have learnt a great deal from him in research and technical writing skills. I am indebted to his constant encouragement especially at the times when I was demoralized due to my personal problems.

I would also like to express my gratitude to Dr. Tay Seng Chuan for his great support and constructive comments especially during the first two years of my research. I am also very grateful to A/P Wong Weng Fai and Prof. Yuen Chung Kwong for their inputs on the strictness of the memory consistency models.

I would like to take this opportunity to thank my friends Tan Cher Min, Tan Chun Peng, Chen Peng, Wang Haiguang, Peng Liang, Hu Yanjun, Ameya Virkar, and Johan Gozali for their support. Many thanks go to Karthik Senoy and Hariharan Sandanagobalane for helping me proofread the first draft of my thesis. I also owe a great deal of gratitude to Alexia Leong for giving me a lot of suggestions to improve the readability of my thesis.

Finally, I would like to thank my family and most importantly, my wife Ling Ling, without whose support, my thesis would not have been possible. Her ever-present love makes me capable to face the ups and downs throughout my doctoral study.

Contents

List of Figures

List of Tables

List of Definitions

List of Lemmas and Theorems

List of Notations

$a \ldots z$	Event name or variable name
$a_i, e_i, \ldots$	Event name or element of a set
a_i^t , d_i^t	Event arrival (a) and departure (d) of job i at (physical or simulation depending on the context) time t.
S, S_i	A set
E, E_i	A set of events (in event ordering) or a set of edges (in graph)
V	A set of vertices
$\|\|S\|\|$	The number of elements in set S
{} or $\varnothing$	Empty set
R, R_i	A relation, an order or an event order
(S, R)	A partial order set, S is a set and R is a set of comparable elements
(E, S_R)	An event ordering, E is a set of events and S_R is a set of comparable events based on event order R.
(V, E)	A graph
AC	Anti-chain
H	Height of a poset
N	Natural number
ψ	Average Concurrent Event
$priority(x)$	The priority of event x, it is used to break a tie in total event order
W	Window size
$I(x)$	Interval of event x, i.e. $[start(x), end(x)]$
$\bullet$	Such that, used outside set definition, e.g.: $x \bullet y$ means x such that y
$\{ x \mid y \bullet z\}$	Set definition, e.g.: $S = \{ x \mid y \bullet z\}$ means set S contains x such that y and z. Part y refers to a membership specification, e.g.: $\forall$, $\exists$, $\notin$, $\in$, etc. Part z refers to a logical specification, e.g.: and, or, etc.
$\leftrightarrow$, iff	If and only if
$x \Rightarrow y$	Event x must be ordered before event y
$\lfloor y \rfloor$	Round down

e.pred	The predecessor of event *e*
e.ante	The antecedent of event *e*
e.timestamp	The simulation time when event *e* occurs.
e.lp	The LP where event *e* occurs
IB[*i*]	Input buffer *i*
OB[*i*]	Output buffer *i*
EL	Event List in parallel simulation
FEL	Future Event List, i.e. global event list in sequential simulation
SENDER(*x*)	A set of LPs that can send events to LP *x* (used in CMB protocol)
SC_i	Service Center *i*
LP_i	Logical process *i*
PP_i	Physical Processor *i*
m	Number of service centers / LPs; message density (PHOLD)
n	Number of PPs
k	Number of LPs mapped onto one PP.
ρ	Traffic intensity
Π	Event parallelism
Π^{prob}	Event parallelism at physical system layer
Π_{norm}^{prob}	Event parallelism at simulation problem (normalized)
Π^{ord}	Event parallelism at simulation model layer
Π_{norm}^{ord}	Event parallelism at simulation model layer (normalized)
Π^{sync}	Event parallelism at simulator layer
Π_{norm}^{sync}	Event parallelism at simulator layer (normalized)
M^{prob}	Space performance at physical system layer
M^{ord}	Space Performance at simulation model layer
M^{sync}	Memory requirement for synchronization overhead
M^{tot}	Total memory requirement (general definition)
M^{dst}	Total memory requirement for distributed memory architecture
M^{shr}	Total memory requirement for shared memory architecture
ς_R	Strictness of event ordering *R*
$<_\varsigma$	Stricter, e.g., $R_1 <_\varsigma R_2$ means R_2 stricter than R_1
D	Measurement duration

D^{prob}	Observation period (in physical time unit)
D^{ord}	Simulation duration (in timestep)
D^{sync}	Simulation execution time (in wall-clock time)
t_e	Average event execution time (simulator layer)
$Q_{i,t}$	The size of queue *i* at wall-clock time *t*
Qmax(D)	Maximum size of queue *i* during an observation period of *D*
$L_{i,t}$	The size of event list *i* at wall-clock time *t*
Lmax(D)	Maximum size of event list *i* during an observation period of *D*
$B_{i,t}$	The size of buffer *i* at wall-clock time *t*
Bmax(D)	Maximum size of buffer *i* during an observation period of *D*
Partial / Par	Partial event order
TI	Time-interval event order
TS	Timestamp event order
Total / Tot	Total event order
BL	Bounded Lag
BTW	Bounded Time Warp
CMB	Chandy-Misra-Bryant
TW	Time Warp

Chapter 1

Introduction

Simulation has been widely used to model real world systems [TROP02, BANK03]. As real world systems become more complex, their simulations tend to become computationally expensive [MART03]. Consequently, it has become increasingly important to understand simulation performance.

At the same time, the advent of parallel and distributed computing technologies has made parallel and distributed simulation (PADS) possible. PADS research in the last decade has resulted in a number of synchronization protocols [FUJI00]. Performance evaluations are carried out by comparing protocols [FUJI00]. However, performance metrics and benchmarks vary among different studies, resulting in the lack of a uniform framework where results can be compared easily [MART03].

In this introductory chapter, we provide a brief review of discrete-event simulation technology, elaborate on state-of-the-art simulation performance evaluation, and describe the objective of this research.

1.1 Discrete-event Simulation

One of the oldest tools in system analysis is simulation. The operations of a real-world system or *physical system* are modeled and implemented as a simulation program [BANK00]. Simulation models can be classified into several categories based on the characteristics shown in Figure 1.1. Based on the characteristic of time, simulation models can be divided into two categories, i.e., static and dynamic. In *static simulation*, changes in the *state of the system* (or *system state*) are independent of time, in contrast to *dynamic simulation* where the system state changes with time. Based on the changes in the system state with respect to time, dynamic simulation models may be further classified into continuous and discrete models. In *continuous simulation*, the system state changes continuously with time. A real system is often modeled using a set of differential equations. The system state in *discrete simulation* changes only at discrete points of time. Time can be advanced using a fixed time increment or irregular time increment. The former is known as *time-stepped simulation*. This thesis concentrates on the latter, which is termed *discrete-event simulation*. It should be noted that fixed time increment simulations can also be implemented as discrete-event simulation [BANK00].

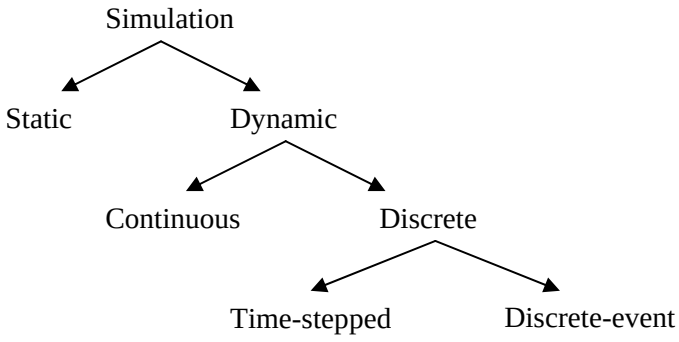


Figure 1.1: Simulation Model Taxonomy [LAW02]

There are three major world-views on simulation modeling, i.e., *activity-oriented*, *process-oriented*, and *event-oriented*. The most frequently used world-views are event-oriented and process-oriented [WONN96]. Since the process-oriented world-view is built on top of the event-oriented world-view, we focus on the event-oriented world-view in our performance characterization. Detailed descriptions of other world-views can be found in [BUXT62, LAW84, RUSS87].

As the name implies, the unit of work in the event-oriented world-view is an event. An event is an instantaneous occurrence that may change the state of a system and schedule other events. In a simulation program, the system state is implemented as a collection of state variables representing the event of interest in a real system. For example, the arrival of a customer in a bank increases the number of people waiting for service or makes the idle teller busy. The number of customers in the queue and the teller status are examples of state variables. A simulator modeler must implement an event handler for each type of events to manipulate the system state and to schedule new events. Scheduled events are sorted based on their time of occurrence in a list called the *future event list* (FEL).

In general, the sequential simulation process retrieves the event with the smallest time, advances the simulation clock, and executes an appropriate event handler which may change the system state and/or schedule another set of events. These steps will be repeated until a stopping condition has been met, as shown in Figure 1.2.

There are some complex systems which take hours or even days to simulate using sequential simulation. Furthermore, limitations in computer resources (such as memory

capacity) can make the simulation of a complex system using a sequential simulator intractable. The following examples show situations where sequential simulation takes a very long time.

```
1. while (stopping condition has not been met) {
2.    remove event e with the smallest timestamp from FEL
3.    simulation_clock = e.timestamp
4.    execute (e)
5.    add the generated events, if any, to FEL
6. }
```

Figure 1.2: Sequential Simulation Algorithm

Personal Communication Systems (PCS) network simulation usually represents networks by hexagonal or square cells. Due to limited computing resources, most studies only examine small-scale networks containing fewer than 50 cells [ZHAN89, KUEK92]. Carothers et al. showed that in order to get unbiased output statistics, at least 256 cells are required with the simulation duration being at least 5×10^4 seconds [CARO95]. This minimal requirement produces in the order of 10^7 events. Ideally, a more complex PCS simulation should model thousands of cells which would translate into 10^9 or 10^{10} events. It is also computationally demanding to analyze an extreme condition of a physical system using simulation. For example, the analysis of overflows in Asynchronous Transfer Mode (ATM) switch buffers requires a simulation to execute more than 10^{12} events in order to get a valid result. This is because the probability of these rare events happening is around 10^{-9} [RONN95].

Efficient discrete-event simulation packages can execute between 10^4 and 10^5 events per second [CARO95], so that a single simulation run of PCS and ATM require 28 hours and four months, respectively. Furthermore, in a simulation study, we need to run a

simulation several times to get statistically correct results, and very often a system analyst has to compare several design alternatives.

Large scale Internet worm infestations such as Code Red, Code Red II, and Nimda may affect the network infrastructure, specifically surges in the routing traffic. Liljenstam et al. developed a simulation model for a large-scale worm infestation [LILJ02]. The execution time for the largest problem size in their experiment is approximately 30 hours. Bodoh and Wieland studied the performance of the Total Airport and Airspace Model (TAAM) [BODO03]. TAAM is a large air traffic simulation for aviation analysis. They noted that it is not practical to run TAAM using sequential simulation. This is because a simulation of a fraction of the traffic in the United States requires at least 35 hours. It is predicted that the simulation for the entire traffic in the United States would require at least 70 hours.

The PCS, ATM, Internet Worm, and TAAM simulation examples show that the size and complexity of a physical system can hinder the application of sequential simulation. The requirement of a faster simulation technique is even more important in a time critical system such as an air traffic controller. Parallel simulation offers an alternative.

1.2 Parallel Discrete-event Simulation

A physical system usually consists of several smaller subsystems with disjoint state variables. Parallel discrete-event simulation (PADS) uses this information to partition a simulation model into smaller components called *logical processes* (LPs). Parallelization in simulation is done by simulating LPs concurrently. There are two

potential benefits of implementing a parallel simulator: reduced execution time and facilitating the execution of larger models.

In simulation, *local causality constraint* (*lcc*) imposes that if event *a* happens before event *b* and both events happen at the same LP, then *a* must be executed before *b*. Parallel simulation must adhere to *lcc* to produce correct simulation results. Based on how *lcc* is maintained, parallel simulation protocols are grouped into two main categories: conservative and optimistic. *Conservative protocols* do not allow any *lcc* violation throughout the duration of the simulation. *Optimistic protocols* allow *lcc* violation, but provide mechanisms to rectify it.

1.2.1 Definitions of Time

Before we discuss the two protocols, it is important to understand the various definitions of time [FUJI00]:

1. *Physical time* refers to time in a physical system.
2. *Simulation time* or *timestamp* is an abstraction used by a simulation to model physical time.
3. *Wall-clock time* refers to the execution time of the simulation program.

1.2.2 Conservative Protocols

Conservative protocols strictly avoid the violation of *lcc* by conservatively executing "safe" events only. An event is safe to be executed if it is guaranteed that no other event with a smaller timestamp will arrive at a later time. In PADS, it is possible that an event

with a smaller timestamp will arrive at a later time (i.e., a *straggler event*) because the time advancement in every LP may not be the same.

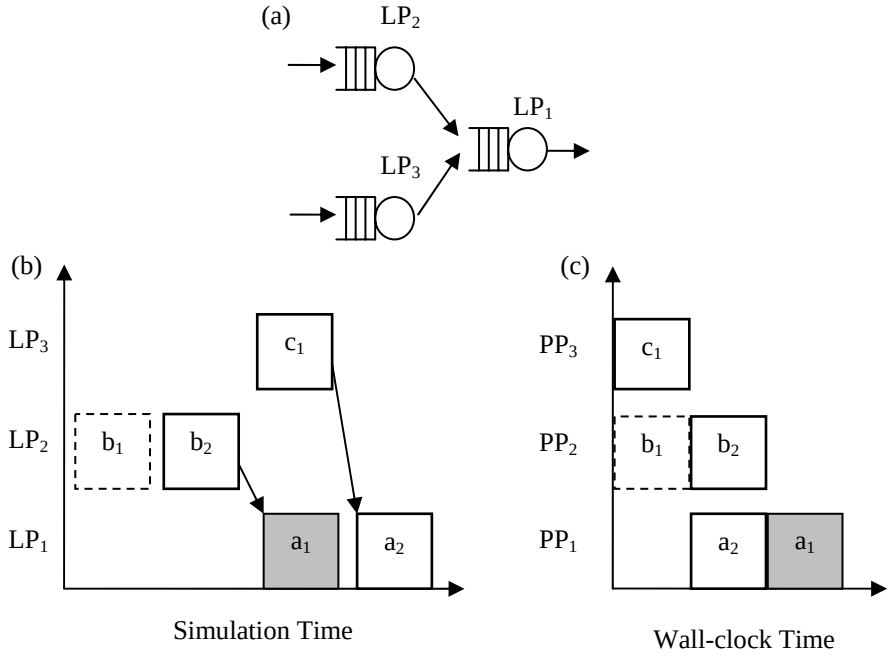


Figure 1.3: Example of Straggler Event

Figure 1.3a shows the topology of three LPs and Figure 1.3b shows a snapshot of their event occurrences. First, event b_1 occurs on LP_2 followed by event b_2 which schedules event a_1 on LP_1. At the same time, event c_1 happens on LP_3 and schedules event a_2 on LP_1. Assuming the three LPs are mapped onto three physical processors (PP_1, PP_2, and PP_3, respectively) and each event requires the same amount of time to execute, Figure 1.3c shows the snapshot of event execution at the three processors. Events b_1 and c_1 are executed concurrently on PP_2 and PP_3, respectively. Then, PP_2 executes event b_2, and at the same time, PP_1 executes event a_2. Finally, PP_2 completes the execution of event b_2 and schedule event a_1 which arrives later at PP_1. Event a_1 is a straggler event because it is executed after event a_2 although it has a smaller simulation time.

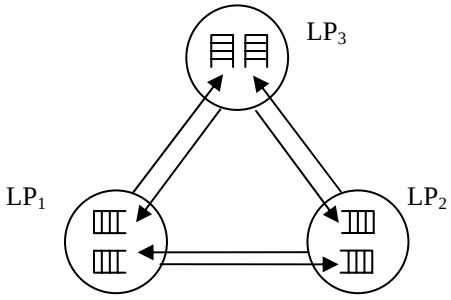


Figure 1.4: LP Structure of CMB Protocol

To avoid the occurrence of straggler events which cause *lcc* violation, Chandy, Misra and Bryant (CMB) proposed building a static communication path for every interacting LP [CHAN79, BRYA84]. A buffer is allocated for every communicating LP. For example, LP_1 in Figure 1.4 allocates two buffers for LP_2 and LP_3 because they may send messages to LP_1. If the communication channel is order-preserving and reliable, it can be proved that to avoid *lcc* violation, every LP has to execute the event with the smallest timestamp in its buffers [CHAN79, BRYA84]. Therefore, an LP must wait until all of its buffers are not empty. This blocking makes the CMB protocol prone to a deadlock where two or more LPs are waiting for each other. The CMB protocol uses a dummy message called *null message* to break the deadlock. The CMB protocol is often referred to as the *null message protocol*. An LP sends a null message with a timestamp *t* to indicate that it will not send any message with a timestamp less than *t*. Null messages are used only for synchronization and do not correspond to real events in the physical system.

Figure 1.5 shows the algorithm of the CMB protocol. The CMB protocol sends null-messages after executing an event (line 6). Each null message has a timestamp equal to its local simulation clock plus a lookahead. The *lookahead* represents the minimal

amount of physical time that is required to complete a process in a physical system. Specifically, at simulation time *t*, a lookahead value of *la* indicates that the sending LP will never transmit any events with a timestamp less than *t*+*la*.

```
1. while (stopping condition has not been met) {
2.   wait until all buffers are not empty
3.   choose event e with the smallest timestamp
4.   simulation_clock = e.timestamp
5.   execute event_handler(e)
6.   send null-message n with
        n.timestamp = simulation_clock + lookahead
7. }
```

Figure 1.5: Algorithm of CMB Protocol Algorithm

Let us consider the example given in Figure 1.4 and assume that the local time at LP_1, LP_2, and LP_3 is 5, 3, and 2, respectively. LP_2 has received an event from LP_3 and is waiting for LP_1 to send its event before LP_2 can proceed. LP_3 is also blocked and waiting for LP_1 to send its event. Meanwhile, LP_1 has received an event from LP_2 with a timestamp of 6 and another from LP_3 with a timestamp of 10. Hence, LP_1 can safely execute the event sent by LP_2 and advance its local time to 6. The situation now is described in Figure 1.6. $Buffer_i$ is the buffer that is used to store the incoming events from LP_i, for example, LP_3 has received an event with a timestamp of 4 from LP_2, but LP_3 has not received any event from LP_1. If the lookahead is 1, after LP_1 executes the event, it will send two null messages with a timestamp equal to 6+1 to LP_2 and LP_3 separately. Now, LP_2 can safely execute an event from LP_3 and LP_3 can safely execute an event from LP_2.

The potential problem with the CMB protocol is the exponential growth of null messages which degrades the time and space performance of the protocol. Some variations of the

CMB protocol that seek to minimize the number of null messages are the demand-driven protocol [BAIN88], flushing protocol [TEO94] and carrier-null message protocol [CAI90, WOOD94].

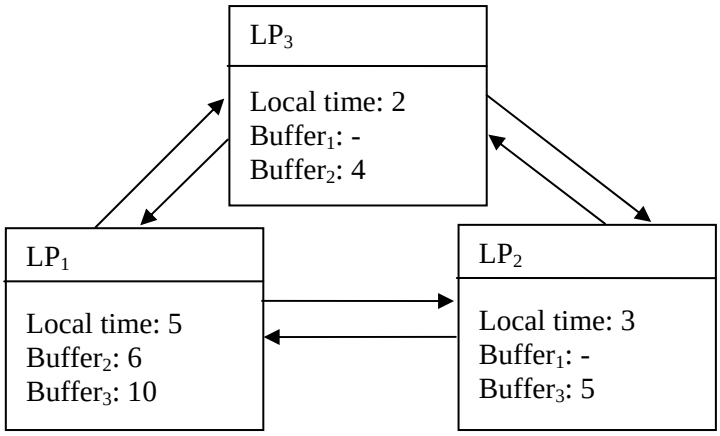


Figure 1.6: Snapshot of a Simulation using CMB Protocol

Bain and Scott proposed the *demand-driven protocol* where LP sends null messages only on demand [BAIN88]. Whenever an LP is about to become blocked, it requests a null message from every LP which has not sent any message to it. This reduces the number of null-messages, but two message transmissions are required to receive a null message. In the *flushing protocol*, when a null message is received, an LP flushes all null messages that have arrived but not been processed [TEO94]. The flushing protocol only sends a null message with the largest timestamp and flushes out the remaining null messages. The flushing mechanism at the input and output channels reduces the number of null messages. The *Carrier null message protocol* attempts to reduce the number of null messages in a physical system with one or more feedback loops [CAI90, WOOD94]. If an LP has sent a null message and later receives this null message back, then it is safe for this LP to execute its event. Therefore, this LP will not forward the null message. The

number of null messages is reduced by this protocol. However, additional information such as routing must be embedded in a null message.

There are other conservative protocols such as the *bounded-lag protocol* [LUBA89, NICO91] and *conservative time window* [AYAN92]. These protocols have a similar structure as shown in Figure 1.7. These protocols start with the identification of safe events, followed by the execution of these events. Barrier synchronization is activated between these two stages so that no LP can progress to the next stage before all LPs have completed their processes. Because of their synchronous behavior, these protocols are more suitable for the shared-memory architecture.

```
1. while (simulation is in progress) {
2.    identify all events that are safe to process
3.    barrier synchronization
4.    process the safe events
5.    barrier synchronization
6. }
```

Figure 1.7: Synchronous Protocol Algorithm

Latest development in the conservative protocols includes the techniques to maximize the exploitation of lookahead [MEYE99, XIAO99], the use of time interval to exploit temporal uncertainty [FUJI99], and the use of causal receive order [ZHOU02].

1.2.3 Optimistic Protocols

Unlike conservative protocols, an optimistic protocol executes events aggressively, i.e., each LP processes every event that is ready for execution. If *lcc* is violated, the

optimistic protocol provides a mechanism to recover from the causality error. The first and most well-known optimistic simulation protocol is *Time Warp* [JEFF85].

Once an LP in the Time Warp (TW) protocol receives a straggler event, it rolls back to the saved state which is consistent with the timestamp of the straggler event and restarts the simulation from that state. The effect of all messages that have been erroneously sent since that state must also be undone by sending special *anti-messages*. When an LP receives an anti-message for an event that has been executed, it has to do another rollback. The protocol guarantees that this rollback chain eventually terminates. To perform a rollback, it is necessary to save the system state and message history. Hence, rollback is computationally expensive and requires a lot of memory. The possibility of the rollback chain worsens the performance of the TW protocol.

Cleary et al. introduced an *incremental state saving* to reduce the memory required for storing simulation state history [CLEA94]. Gafni proposed a *lazy cancellation* technique where anti-messages are not sent immediately, in contrast to immediate cancellation in the original version. The assumption of the lazy cancellation technique is that the re-execution of the simulation will produce the same events, hence these events do not have to be cancelled [GAFN88]. Carothers et al. employed a *reverse execution* instead of a rollback to reconstruct the states of the system [CARO00].

There are some window-based optimistic protocols such as *Moving Time Window*, *Bounded Time Warp*, and *Breathing Time Buckets*. The *Moving Time Window* (MTW) protocol only executes events within a fixed time window [SOKO88, SOKO91]. The MTW protocol resizes the time window when the number of events to be executed falls

below a predefined threshold. The new time window starts from the earliest timestamp of the unprocessed events. This protocol favors a simulation model where every LP has a uniform number of events that falls within the time window. Unfortunately, it is difficult to determine an optimum window size. The earlier version of MTW protocol does not guarantee the correctness of the simulation result [SOKO88]. In the later version, rollback is used to recover from errors caused by *lcc* violation [SOKO91].

Turner and Xu developed the Bounded Time Warp (BTW) protocol which uses a time window to limit the optimistic behavior of the TW protocol [TURN92]. No LP can pass this limit until all LPs have reached it. This approach may reduce the number of rollbacks and the possibility of rollback thrashing.

The *Breathing Time Buckets* protocol uses two time windows, i.e., the local event horizon and global event horizon [STEI92]. It maps several LPs onto a processor. Any LP on one processor is allowed to execute events with a timestamp less than the LP's local event horizon, but it is not allowed to send messages to LPs on other processors. The global event horizon is calculated after all processors have reached their local event horizon. Then, events with a timestamp less than the global event horizon can be sent to other LPs on other processors.

Recently, researchers introduced a number of new techniques to improve the performance of optimistic protocols, such as the use of reverse computation to replace the rollback process [CARO00], direct cancellation to reduce overly optimistic execution [ZHAN01], and the concept of lookback that avoids anti-messages [CHEN03].

1.3 Approaches in Simulation Performance Analysis

We have shown in the previous section that many simulation protocols and their variations have been proposed. Therefore, a decision must be made to determine which protocol should be used if we want to simulate a certain physical system. Ideally, this decision should be made prior to implementation. This section reviews different approaches to evaluating simulation performance. We categorize these approaches based on Jain's classification, i.e., measurement, analytical, and simulation [JAIN91].

1.3.1 Measurement Approach

The *measurement approach* evaluates the performance of a simulation protocol through direct observation by using instrumentation. The measurement approach can be used only after the simulation protocol has been implemented. As shown in the following examples, the measurement approach is usually used to measure speed-up, null-message ratio, execution time etc.

The first conservative protocol was proposed in the absence of parallel computer technology [CHAN79]. With the advent of parallel computers, some measurements were carried out to analyze the performance of parallel simulation protocols. Following this, new protocols and variations have been proposed to improve their performance. In the discussion that follows, we review some performance evaluation studies of the conservative protocol, followed by the optimistic protocol. Finally, the performance comparison between conservative and optimistic protocols is presented.

Conservative Protocol

Teo and Tay demonstrated that the flushing protocol reduces the exponential number of null messages in the original CMB protocol to linear [TEO94]. They implemented a multistage interconnection network simulation and reported a speed-up of 7 on eight processors.

Cai and Turner showed that carrier null messages reduce the number of null messages and increase simulation performance [CAI90]. They reported that the carrier null message protocol yields better performance than the CMB protocol for queuing networks which contain at least one feedback loop. Later, Wood, and Turner modified the carrier null message protocol to cover the arbitrary feedback structure [WOOD94]. The measurement shows that the number of null messages is reduced by a reasonable amount for small numbers of processors. However, for larger numbers of processors, the reduction is trivial. The measurement also reveals that the overhead cost of the carrier null message protocol is considerably high, to an extent that it cannot achieve a better speed-up than the CMB protocol.

Xu and Moon showed that for homogenous applications which have a very high granularity, a simple synchronous protocol can achieve a good speed-up [XU01]. They benchmarked several VHDL circuits and reported a speed-up between 6 and 10 on 16 processors.

Ayani and Rajaei measured the performance of the CTW protocol using two benchmarks: feed-forward networks and networks with feedback [AYAN92]. The result shows that the CTW protocol produces a good speed-up for a simulation with a

symmetric workload and large granularity (around 10 for feed-forward networks and 7 for networks with feedback on 15 processors). The measurement demonstrates that the CTW protocol performs poorly with a heterogeneous application with a small problem size.

An empirical study of the CMB protocol's overhead costs was carried out by Bailey and Pagels by estimating the number of null messages and the null message cost [BAIL94]. They found that network characteristics affect the number of null messages. They also showed that the time spent in sending null messages is not a simple constant, but depends on the mapping between LPs and processors and the average number of neighbors.

Song et al. studied the effect of different scheduling algorithms on the number of null messages and speed-up [SONG00]. They showed that scheduling based on the earliest output time can improve CMB protocol performance. The earliest output time is the earliest time at which an LP might send a message to any other LP. Later, Song measured the blocking time overhead of the CMB protocol [SONG01].

Recently, Park et al. compared the performance of synchronous and asynchronous algorithms for conservative parallel simulation running on a cluster of 512 processors [PARK04]. They noted that the scale of previous studies have been limited to modest sized configurations (using fewer than 100 processors). Further, they noted that the conclusions based on small-scale simulation studies may not apply to a large-scale simulation.

Optimistic Protocol

Jefferson et al. completed the Time Warp Operating System (TWOS) which includes the implementation of the TW protocol [JEFF87]. They reported a speed-up of 8 on 32 processors using a military application.

Cleary et al. compared the performance of copy state saving (original TW protocol) and incremental state saving [CLEA94]. The results suggest that incremental state saving improves the performance of the TW protocol for at least one real application. Since then, many extensive works on improving the state saving mechanism have been reported [RONN94, SKOL96, WEST96, FRAN97, QUAG99, SOLI99]. In general, these works can be classified into three categories: incremental state saving, sparse state saving, and hybrid state saving.

Three years after proposing the MTW protocol, Sokol et al. reported a successful implementation on Sequent Symmetry shared-memory [SOKO91]. They reported a reduction in the execution time with an increase in the number of processors. An increase from two to seven processors decreased execution time from 240ms to 100ms. However, no figure on the speed-up was reported.

The performance comparison between the BTB protocol and TW protocol was reported in [STEI93]. The hypercube model was chosen as a benchmark. The results showed that the BTB protocol performed better than the TW protocol.

Li and Tropper studied the performance of TW protocol with event reconstruction technique [LI04]. They showed that event reconstruction improves the performance of

TW protocol. Chen and Zymanski proposed a technique called lookback to reduce the number of rollback in TW protocol [CHEN03]. They studied the performance of four types of lookback using mobile communication system simulation. Zeng et al. proposed a new batch-based cancellation scheme and compared its performance with the conventional per-event based cancellation scheme in TW protocol [ZENG04].

Conservative versus Optimistic Protocol

Measurement techniques have been used to compare the performance of a conservative protocol and an optimistic protocol, specifically between the CMB protocol and the TW protocol. Fujimoto measured the performance of the CMB protocol and the TW protocol simulating closed queuing networks on the shared-memory architecture [FUJI89]. He found that in most cases, the TW protocol outperformed the CMB protocol. On the other hand, Preiss reported that the CMB protocol was better than the TW protocol [PREI90]. Recently, Unger et al. compared the performance of the CMB protocol and the TW protocol on a shared-memory architecture simulating a cell-level ATM (Asynchronous Transfer Mode) network simulator [UNGE01]. They concluded that the relative performance between the two protocols depended on the size of the ATM network, the number of traffic sources, and the traffic source types.

There are many other works on the performance of PADS using the measurement approach [FUJI00]. In general, the measurement approach has been used for two main reasons, i.e., to evaluate the performance of new protocols and to compare the performance of the existing protocols, particularly the relative performance between the conservative and optimistic protocols.

The measurement approach can only be done after implementation. Moreover, comparing the performance of PADS with many different combinations of physical systems, protocols, and execution platforms by measurement is intractable because of the scalability problem. We have shown that most measurements compare a limited number of protocols using a limited number of benchmarks which cannot be generalized [STEI93, CLEA94, XU01, UNGE01].

1.3.2 Analytical Approach

There have been a number of publications on simulation performance evaluation using the analytical approach. In general, the objectives are to predict the performance of a protocol, to study the potential and limitation of a protocol, and to compare the performance of different protocols.

Lavenberg et al. proposed an analytical model to estimate the speed-up of the TW protocol running on two processors [LAVE83]. The focus of the study is on a *self-initiating system* where an event on one LP does not schedule events to other LPs. The model is valid only when the interaction between two processors is small. Felderman and Kleinrock [FELD91] developed another quantitative model to improve the model in [LAVE83]. The new model is valid for arbitrary P number of processors. The model assumes that the event execution time follows an exponential distribution function with a mean of one. After the event is executed, the processor will advance its local clock by one unit. Further, it assumes that each processor always sends K events to K processors that are uniformly chosen from the other P–1 processors ($K < P$). The upper and lower bounds on speedup are estimated as a function of P and K.

The performance analysis of the TW protocol on multiple homogeneous processors was proposed by Gupta et al. [GUPT91]. They chose a *message-initiating system* called PHOLD as a benchmark. Unlike the self-initiating system, in the message-initiating system, an event in one LP may schedule events to other LPs. They used a Markov chain model to estimate some performance metrics including speed-up. The assumptions in the model are: the event execution time on each processor (*event execution time* in short) is exponentially distributed; the time advancement in the simulation model (*time increment*) follows an exponential distribution; the synchronization cost is negligible; and each LP is mapped onto one processor.

Nicol developed a model for the TW protocol running on multiple homogeneous processors implementing a self-initiating system [NICO91]. He provided an upper bound and a lower bound on speed-up. Felderman and Kleinrock also proposed another model for the same system [FELD91]. The computed upper bound and lower bound are different from Nicol's model because of the difference in their assumptions. Nicol assumed a deterministic event execution time and the time increment in the model is random. On the other hand, Felderman and Kleinrock assumed a random event execution time and deterministic time increment in the model. The contrast between the two models shows that different assumptions in analytical models targeting the same system may lead to different results.

The performance of the TW protocol on multiple homogeneous processors with a limited memory was proposed by Akyildiz et al. [AKYI93]. They approximated speed-up as a function of memory capacity using a Markov chain model. The assumptions are similar

to those in [GUPT91] with one additional assumption, i.e., there is always at least one event in each LP's buffer. Steinman provided an analytical model to estimate the number of events that can be processed in a cycle. This number can be used to estimate the parallelism in the Breathing Time Bucket protocol [STEI94]. The model assumes that each event generates only one event, hence the number of events in the system is constant. The event generation is assumed to follow a beta distribution.

Nicol used a stochastic model to estimate the overhead costs of his bounded-lag protocol [NICO93]. The overhead costs include event list manipulation, lookahead computation, synchronization and the processor's idle time. These overhead costs are used to approximate the utilization of each processor.

Xu and Moon developed a performance model to estimate the speed-up of a specific VHDL application using a simple synchronous protocol [XU01]. They assumed that each LP requires a unit time to execute an event, and each LP has equal probability of being active (i.e., has some events to execute). Song developed a probabilistic model to estimate the blocking time overhead of a CMB protocol variation for an arbitrary number of LPs [SONG01]. The model is verified using a closed queuing network and a mobile communication system as benchmarks. Nicol developed a model for the composite synchronization [NICO02], which combines localized asynchronous coordination with a global synchronization window, to predict the theoretical achievable speed-up.

Apart from understanding the potential and limit of a protocol, the analytical approach has been used to compare the performance of different protocols, particularly between conservative and optimistic protocols. Lin and Lazowska developed a model comparing

the TW and CMB protocols. Rollback and state-saving costs are assumed to be zero in the model. It shows that as long as a correct computation is never rollbacked by an incorrect computation, the TW protocol always performs at least as well as the CMB protocol [LIN90].

Lipton and Mizel conducted a worst case analysis of the TW and CMB protocols. Assuming the state of an LP is saved after every event execution but not considering state saving costs, they showed that there exists a simulation such that the TW protocol can arbitrarily outperform the CMB protocol. They proved that the converse is not true. However, they showed that the performance of the TW protocol can be worse depending on the assumption on the rollback cost [LIPT90].

The analytical approach can be applied prior to implementation. However, to make a model tractable, simplifying assumptions are made that often result in loss of accuracy. Further, analytical approaches model a system as a function (or a black box) that maps a set of input onto a set of output. Therefore, it may not be suitable to model the interaction among events in a simulation system. However, analytical approaches are useful in deriving the theoretical simulation performance bounds [XU01, SONG01, NICO02].

1.3.3 Simulation Approach

Ferscha et al. stated that the analytical approach fails to achieve satisfactory accuracy due to: (i) unrealistic and inadequate assumptions in the model, (ii) the complexity of the detailed models, (iii) simplifying assumptions that make the evaluation of those models tractable, and (iv) the possibility of modeling errors [FERS97]. They proposed a

performance analysis based on simulation, i.e., to use the simulation approach to study the performance of a simulation protocol. In fact, when Chandy and Misra proposed the first PADS protocol, they had to evaluate its performance using simulation since parallel computers were not available at that time [CHAN79]. Hence, the use of simulation to evaluate the performance of a simulation protocol dates back to the early development of the PADS protocol.

Ferscha et al. implemented a simulator which combines protocol and execution platform variations into a single skeleton program called N-MAP, shown in Figure 1.8 [FERS97]. The *simulation model attributes* represent the characteristics of a simulation model. A 32-node Torus serves as the benchmark; the size of the message density and the service time distribution function are varied. The *strategy attributes* represent different protocol variations. The CMB protocol is characterized according to two attributes: GVT reduction (with/without) and null-message sending initiation (sender/receiver). Hence, it considers four CMB protocol variations. The TW protocol is characterized by the cancellation policy (aggressive/lazy), state saving policy (incremental/full), and optimism control (with/without). Thus, eight TW protocol variations are considered. The *platform attributes* represent the execution platform; in the experiment the researchers considered only the number of processors. Keeping to the various parameters, the code simulating parallel simulation execution is run to predict system performance.

There are two main drawbacks to this approach. First, the number of simulation protocols and their variations are huge. Therefore, it is virtually impossible to implement a simulator that combines all of them into a single skeleton program. Second, it is

difficult to include major platform attributes such as the processor's speed and communication delay in the simulator.

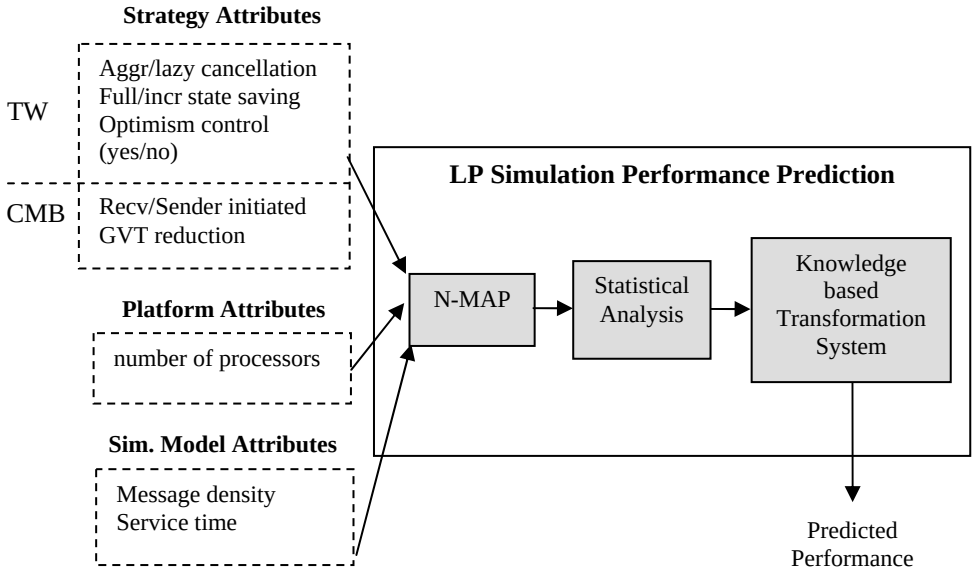


Figure 1.8: Ferscha's Simulation Performance Analysis [FERS97]

Another commonly used simulation-based approach is *critical path analysis* (CPA). Berry et al. first introduced CPA to analyze the performance of parallel simulation [BERR85]. The critical path is derived from running a sequential simulation. The term CPA is borrowed from Project Management and has been suggested as a technique to establish a theoretical lower bound on PADS execution time. Based on CPA, Lin developed a program to analyze the parallelism in parallel simulation [LIN92]. Using the same methodology, Wong and Hwang developed an algorithm to predict the space requirement of the CMB protocol by measuring the event population list [WONG95].

The CPA method assumes that events are executed based on a causal effect relationship. An event which spawns another event must be executed before the spawned event, and events that happen in the same LP must be executed in timestamp order. Further, CPA

assumes that each logical process is mapped onto one processor. Once the critical path has been established, the average event execution time is used to estimate the simulation execution time. Communication time is assumed to be negligible. These assumptions suggest that CPA is protocol independent, i.e., the result will depend only on the physical system and the average event execution time. Therefore, CPA cannot be used to compare the performance of different simulation protocols.

1.4 Objective of Research

Researchers have lamented that the lack of an adequate performance evaluation framework is one of the major obstacles hindering the widespread adoption of PADS [LIN93, LIU99, TEO99]. A number of frameworks have been proposed [BARR95, JHA96, FERS97, TEO99, LIU99, SONG01]. However, they focus on either certain simulator only (e.g., TW protocol, CMB protocol) or on certain aspect of performance study (e.g., benchmark, workload characterization).

The main objective of this research is to propose a framework for simulation performance analysis that provides an understanding of performance ranging from simulation problem, to simulation models, and simulation implementations. We focus on time and space performance of a simulation. The time and space performance is characterized in three different layers, i.e., physical system, simulation model, and simulator. Our framework encompasses the *formalization* of simulation event ordering and the *characterization* of simulation performance.

Simulation event ordering is used as the unifying concept for performance analysis at different layers. We propose simulation event ordering as the unifying concept because it exists at the three layers. Simulation event ordering is formalized based on a partially ordered set (poset). If events with the same physical time are grouped as a set, there is only one event order in a physical system. However, different event orders can be used at the simulation model layer. A simulator employs synchronization algorithm (protocol) to maintain its event order at runtime. The same event order at the simulation model layer can be implemented using different protocols at the simulator layer. We identify and formalize the event orders of a number of simulation protocols such as CMB [CHAN79] and Time Warp [JEFF85]. In addition to providing a unifying concept for performance analysis, the distinction between simulation event ordering from the synchronization algorithm is crucial as it means that simulation event ordering can now be used to study the performance of a simulation protocol ahead of its implementation.

Our performance evaluation framework characterizes the time and space performance in three layers. At the physical system layer, the analysis focuses on comparing different physical systems (for example, comparing their inherent parallelism). At the simulation model layer, we analyze the time and space performance of different event orders used in the simulation. At the simulator layer, we analyze the time and space performance of a simulator. To compare the time performance across layers, we need to normalize event parallelism because each layer uses different time unit (i.e., physical time unit, timestep, and wall-clock time unit). Next, we propose a relation called *stricter* and a measure called *strictness* for comparing and quantifying the degree of event dependency of simulation event orderings, respectively. *Stricter* and *strictness* are time independent so that the strictness of event orderings at different layers can be compared directly.

1.5 Thesis Overview

The rest of the thesis is organized as follows.

We formalize simulation event ordering based on the partially ordered set (poset) in Chapter 2. Poset is a branch of discrete mathematics which studies how elements of a given set are ordered. This chapter starts with the motivation of formalizing simulation event ordering. This is followed by an introduction to poset. Lastly, we define simulation event ordering and formalize a number of simulation event orderings.

We propose our time (event parallelism) and space (memory requirement) performance characterization in Chapter 3. The time and space performance of a simulation is characterized into three layers, i.e., physical system, simulation model, and simulator. Next, we discuss event parallelism normalization and total memory requirement. Lastly, we propose to compare and measure event dependency using a relation called *stricter* and a measure called *strictness*, respectively.

We discuss the application of our framework in Chapter 4. First, measurement tools to measure the time and space performance at the three layers is presented. Next, we validate and demonstrate the proposed framework using an *open* and a *closed* systems. Next, we apply the framework to study the performance of Ethernet simulation.

Finally, Chapter 5 summarizes the results of this thesis and discusses some issues that require further investigation.

Chapter 2

Formalization of Simulation Event Orderings

In a physical system, a set of events occur in a chronological (physical) time order. A simulator can execute these events using different orderings as long as the simulation result is correct. The advance of PADS has increased the number of possible event orderings. As will be shown later, event ordering affects simulation performance. Hence, simulation event ordering is an important concept in simulation. However, there is a lack of formal analysis of simulation event ordering. The most comprehensive work was done by Fujimoto and Weatherly [FUJI96, FUJI00]. They studied different message orderings to reduce or eliminate temporal anomalies in the Time Management service of the High Level Architecture, a standard for distributed simulation interoperability. Recently, Zhou et al. investigated the causality issue in distributed simulation and proposed the causal receive ordering [ZHOU02].

In this chapter, we propose the formalization of simulation event ordering based on the partially ordered set (poset). We start with some relevant works on the formalization of event orderings in distributed simulation which have motivated our proposed formalization. This is followed by a review on poset and some definitions that are

relevant to simulation event ordering. Lastly, we define what simulation event ordering is, and formalize a number of major simulation event orderings.

2.1 Motivation

The need for formalization of simulation event ordering is motivated by research in memory operation orderings in memory consistency model [CULL99, GHAR95] and message ordering in broadcast communication services [HADZ93, ATTI98].

A memory consistency model specifies the ordering rules in which memory operations must be executed. Lamport first introduced the sequential consistency (SC) model [LAMP79]. However, this model is too restrictive as it does not allow compilers or processors to do much optimization to exploit parallelism. More relaxed models were later proposed, such as weak ordering [DUBO90], release consistency [GHAR90], processor consistency [GHAR90], and relaxed memory ordering [WEAV94]. A more complete list of memory consistency models can be found in [GHAR95, CULL99]. On the implementation side, Shasha and Snir proposed a method based on program-specific information to implement the SC model [SHAS88]. Subsequently, Afek et al. proposed a more efficient method called lazy caching to implement the SC model [AFEK89]. Later, Landin et al. proposed an SC implementation for some network topologies [LAND91]. Another promising implementation based on the speculative read and write prefetching technique was proposed by Gharachorloo [GHAR91]. The implementation of other memory consistency models can be found in [GHAR95, CULL99].

Broadcast communication service specifications recognize several event orderings such as: FIFO order, causal order, total order, FIFO atomic order, and causal atomic order [HADZ93, ATTI98]. Hadzilacos and Toueg noted the necessity of uniform notation to understand the close relationship among broadcast event orderings [HADZ93]. Meanwhile, many algorithms have been proposed to implement different broadcast event orderings. For example, causal order was first implemented in ISIS [BIRM87]. Many other implementation strategies for causal order have since been proposed [SCHI89, RAYN91, SCHW94, GAMB00].

Research in the memory consistency model and broadcast communication services separate the specification of event orderings from their implementation strategies [CULL99, GHAR95, HADZ93, ATTI98]. Here, we note two major benefits that may be derived from the separation. First, we can organize different event orderings in a uniform and coherent way. Second, it is possible to evaluate different event orderings independently from the implementation factors. These considerations motivate us to separate the specification of simulation event ordering in simulation model layer from its implementation in the simulator layer.

2.2 Overview of the Partially Ordered Set

Simulation event ordering is formalized using a notation that is commonly used in the partially ordered set (poset). Poset theory forms a branch of discrete mathematics which studies how elements of a given set are ordered. The ordering of the elements of a set permeates our daily life. In general, set ordering is *transitive*, e.g., if $1<2$ and $2<3$ then $1<3$. Set ordering is also *anti-symmetric*, e.g., if 1 is less than 2 then 2 is not less than 1.

These properties imply that set ordering is *anti-reflexive*, i.e., 1 is not less than itself. In discrete mathematics, this set ordering is commonly called *partial order* [ROSE99].

2.2.1 Partial Order and Partially Ordered Set

Dushnik and Miller introduced the notion of partial order in 1941 [DUSH41]. This classical paper has played an important role in shaping the direction of research in combinatorics and set theory. Definition 2.1 gives the formal definition of partial order. This definition is called the *strong inclusion definition* [NEGG98]. There is another type of definition called *weak inclusion definition* [NEGG98]. A weak inclusion definition relaxes the anti-symmetric property of partial order. As will be shown later, simulation event ordering adopts the strong inclusion definition, and hence, this thesis concentrates on it.

Definition 2.1. *An order R over S (where S is a set) is called a* ***partial order*** *if:*

1. $\forall x \in S \bullet (x, x) \notin R$ *(anti-reflexive)*
2. $(x, y) \in R$ *then* $(y, x) \notin R$*, and vice versa (anti-symmetric)*
3. $(x, y) \in R$ *and* $(y, z) \in R$ *then* $(x, z) \in R$ *(transitive).*

It is common to represent an order R as a set of pairs where a pair $(x, y) \in R$ denotes that x is ordered before y in R [NEGG98]. Thus, for a given set $S = \{1, 2, 3\}$, an order LT (i.e., less than) is a set of $\{(1,2), (2,3), (1,3)\}$. This shows that LT is anti-reflexive, anti-symmetric, and transitive; hence, it is a partial order. An order "descendant of" a given set of people is another example of partial order. However, an order "friend of" a given set of people may not be a partial order, depending on the given set of people. This leads

to the concept of a *partially ordered set*. Dushnik and Miller's definition of a partially ordered set is given in Definition 2.2 [DUSH41].

Definition 2.2 *A* ***partially ordered set (poset)*** *is a tuple* (S, R) *where S is a set and R is a partial order on the set S.*

Poset combines a partial order R with the universe (i.e., S) on which it operates. Hence, the same two partial orders operating on two different universes are two different posets. Similarly, two different partial orders operating on the same universe are two different posets.

For a given poset (S, R), two distinct elements x and y of S are considered *comparable* if either $(x, y) \in R$ or $(y, x) \in R$, and *incomparable* (or *concurrent*) otherwise. Every two distinct elements of S in a poset (S, R) are either comparable or incomparable. If two elements are comparable, it implies that we can order them. Therefore, it is possible that an element is ordered immediately after another element. This relation is called *cover*.

A poset (S, R) can be represented as a directed acyclic graph (V, E) where V is a set of vertices and E is a set of edges. The most commonly used graph is the *Hasse diagram* [TROT92, NEGG98]. A vertex $v \in V$ in the Hasse diagram represents a unique element $s \in S$ in the poset (hence $V=S$). An edge (x, y) exists in the diagram if and only if y covers x in the poset. Normally, y is situated higher than x in the Hasse diagram, but in this thesis, we prefer to put y to the right of x instead. For example, Figure 2.1 shows a Hasse diagram representing poset (S, R) where $S = \{p, q, r, s, t\}$ and $R = \{(p, q), (q, r), (p, r), (p, s), (s, t), (p, t)\}$. In this poset, p and q are comparable. However, q and s are

incomparable. Further, *r* covers *q* and *q* covers *p*, but *r* does not cover *p* (although *p* and *r* are comparable). In this thesis, we use Hasse diagram to represent a simulation event ordering where each vertex represents an event and an arrow from event *x* to event *y* denotes that event *x* must be ordered before event *y*.

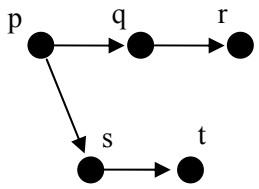


Figure 2.1: Hasse Diagram

Dilworth's chain covering theorem shows that a poset is formed by one or more disjoint chains using a set inclusion [TROT92]. A chain is a set where all of its elements are comparable. The longest chain among them is called the maximum chain and its length is the height of the poset. The dual version of Dilworth's chain covering theorem shows that a poset is also formed by one or more disjoint anti-chains. Anti-chain is a set where all of its elements are incomparable. The Dilworth's chain covering theorem and its dual are shown in Definition 2.3 and Definition 2.4, respectively.

Definition 2.3. *For a given poset* (*S, R*)*, there exists a set of posets* (S_i, R_i) *such that* $\forall_i$ $S_i \subseteq S$ *and* R_i *is a chain.* R_i *is the* ***maximum chain*** *if there is no other* R_j *where* $j \neq i$ *such that* $\|S_j\| > \|S_i\|$. *The size of the maximum chain* (*i.e.,* $\|S_i\|$) *is the* ***height*** *of the poset* (*S, R*).

Definition 2.4. *For a given poset* (*S, R*)*, there exists a set of posets* (S_i, R_i) *such that* $\forall_i$ $Si \subseteq S$ *and* R_i *is an anti-chain.* R_i *is the maximum anti-chain if there is no other* R_j *where*

$j \neq i$ such that $\|S_j\| > \|S_i\|$. The size of the ***maximum anti-chain*** *(i.e., $\|S_i\|$) is the* ***width*** *of the poset* (*S, R*).

Figure 2.2a shows the two chains that form the poset given in Figure 2.1. The maximum chain is {*p*, *q*, *r*} and therefore the *height* of the poset is three. Similarly, the anti-chains that form the poset are given in Figure 2.2b. The maximum anti-chain is {*s*, *q*} or {*r*, *t*} and the width of the poset is two. Dilworth's chain covering theorem and its dual are used to relate the degree of event dependency and parallelism in Chapter 3.

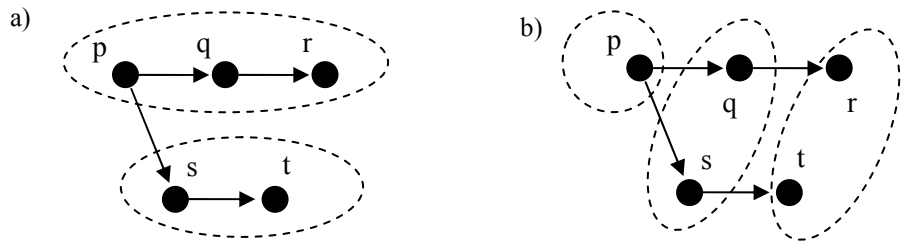


Figure 2.2: Dilworth's Chain Covering Theorem and Its Dual

2.2.2 Total Order and Interval Order

Researchers in poset have proposed several orders such as total order [DUSH41], interval order [FISH88], circle order [FISH88], angle order [FISH89], tolerance order [BOGA95], split semi-order [FISH99], etc. The special journal *Order* published by Kluwer is devoted to the theory of order and its applications (see http://www.kluweronline.com). We will discuss total order and interval order as they are relevant to simulation event ordering. Their definitions are given in Definition 2.5 and 2.6, respectively.

Definition 2.5. *A relation R over S is called a* ***total order*** *if there exists a function f for all* $x \in S$ *such that every x is mapped onto a unique* $n \in N$*, where* N *is the set of natural numbers. Hence, for all* $x,y \in S$*, x is ordered before y if and only if* $f(x) < f(y)$.

Every element of S in total order is mapped onto a unique natural number. Therefore, as the name implies, the elements of S can be arranged totally such that every two distinct elements are comparable. For example, given $S = \{a, b, c\}$ and $R = \{(a, b), (b, c), (a, c)\}$, the order R is a total order. An order "less than" on a set of integers is another example of total order because every integer is comparable to other integers.

Definition 2.6. *Let a function I be assigned to each* $x \in S$ *such that* $I(x) = [start(x), end(x)]$*, where* $\forall x \in S \bullet start(x) \leq end(x)$**.** *A relation R over S is called an* ***interval order*** *if* $\forall x,y \in S \bullet (x, y) \in R \leftrightarrow end(x) < start(y)$

Interval order assigns an interval I to each of its elements. Two distinct elements x and y are comparable if their intervals do not intersect, i.e., $I(x) \cap I(y) = \varnothing$. There is a special case where the interval length, i.e., $end(x) - start(x)$, is a constant for all $x \in S$. This order is called ***semi-order*** [PIRL97]. Interval order models inexact measurement, for examples: task scheduling where each task completion time is uncertain, or the time spans over which animal species are found in archaeological strata, etc.

2.3 Definition of Simulation Event Orderings

Based on poset, we formalize simulation event ordering in Definition 2.7 [TEO01, TEO04]. Just as a poset has two components (Definition 2.2), a simulation event

ordering (referred to as event ordering in short) also comprises two main components: a set of events and an event order.

Definition 2.7. *A **simulation event ordering** is a tuple* (E, S_R) *where E is **a set of events** and* S_R *is a set of comparable events based on **simulation event order** R.*

The set of comparable events S_R is represented as a set of pairs where a pair $(x, y) \in S_R$ denotes event x is ordered before event y in event order R. In simulation, we never say that an event x is ordered before itself (i.e., $(x, x) \notin S_R$). This shows that the event ordering uses a strong inclusion definition of poset because a weak inclusion definition imposes that $(x, x) \in S_R$ must hold [NEGG98]. Therefore, an event order R must be anti-reflexive, anti-symmetric, and transitive as in the strong inclusion definition of poset.

Definition 2.8. *An event order R is:*

1. $\forall x \in E \bullet (x, x) \notin S_R$ *(anti-reflexive)*
2. $(x, y) \in S_R$ *then* $(y, x) \notin S_R$*, and vice versa (anti-symmetric)*
3. $(x, y) \in S_R$ *and* $(y, z) \in S_R$ *then* $(x, z) \in S_R$ *(transitive).*

2.3.1 Physical System

Simulation can adopt different event orderings to simulate a physical system provided that the simulation result is the same as if we simulate it using the event ordering of the physical system. Before we discuss the event ordering in the physical system, it is important to introduce the concept of *causal dependency* [FUJI00]. The causal dependency among events is based on the relation *happened before* [LAMP78]. An

event is dependent on another event if they happen at the same service center (i.e., access the same resource) and one of them happens before the other. An event is also dependent on other event, if one of them triggers the other. The two conditions are reflected in the definition of predecessor and antecedent given below [FUJI99].

Definition 2.9. *Let x be an event and x.t the physical time when event x happens. Event x is the* ***predecessor*** *of y* (*denoted by y.pred = x*), *if x and y occur at the same service center with x.t < y.t and there is no other event z that is also at the same service center such that x.t < z.t < y.t.*

Definition 2.10. *Let x be an event. Event x is the* ***antecedent*** *of y* (*denoted by y.ante = x*), *if x spawns y.*

Figure 2.3a shows a physical system with four service centers, S_1, S_2, S_3, and S_4. Figure 2.3b shows the corresponding snapshot of event occurrences. Horizontal axis represents physical time and vertical axis represents service centers. Label a_i^t represents the i^{th} arrival event and d_i^t represents the corresponding departure at time t. A shaded circle represents an event arrival and unshaded one represents an event departure. The snapshot shows that at time 0, job 1 arrives at S_1. Since, S_1 is idle, job 1 is processed until time 4. Job 2 arrives at S_1 at time 2. Since S_1 is busy, this job must wait until S_1 completes job 1, and so on. A dashed arrow from x to y shows that x is the predecessor of y; and a solid arrow from x to y shows that x is the antecedent of y. These arrows show the causal relationship among events in the physical system. The causal relation is transitive, i.e., if event x causally affects event y, and event y causally affects event z, then event x also causally affects event z.

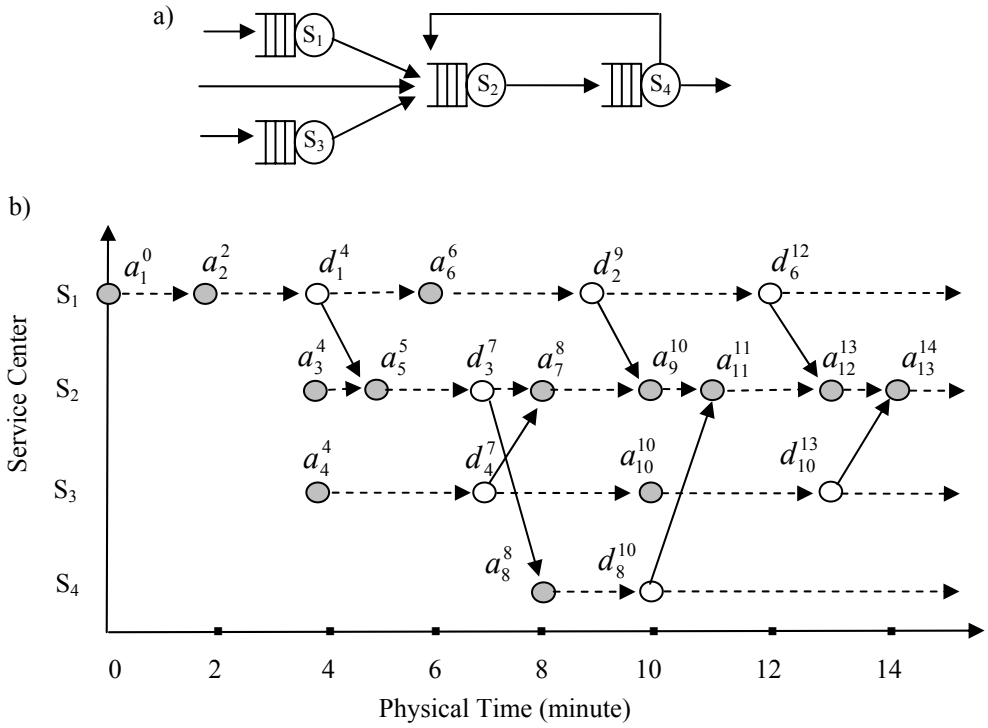


Figure 2.3: Causal Dependency – Physical System

An event order in the physical system corresponds to how events in the physical system are ordered. Based on the physical time, there is only one event order for any physical system, i.e., an event with a smaller physical time is ordered before an event with a larger physical time (Definition 2.11). The causal dependency among events in the physical system is reflected in the physical time when the events occur. If event *x* causally affects event *y* then event *x* happens at an earlier physical time than event *y*, but the converse may not be true.

Definition 2.11. *Let x be an event in a physical system and x.t the physical time when event x happens. The event order in any physical system dictates that for all x and y (where* $x \neq y$*), x is ordered before y if and only if* $x.t < y.t$.

Figure 2.4 shows the Hasse Diagram of the event ordering in the physical system for the set of events given in Figure 2.3. The arrow from event *x* to event *y* in a Hasse Diagram indicates that event *x* must be ordered before event *y*. Since an event order is transitive, the arrows can also be traversed transitively. For example, if event *x* must be ordered before event *y* and event *y* must be ordered before event *z*, then event *x* must also be ordered before event *z*.

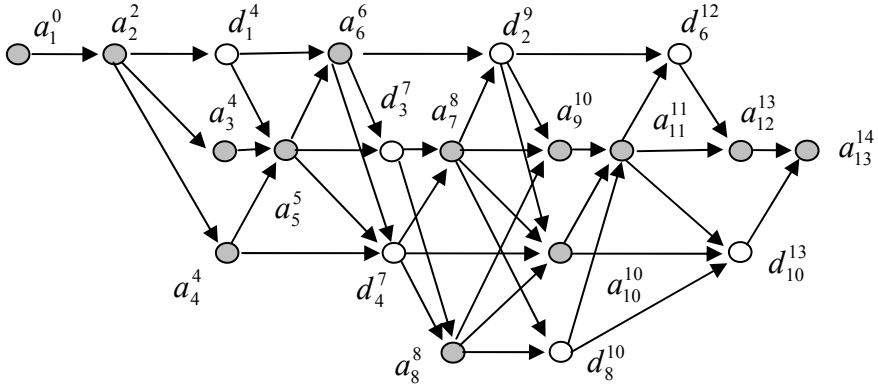


Figure 2.4: Hasse Diagram – Physical System

2.3.2 Simulation Model

In the Virtual Time simulation modeling paradigm [JEFF85], a simulation model emulates a physical system and the interaction among physical processes in the physical system (see Figure 2.5). Each physical process in the physical system is mapped onto a logical process (LP) in the simulation model. Each event in the simulation model represents an event in the physical system. The simulation time of an event in the simulation model represents the physical time of the corresponding event in the physical system. The event ordering in a physical system can be modeled and simulated using various event orderings to exploit different degrees of event parallelism.

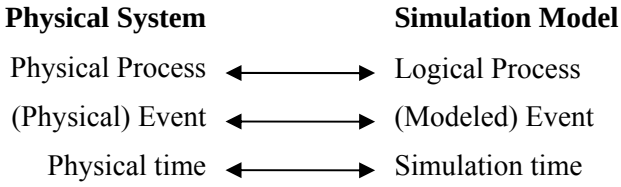


Figure 2.5: Mapping between Physical System and Simulation Model

Lamport defined *happened before partial order* and *total order* [LAMP78]. He proved that both orders are anti-reflexive, anti-symmetric, and transitive which match our definition of simulation event order (Definition 2.8). Hence, we refer to these event orders as partial event order and total event order, respectively. The definition of partial event order is given in Definition 2.12. Figure 2.6 shows the Hasse Diagram of the partial event ordering (E, $S_{partial}$) for the set of events E in Figure 2.4.

Definition 2.12. *Event x is ordered before event y in* ***partial event order*** *if y.pred = x or y.ante = x.*

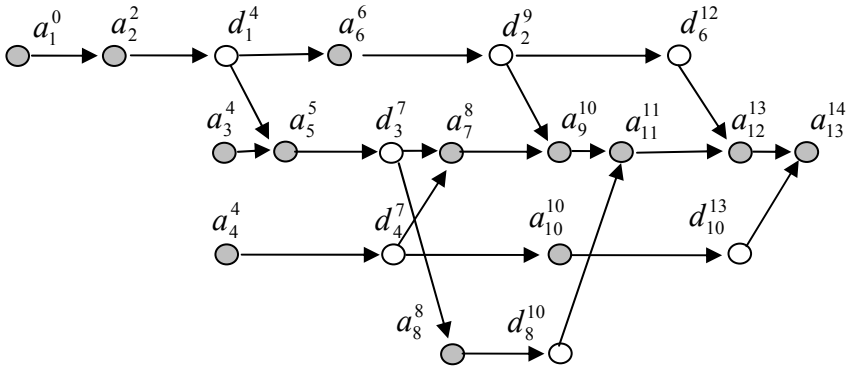


Figure 2.6: Hasse Diagram – Partial Event Ordering

The definition of a total event order is given in Definition 2.13. The priority function in total event order is used to decide which event should be processed when two or more

events have the same timestamp. For example, two events have the same timestamp, the event with higher priority will be executed first. Figure 2.7 shows the Hasse Diagram of the total event ordering (E, S_{total}) for the set of events E in Figure 2.4.

Definition 2.13. *Event x is ordered before event y in* ***total event order*** *if and only if:*

1. *x.timestamp < y.timestamp, or*
2. *x.timestamp = y.timestamp and priority*(x) < *priority*(y).

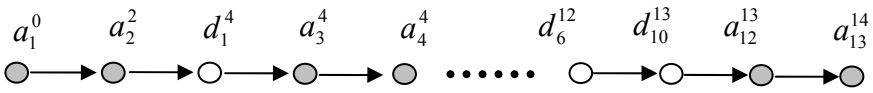


Figure 2.7: Hasse Diagram – Total Event Ordering

Teo et al. proposed time-interval (TI) event order based on the interval order in poset [TEO01]. Each event is assigned a time interval, with the event timestamp as the starting point and the timestamp plus a constant W as the ending point, where W is the window size. Definition 2.14 formalizes TI event order. Figure 2.8 shows the Hasse Diagram of the TI event ordering (E, $S_{ti(4)}$) with a window size of four for the set of events E given in Figure 2.4.

Definition 2.14. *Event x is ordered before event y in* ***Time-interval (TI) event order*** *if y.pred = x or y.ante = x or x.timestamp + W < y.timestamp.*

TI order is similar to partial event order with a time window. In addition to the ordering rules of partial event order (Definition 2.12), TI event order imposes that an event x is

ordered before event *y* if *x* belongs to a time window that is earlier than the time window of *y* and their intervals do not intersect. Therefore, if we increase the window size until a certain value, TI event order will become partial event order as shown in Theorem 2.1.

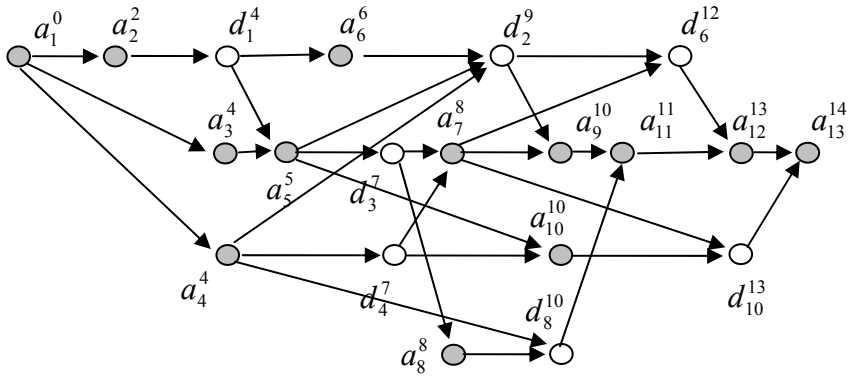


Figure 2.8: Hasse Diagram – Time-interval Event Ordering

Theorem 2.1. *For a given set of events E,* $\exists c$ *such that* $W > c > 0$ *where a time-interval (TI) event order will become a partial event order.*

Proof. To prove this, we show that if $W > c > 0$, the third rule of TI event order (i.e., *x.timestamp* + W < *y.timestamp*) is redundant. Let *a* and *b* be two distinct events in *E* where *b.pred* ≠ *a* and *b.ante* ≠ *a* and *b.timestamp* – *a.timestamp* = *c* is the largest. If window size $W > c$, then rule *x.timestamp* + W < *y.timestamp* will produce an empty set. Hence, only the first two rules (*y.pred* = *x and y.ante* = *x*) will be used, resulting in TI event order with $W > c$ and partial event order producing exactly the same event ordering. For example, if we use $W > 9$, the links such as a_1^0 - a_3^4, a_1^0 - a_4^4, and a_5^5 - d_2^9 are not comparable (these links will not appear in Figure 2.8) and results in the same event ordering as partial event order (see Figure 2.6). ❑

Timestamp (TS) event order is a special case of time-interval event order with a window size W equal to zero [TEO01]. Hence, an event x is ordered before y if and only if *x.timestamp* is smaller than *y.timestamp* (Definition 2.15). Figure 2.9 shows the Hasse Diagram of the TS event ordering (E, S_{ts}) for the set of events E given in Figure 2.4.

Definition 2.15. *Event x is ordered before event y in* ***timestamp (TS) event order*** *if and only if x.timestamp < y.timestamp.*

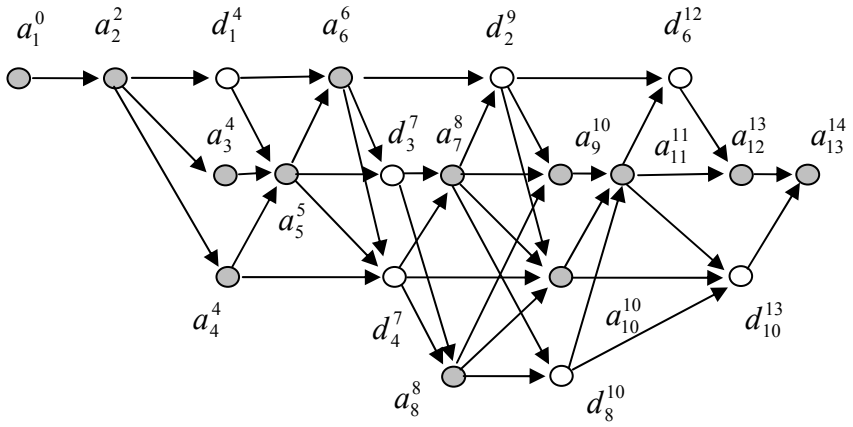


Figure 2.9: Hasse Diagram – Timestamp Event Ordering

2.4 Formalization

A simulator is the implementation of a simulation model. A simulator can be implemented as a sequential program or a parallel program. The sequential simulation maintains its event ordering by using a global event list called future event list (FEL). Parallel simulation may employ several distributed event lists (EL) and a synchronization algorithm (or simulation protocol) is required to maintain its event ordering. For example, in the CMB protocol, null-messages are introduced.

In this section, we shall extract and formalize the event ordering of a number of simulators based on poset. These include sequential simulation and some parallel simulation protocols (such as CMB [CHAN79], Bounded Lag [LUBA89], Time Warp [JEFF85], and Bounded Time Warp [TURN92]).

To show how a simulator executes events during runtime, for simplicity, in this section we assume that each event requires one wall-clock time unit to execute, zero communication delay, and for the parallel simulator each logical process (LP) is mapped onto one physical processor (PP).

2.4.1 Sequential Simulation

The sequential simulation algorithm is presented in Figure 2.10. Events in sequential simulation are totally ordered (only one event is executed at any time). To enforce this ordering, sequential simulation maintains a future event list (FEL) where events are sorted in chronological timestamp order. FEL enables sequential simulation to execute an event with the smallest timestamp (line 12). In case of a tie (i.e., $M \neq \varnothing$ in line 14), it will return an event with the highest priority (z in line 15). Issues and examples on implementing the priority function have been studied in [MEHL92, WIEL97, RONN99].

Assuming we use a priority function that assigns the highest priority to the earliest event that is created, Figure 2.11 shows how this sequential simulation executes the events given in Figure 2.4. Lemma 2.1 proves that events in a sequential simulation are executed according to a total event order [TEO04]. The arrow from event x to event y is added in the figure to indicate that event x must be ordered before event y.

```
SEQUENTIAL SIMULATION
1.  initialize
2.  while (~stop) {
3.    e ← f()
4.    local_clock ← e.timestamp
5.    FEL ← FEL − {e}
6.    E ← execute (e)
7.    FEL ← FEL ∪ E
8.    stop ← g()
9.  }
10.
11. f():event {
12.   x ← head(FEL)
13.   M ← {y | ∀y∈FEL • y.timestamp = x.timestamp}
14.   if (M = ∅) return x
15.   else return {z|∀y∈M ∃!z∈M • priority(z)>priority(y)}
16. }
```

Figure 2.10: Algorithm of Sequential Simulation

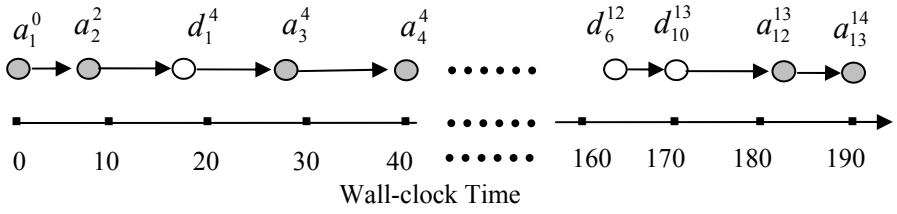


Figure 2.11: Event Execution – Sequential Simulation

Lemma 2.1. *Sequential simulation implements a total event order.*

Proof. Sequential simulation employs a global event list that is sorted by the smallest timestamp first. This guarantees that event x is ordered before event y if and only if $x.ts < y.ts$. The use of a priority function when more than one event have the smallest timestamp guarantees that if $x.ts = y.ts$ event x is ordered before event y if and only if $priority(x) < priority(y)$. Therefore, events in a sequential simulation are executed according to the total event order defined in Definition 2.13. ❑

2.4.2 CMB Protocol

The algorithm of the CMB protocol [CHAN79] is given in Figure 2.12. Each LP maintains a list of LPs that may send events to it (for LP *x*, it is denoted by *SENDER*(*x*)). The ordering rule of CMB protocol imposes that only safe events can be executed. An event in LP *x* is safe for execution if no other LP $\in$ *SENDER*(*x*) will send any event with a smaller timestamp to LP *x*. Therefore, to maintain this ordering, LP *x* must wait for other LP $\in$ *SENDER*(*x*) to send their events (see line 5). This blocking mechanism may lead into a deadlock, therefore null messages are used to solve this problem. Each null message is stamped with a timestamp *ts* which is equal to LP's local simulation clock plus a lookahead value (line 13) to indicate that the sending LP will never transmit any events with a smaller timestamp than *ts*.

Static communication channels are built based on the *SENDER* list of each LP. The CMB protocol assumes an order-preserving communication channel where events that are sent through a channel will be received in the same order. The local clock, state variables, queues, and event list (EL) of each LP are then initialized. Finally, all LPs are activated. LPs are numbered from *0* to *n-1*, where *n* is the number of LPs. Each LP maintains a set of input buffers (*IB*) and a set of output buffers (*OB*). *IB*[*i*] of an LP *x* stores the incoming message from $LP_i \in$ *SENDER*(*x*). *OB*[*i*] stores the messages that will schedule events in LP_i.

An LP is blocked if at least one of its *IB*s is empty (line 5). In lines 6-7, an event with the smallest $f(x)$ is chosen from the *IB*s and *EL* of the LP for execution. Function *f* is the same function that is used in the sequential simulation in Figure 2.10. Therefore, each

LP actually applies a total event order to events that are scheduled on it. Line 8 removes the chosen event from the corresponding list (one of the *IB*s or *EL*). The local clock is updated in line 9. In line 10, event execution may schedule a set of internal events (*IE*) and a set of external events (*EE*). The internal events are saved to *EL* (line 11), and external events to their respective *OBs* (line 12). Line 13 prepares a null message with a timestamp equal to the local clock plus a lookahead value. Lines 14-15 add a null message to any empty *OB*. Line 17 sends all the external events and null messages in the *OB*s. Finally, line 18 checks the stopping condition.

```
CMB PROTOCOL
1. setup static channels for each communicating LP
2. initialize LPs
3. run all LPs

LOGICAL PROCESS
4. while (~stop) {
5.    while (∃i IB[i] = ∅) {}
6.    L ← EL ∪ {Ui IB[i]}
7.    e ← f()
8.    if (∃i e∈IB[i]) IB[i] ← IB[i]-{e} else EL ← EL-{e}
9.    local_clock ← e.timestamp
10.   {IE, EE} ← execute (e)
11.   EL ← EL ∪ IE
12.   ∀i OB[i] ← OB[i] ∪ {z | z∈EE • z.lp = i}
13.   nullMsg.timestamp ← local_clock + lookahead
14.   for i←0 to n-1 do {
15.      if (OB[i] = ∅) OB[i] ← OB[i] ∪ {nullMsg}
16.   }
17.   ∀i send (OB[i])
18.   stop ← g()
19.}
```

Figure 2.12: Algorithm of the CMB Protocol

Assuming a lookahead of 1, Figure 2.13 shows how the CMB protocol executes events in Figure 2.3. For simplicity, we do not show the null messages. During initialization, events a_1^0, a_3^4, and a_4^4 are created at the respective LPs. LP_1 and LP_3 have no

dependency on other LPs (their *SENDER* list = ∅); hence, they can execute the events received right away. LP_1 executes event a_1^0 and schedules events a_2^2 and d_1^4. At the same time, LP_3 executes event a_4^4 and schedules events d_4^7 and a_{10}^{10}. LP_2 cannot execute event a_3^4 because there is no guarantee from LP_1 that it will not send events with a timestamp less than 4. Next, LP_1 executes event a_2^2 while LP_3 executes event d_4^7. The executions produce events a_6^6 and a_7^8. At this time, a null message is sent from LP_1 to LP_2 that it will not send any event with a timestamp earlier than five. It LP_1 guarantees that event a_3^4 is safe. Hence event a_3^4 can be executed in parallel with events d_1^4 and a_{10}^{10} at timestep 2. This process continues until the simulation completes the execution of event a_{13}^{14}. Lemma 2.2 shows the event ordering implemented by the CMB protocol [TEO04].

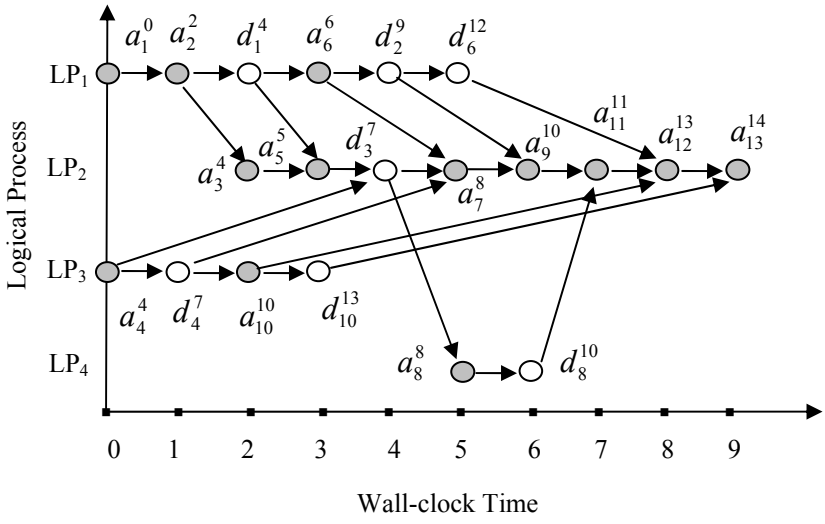


Figure 2.13: Event Execution– the CMB Protocol

Lemma 2.2. *CMB protocol implements an event order whereby event x is ordered before event y if:*

1. y.pred = x, or

2. x.lp ∈ *SENDER*(*y.lp*) *and x.timestamp + lookahead < y.timestamp*

Proof. The blocking mechanism (line 5) ensures that an LP has to wait until all LPs in its SENDER list have sent their events. This ensures that an LP always executes events scheduled in it in timestamp order. Hence, for all events in the same LP, if *y.pred = x* then *x* is ordered before *y*. Further, event *y* in LP_j is executed only if it has the smallest timestamp among the unprocessed events of all LP ∈ *SENDER*(LP_j). Therefore, event *x* in any LP ∈ *SENDER*(LP_j) is ordered before event *y* only if *x.timestamp + lookahead < y.timestamp* where *lookahead* is the lookahead value. ❑

Researchers have proposed various optimizations of the original CMB protocol such as: the demand driven protocol [BAIN88], flushing protocol [TEO94], and carrier null message protocol [CAI92, WOOD94]. We shall show that these optimizations do not alter simulation event ordering in the original CMB protocol, but rather, they can be seen as three different implementations of the same simulation event order.

The algorithm of the demand driven protocol is similar to the algorithm of the CMB protocol. The demand driven protocol modifies line 5 in the algorithm given in Figure 2.12. Instead of being blocked, an LP sends a request to any LP in its *SENDER* list from which it has not received any event. When an LP receives a request, it sends a null message with a timestamp equal to its local clock plus a lookahead value. Hence, instead of sending null messages after each event execution, an LP sends a null message only when it is "required". Although the demand driven protocol reduces the number of null messages, the null message still serves the same purpose as in the original CMB protocol. Therefore, for the same set of events, the demand driven protocol will execute the events in the same order as in the original CMB protocol.

The algorithm of the flushing protocol is also similar to the algorithm of the CMB protocol given in Figure 2.12. When an LP receives a null message from another LP, all unprocessed null messages with a smaller timestamp than the incoming null message at the recipient LP will be removed. Hence, an LP only needs to process a null message with the largest timestamp. When a null message is pumped to an OB, all unsent null messages with a timestamp less than the new null message will be removed. Hence, an LP only needs to send a null message with the largest timestamp. This approach reduces the number of null messages but does not change the main part of the algorithm (lines 6-12) that controls event ordering.

The algorithm of the carrier null message protocol is similar to that of the CMB protocol except for the structure of the null message [CAI90, WOOD94]. The null message in the carrier null message protocol carries routing information to shorten the circulation of null messages on a system with cyclic topology. The change in the null message structure definitely does not make the event ordering any different from that in the original CMB protocol.

2.4.3 Bounded Lag Protocol

Lubachevsky proposed the Bounded Lag (BL) protocol which combines two main rules: bounded lag restriction and minimum propagation delay [LUBA89]. Bounded lag restriction imposes that events can be executed concurrently if they are within the same time window. Minimum propagation delay between LPs is used to determine whether an event is safe to execute. The latter is similar to the rule in CMB protocol, however in the

implementation BL protocol uses a distance matrix instead of using null messages. To maintain its ordering, BL protocol uses barrier synchronization because the global clock (for imposing bounded lag restriction) and the minimum propagation delay must be broadcast to all LPs. The algorithm is given in Figure 2.14. There are two main processes: the nomination of safe events (lines 7-10) and the execution of safe events (lines 11-18).

```
BL PROTOCOL
1.  setup static channels for each communicating LP
2.  setup a distance matrix
3.  initialize global_clock
4.  initialize LPs
5.  run all LPs

LOGICAL PROCESS
6.  while (~stop) {
7.    M ← {∀lp∈LP, lp≠this • head(lp.EL)}
      α ← min {∀e∈M • e.timestamp+d(e.lp, this)}
      broadcast α
8.    barrier synchronization
9.    E ← {∀e∈EL • e.timestamp ≤ min(α, global_clock+W)}
10.   EL ← EL - E
11.   while (E ≠ ∅) {
12.      e ← head(E)
13.      E ← E - {e}
14.      {IE, EE} ← execute (e)
15.      local_clock ← e.timestamp
16.      EL ← EL ∪ IE
17.      Send(EE)
18.   }
19.   stop ← g()
20.   barrier synchronization
21.   global_clock ← min {∀lp∈LP • lp.local_clock}
      broadcast global_clock
22.}
```

Figure 2.14: Algorithm of the Bounded Lag Protocol

The BL protocol makes use of a distance matrix *d* to store the lookahead between any two LPs. Based on the distance matrix, an LP (denoted by *this* in Figure 2.14) determines the earliest time α when its system state can be affected by other LPs (line 7).

The barrier synchronization in line 8 ensures that all LPs calculate α before continuing to the next line. Each LP identifies its safe events based on this rule: events with a timestamp less than α and within a time window of *W* are safe to process (line 9). Line 10 removes all safe events from EL for execution.

The BL protocol retrieves a safe event with the least timestamp in line 12 and removes it from the list *E* in line 13. In line 14, event execution may schedule a set of internal events (*IE*) and a set of external events (*EE*). The internal events will be added to the EL (line 16) and the external events will be sent to their respective LPs (line 17). The barrier synchronization in line 20 is used to ensure that all LPs have processed their safe events before the time window is moved. Line 21 computes the global clock as the minimum of all LPs' local clock. This process is repeated until the stopping condition is met. Lemma 2.3 shows the event ordering that is implemented by the BL protocol [TEO04].

Lemma 2.3. *BL protocol implements an event order whereby event x is ordered before event y if:*

1. $y.pred = x$, *or*
2. $x.timestamp + lookahead < y.timestamp$, *or*
3. $\lfloor x.timestamp/W \rfloor < \lfloor y.timestamp/W \rfloor$.

Proof. The value of α in line 7 gives the smallest timestamp of an unprocessed event *x* (plus lookahead) that may be sent to a particular LP (Figure 2.14). Line 9 shows that if event *y* in LP_i can be executed in parallel with event *x* from another LP_j, then $y.timestamp \leq \alpha$ (i.e., $x.timestamp+distance(LP_i, LP_j)$), and both *x* and *y* must be in the same time window of size *W* (Note that the distance between LP_i and LP_j is the lookahead between the two LPs). Therefore, the contra positive, i.e., event *x* is executed

before event *y* only if *x.timestamp* + *lookahead* < *y.timestamp* or events *x* and *y* are in two different time windows of size *W,* is true (of course the time window of *x* should be earlier than the time window of *y*). ❑

2.4.4 Time Warp Protocol

Jefferson proposed Time Warp (TW) protocol which implements a rule that if event *x* causes event *y*, then the execution of event *x* must be completed before the execution of event *y* starts [JEFF85]. The definition of ″*x* causes y″ follows the relation *happened before* in [LAMP78]. To implement this ordering, TW protocol uses what is called local control mechanism (rollback and state saving) and global control mechanism (global clock calculation and fossil collection). The algorithm is given in Figure 2.15.

First, the TW protocol initializes the local clock, state variables, queue, and event list of each LP before activating all LPs. Each LP stores all incoming events in an input buffer (*IB*) which is sorted based on the timestamp of the incoming events. Lines 4-11 find the first real event *m*. Line 5 retrieves an event *m* with the smallest timestamp for execution. Line 6 checks if *lcc* is violated.

The function *dual*(*m*) returns the anti-message of event *m* (if *m* is an event), or it returns the event *x* is the *m* is the anti-message of event *x*. Line 7 detects whether rollback has to be done. Rollback happens when a straggler event *m* is received and one of the following conditions is satisfied:

- *m* is a real event (not an anti-message) and no anti-message for event *m* has been received (because if the anti-message has been received, we can simply annihilate event *m*, hence rollback is not necessary); or
- *m* is an anti-message for event *x* and event *x* has been executed

```
TIME WARP PROTOCOL
1. initialize LPs
2. run all LPs

LOGICAL PROCESS
3. while (~stop) {
4.   do {
5.     m ← head(IB)
6.     if (m.timestamp < local_clock) {
7.       if (((m ≠ anti_message) and dual(m)∉ IQ) or
         ((m = anti_message) and dual(m)∈ IQ)) RollBack()
8.     }
9.     if (dual(m) ∈ IQ) Annihilate(m) else IQ ← IQ ∪ {m}
10.    IB ← IB − {m}
11.  } while ((m = anti_message) and (IB ≠ ∅))
12.  if (m = anti_message) e ← head(EL)
13.  else {
14.    if (m.timestamp < head(EL).timestamp) e ← m
15.    else {e ← head(FEL); EL ← EL − {e}; EL ← EL ∪ {m}}
16.  }
17.  {IE, EE} ← execute (e)
18.  local_clock ← e.timestamp
19.  EL ← EL ∪ IE
20.  StateSaving()
21.  Update(global_clock)
22.  FossilCollection()
23.  Send(EE)
24.  stop ← g()
25.}
```

Figure 2.15: Algorithm of the Time Warp Protocol

If *m* is an anti-message, line 9 will annihilate the associated event that has to be cancelled; otherwise, it will add *m* to a list called the input queue (*IQ*). *IQ* is used to store the history of all incoming messages (processed and unprocessed). Line 10 removes *m* from *IB*. Lines 12-15 retrieve an event *e* which has the smallest timestamp

from the event list (EL) and choose the event with a smaller timestamp, between *m* and *e*. Line 17 executes the chosen event. This execution may produce a set of *internal events* (*IE*), i.e., events that are scheduled to happen in the same LP, and a set of *external events* (*EE*), i.e., events that are scheduled to happen in other LPs. Line 18 updates the local clock and line 19 updates the EL. Line 20 saves the state of an LP. The global clock is updated in line 21. Events with timestamp less than the global clock will never be rollbacked. These events are called *committed events*. Hence, the memory that has been allocated to them can be reclaimed with the fossil collection process in line 22. Line 23 sends out the external events. Finally, line 24 checks the stopping condition.

Figure 2.16 shows how one of the possible executions using TW protocol for the set of events in Figure 2.3. To better illustrate this, we include non-committed events in the graph (denoted by the dotted circles). During initialization (Figure 2.15 line 1), events a_1^0, a_3^4, and a_4^4 are stored in the ELs of the respective LPs (LP_1, LP_2, and LP_3). These LPs optimistically execute events a_1^0, a_3^4, and a_4^4 in parallel and produce events a_2^2, d_1^4, d_3^7, d_4^7, and a_{10}^{10}. In the next stage, events a_2^2, d_3^7, and d_4^7 are processed and generate events a_6^6, a_7^8, and a_8^8. Now, the unprocessed events are d_1^4, a_6^6, a_7^8, a_{10}^{10}, and a_8^8. Next, events d_1^4, a_7^8, a_{10}^{10}, and a_8^8 are executed producing new events a_5^5, d_2^9, d_8^{10}, and d_{10}^{13}. While other LPs are executing events a_6^6, d_{10}^{13}, and d_8^{10}, LP_2 knows that it has received a straggler event a_5^5 (the current local clock at LP_2 is 8). Hence, LP_2 has to rollback to local clock 4, cancel the execution of events d_3^7 and a_7^8, and finally execute event a_5^5. Since event a_8^8 has been sent by event d_3^7, event a_8^8 has to be cancelled by sending an anti-message to LP_4. In the next steps, LP_2 has to re-execute events d_3^7 and

a_7^8. Similarly, LP_4 has to re-execute events a_8^8 and d_8^{10}. This process continues until the simulation is completed. If we remove all non-committed events from Figure 2.16, the event execution shows that all events are ordered based on partial event order as proven in Lemma 2.4 [TEO04].

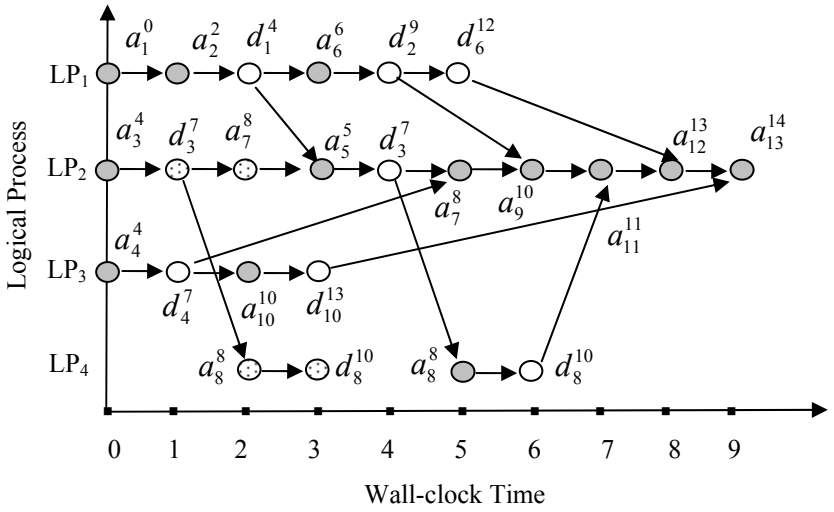


Figure 2.16: Event Execution– Time Warp Protocol

Lemma 2.4. *TW protocol implements a partial event order.*

Proof. The rollback process ensures that all events in the same LP are executed in timestamp order. This implies that event *x* is ordered before event *y* if *y.pred* = *x*. The insertion of internal events to EL and transmission of external events are done after the event execution in line 17. This ensures that event *x* is ordered before event *y*, if *y.ante* = *x*. These rules match the definition of the partial event order (Definition 2.12). ❑

The original TW protocol uses the copy state saving mechanism [JEFF85]. Other state saving mechanisms include incremental state saving, sparse state saving, and hybrid state saving [CLEA94, RONN94, SKOL96, WEST96, FRAN97, QUAG99, SOLI99]. A state

saving mechanism only affects the way a system state is saved. Hence, TW protocols with various state saving mechanisms still implement the same partial event order defined in Definition 2.12.

2.4.5 Bounded Time Warp Protocol

Bounded Time Warp (BTW) protocol [TURN92] is proposed to limit the degree of optimism in TW protocol by setting a bound on how far an LP can advance ahead of other LPs. This is accomplished by using a time window synchronization. A window size *W* is defined so that all LPs are allowed to optimistically process events with the timestamp between *GVT* and *GVT*+*W* where *GVT* is the global virtual time or global clock. No LP can advance beyond *GVT*+*W* before all LPs have reached this boundary. The algorithm of the Bounded Time Warp (BTW) protocol [TURN92] is given in Figure 2.17. The algorithm comprises two main parts: the execution of events using the TW protocol (lines 5-26) and time window synchronization (line 28).

During initialization, the time window interval is computed (*wstart* and *wstop*). Lines 5-26 are similar to the corresponding lines in the TW protocol (Figure 2.15). All LPs are synchronized to optimistically execute events as in the TW protocol, but with the condition that the event must be within the current time window. In line 27, barrier synchronization ascertains that every LP has executed all events within the time window before the new time window is computed in line 28. Line 29 updates the global clock that will be used in the fossil collection process in line 30. Finally, line 31 checks the stopping condition.

```
BTW PROTOCOL
1.  wstart ← 0; wstop ← wstart + W
2.  initialize LPs
3.  run all LPs

LOGICAL PROCESS
4.  while (~stop) {
5.    while (local_clock ≤ wstop) {
6.      do {
7.        m ← head(IB)
8.        if (m.timestamp < local_clock) {
9.          if (((m ≠ antimessage) and dual(m)∉ IQ) or
10.         ((m = antimessage) and dual(m)∈ IQ)) RollBack()
11.       }
12.       if (dual(m) ∈ IQ) Annihilate(m) else IQ ← IQ ∪ {m}
13.       IB ← IB − {m}
14.     } while ((m = antimessage) and (IB ≠ ∅) and
15.     (m.timestamp ≤ wstop))
16.     if (m = anti_message) e ← head(EL)
17.     else {
18.       if (m.timestamp < head(EL).timestamp) e ← m
19.       else {e ← head(FEL); EL ← EL − {e}; EL ← EL ∪ {m}}
20.     }
21.     {IE, EE} ← execute (e)
22.     local_clock ← e.timestamp
23.     StateSaving()
24.     EL ← EL ∪ IE
25.     Send(EE)
26.   }
27.   barrier synchronization
28.   wstart ← wstop; wstop ← wstart + W
29.   Update(global_clock)
30.   FossilCollection()
31.   stop ← g()
32. }
```

Figure 2.17: Algorithm of the Bounded Time Warp Protocol

Assuming a window size of 4, Figure 2.18 shows how the BTW protocol executes the events given in Figure 2.3. The initial time window interval is from 0 (inclusive) to 4 (exclusive) denoted by [0, 4). At initialization, events a_1^0, a_3^4, and a_4^4 are put in the respective ELs. Event a_1^0 is the only event within the time window, so it will be executed and will produce events a_2^2 and d_1^4. Next, event a_2^2 will be processed because

its timestamp is still within the time window. This execution produces event a_6^6. After barrier synchronization, the time window interval becomes [4, 8). The current unprocessed events are d_1^4, a_6^6, a_3^4, and a_4^4. All of them are within the new time window; hence, they will be executed. These steps are repeated until event a_{13}^{14} is executed.

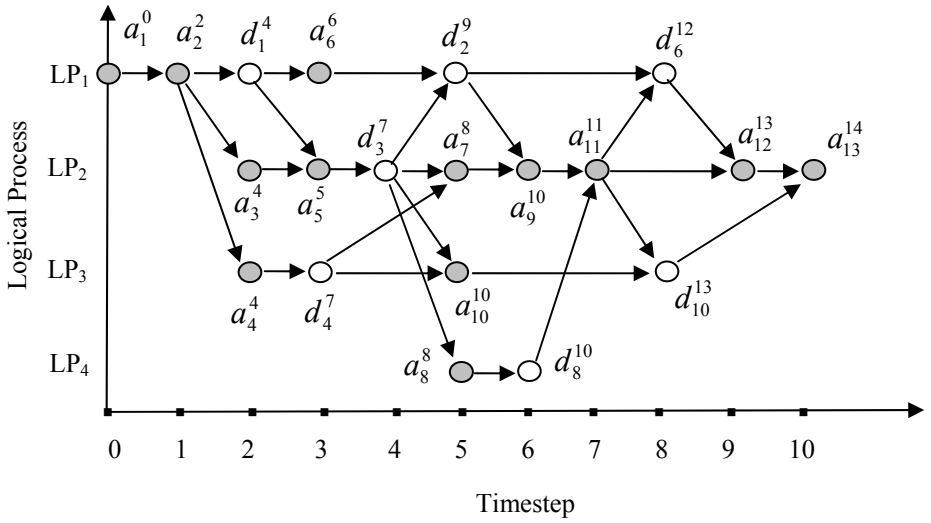


Figure 2.18: Event Execution – the BTW Protocol

Lemma 2.5. *BTW protocol implements an event order whereby event x is ordered before event y if:*

1. *y.pred = x, or*
2. *y.ante = x, or*
3. $\lfloor x.timestamp/W \rfloor < \lfloor y.timestamp/W \rfloor$.

Proof. Without time window, BTW protocol is the same as TW protocol hence the ordering rules of partial event order hold, i.e., event *x* is ordered before event *y* if *y.pred* = *x* or *y.ante* = *x*. The additional time window synchronization imposes that the partial event order is applied to a set of events that occur within the same time window. Consequently, only events within the same time window can potentially be executed in

parallel. Therefore, if event *x* occurs within a time window that is earlier than the time window of event *y*, event *x* will be executed before event *y*. ❑

2.4.6 Protocols that Ignore Causality

All simulators that have been discussed in this chapter produce a correct simulation result because its event ordering adheres to *local causality constraint* (*lcc*). Definition 2.16 formalizes *lcc*.

Definition 2.16. *An event order R adheres to lcc if for any two distinct events* $x,y \in E$ *and* $y.pred = x$, *then x is ordered before y.*

Some researchers argued that as the cost of synchronization is high, in cases where the error in simulation result is statistically insignificant, it is better to trade off correctness with performance [FUJI96, MART97]. Based on this argument, protocols that ignore *lcc* were proposed [FUJI96, MART97, RAO98, THON99, FUJI99]. Our event ordering analysis focuses on simulation event order that adheres to *lcc* because this approach produces correct simulation results. However, our formalization can still be applied to protocols that ignore *lcc* as shown in the following example.

We demonstrate the formalization of the event ordering of unsynchronized protocol proposed in [MART97]. In this protocol, events are executed whenever they are ready. If *lcc* is violated, it does not provide any mechanism to rectify it; therefore, no synchronization algorithm is required. Errors in simulation results due to *lcc* violation are studied in [MART97, RAO98, THON99]. In general, the degree of significance is

application-dependent. Figure 2.19 shows how the events in Figure 2.4 are executed using this protocol. This protocol does not re-execute events d_3^7, a_7^8, and a_{11}^{11}, even though they are not executed in chronological timestamp order (*lcc* violation).

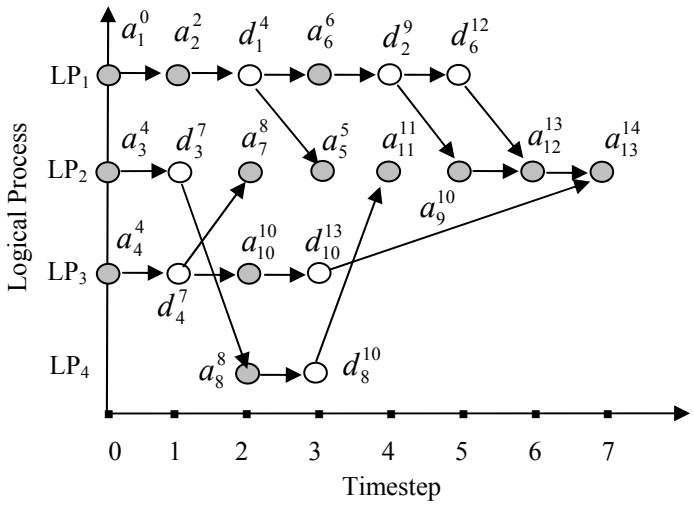


Figure 2.19: Event Execution – Unsynchronized Protocol

Lemma 2.6. *Unsynchronized protocol implements an event order whereby event x is ordered before event y if y.ante = x.*

Proof. Unsynchronized protocol is similar to TW protocol without rollback. Hence, it does not guarantee that event *x* is ordered before event *y* if *y.pred* = *x* as in TW protocol. However, it still guarantees that event *x* is ordered before event *y* if *y.ante* = *x* because event *y* is sent only after the execution of event *x* is completed. ❑

Unsynchronized protocol does not guarantee the correctness of the simulation result. In view of that, Carothers et al. proposed a protocol that implements the same event ordering and guarantees the correctness of the simulation result [CARO00]. They applied a reverse computation technique to rectify the effect of *lcc* violation.

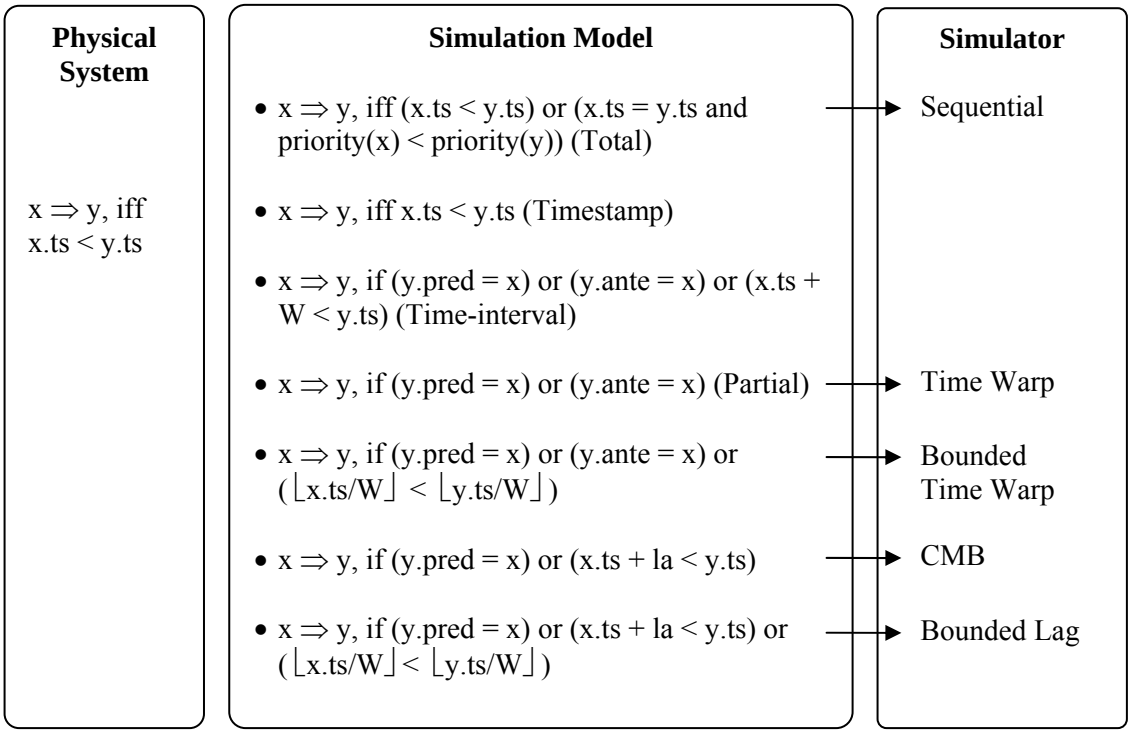


Figure 2.20: Summary on Simulation Event Ordering Formalization

We summarize the formalization of the discussed event orderings in Figure 2.20 [TEO04]. There is only one event ordering in the physical system. In the simulation model, different event orderings can be used to simulate the physical system. The ordering rules of each event order are shown in the form of *x* is ordered before *y* (denoted by $x \Rightarrow y$) if a list of conditions hold. A simulator implements a certain event order. The arrow from event ordering *R* in simulation model to simulator *S* denotes that *S* implements *R*. Notice the close relation among simulation event orderings; for example, the rule of partial event order is a subset of time-interval event order. We will use this relation to compare the degree of event dependency among different event orders in the next chapter.

2.5 Summary

Event ordering plays an important role in distributed computing. Therefore, many formalization and analytical works on event ordering in distributed systems have been proposed. In two areas of distributed systems, i.e., memory consistency model and broadcast communication services, researchers separate the specification of event ordering from its implementation. There are at least two benefits to separating them. First, it helps us understand the relation between different event orderings. Second, it provides a performance analysis that is independent of implementation factors. This is important because the performance comparison between two implementations (each implementing a different event ordering) cannot be used to conclude which event ordering is better. It is possible that one of the event orderings is better but implemented poorly.

In this chapter, we have separated simulation event ordering from its implementation. We have proposed the formalization of simulation event orderings based on the partially ordered set. We have also extracted and formalized the event ordering of a number of simulator implementations.

Chapter 3

Performance Characterization

Simulation performance analysis is important because it can be used to identify opportunities for performance improvement and to compare different modeling and parallelism strategies. However, analyzing simulation performance is a complex task because it depends on many interwoven factors [FERS97].

In this chapter, we propose a framework for characterizing simulation performance. Simulation performance is characterized along the three natural boundaries in modeling and simulation, i.e., *physical system* (simulation problem), *simulation model*, and *simulator* (implementation). The main objective is to provide a basis for analyzing simulation performance from a simulation problem to its implementation. We focus on time (event parallelism) and space (memory requirement) performance at each layer. Event parallelism is defined as the number of events executed per unit of time. Therefore, event parallelism is influenced by the unit of time which complicates performance comparison across layers because the time units used at different layers are different. An additional process is therefore necessary to allow performance comparison across layers. We propose a time independent performance measure called *strictness* which focuses on the dependency among events only.

This chapter is organized as follows. First, we present our motivation and review a number of related works that influence our research. Next, we propose our performance characterization framework. This is followed by a discussion on time performance analysis. The next section presents space performance analysis. Next, we discuss the concept of event ordering strictness. Finally, we conclude this chapter with a summary.

3.1 Motivation

In this section, we review a number of performance evaluation frameworks that motivate our research. They focus on either a certain simulator (e.g., Time Warp protocol, CMB protocol) or a certain aspect of performance study (e.g., benchmark, workload) as shown in the following discussion. This motivates us to propose a framework that unifies them.

3.1.1 Related Works

Barriga et al. noted that a common benchmark suite is required in evaluating the performance of a simulation [BARR95]. They advocated an *incremental benchmark methodology* to evaluate the time performance (event rate) of a Time Warp protocol. The ingenious idea here is that they start from a simple benchmark (i.e., self-ping), and by incrementally adding more complexity to the benchmark, they measure various overheads of the Time Warp protocol running on a multiprocessor. They also showed that the incremental benchmark methodology can be used to compare the performance of different variations of Time Warp protocol.

Balakrishnan et al. presented a general performance analysis framework for parallel simulators in [BALA97]. The main objective is to provide a common benchmark suite that studies the performance of simulators using synthetic and realistic benchmarks. To achieve this objective, they implemented several tools, i.e. Workload Specification Language (WSL) and Synthetic Workload Generator (SWG). WSL is a language that describes a benchmark and its workload parameters. SWG generates synthetic workloads based on a given WSL. A translator is required to translate WSL to the code recognized by a target simulator. They applied this framework to analyze the time performance (event rate) of a Time Warp protocol. These tools can also be used to support the incremental benchmark methodology [BARR95].

Jha and Bagrodia characterized simulation performance as a function of protocol independent factors and protocol dependent factors [JHA96]. The protocol independent category includes factors such as processor speed and communication latency. The protocol specific category includes factors such as null message overhead in the CMB protocol. The same performance characterization is also mentioned in [BARR95]. However, Jha and Bagrodia's proposed framework analyzes protocol independent factors only. They implemented an Ideal Simulation Protocol (ISP) based on the concept of critical path analysis (CPA). ISP computes the critical path by actually executing the simulation model on parallel computers in contrast to a uniprocessor in the original CPA. Therefore, they claimed that ISP gives a more realistic upper bound on speed-up than CPA. Further, they defined the efficiency of protocol as the ratio of the execution time of ISP to the execution time of the target protocol. Of course, as in CPA, their performance evaluation framework is limited to non-supercritical protocols such as the CMB protocol [JEFF91]. Recently, based on the same performance characterization as

in [BARR95, JHA96], Song evaluated the time performance of a CMB protocol [SONG01]. However, his work focuses on the protocol dependent factors, i.e., the blocking time in the CMB protocol.

Teo et al. proposed a different performance evaluation framework which evaluates performance along three components: simulation model, parallel simulation strategy, and execution platform [TEO99]. The simulation model views the physical system to be simulated as a queuing network of LPs. The parallel simulation strategy refers to the protocol dependent factors. The execution platform refers to platform dependent factors, such as the speed of processors and communication latency. The paper focuses on the event parallelism analysis at the simulation model.

Liu et al. implemented a parallel simulator suite called Dartmouth Scalable Simulation Framework (DaSSF) [LIU99]. They proposed a simple high level approach to estimate the performance of their simulator. They measured the simulator's internal overheads such as context switching, dynamic object management, procedure call, dynamic channel, process orientation, event list, and barrier synchronization. They used these measurements to estimate the performance of the simulator in simulating a given physical system.

In the early days, most work in the performance evaluation of parallel simulation concentrated on time performance and assumed that the amount of memory was unlimited [LIN91]. Since then, there has been a growing body of research that studies the space aspect of parallel simulation but most of it concentrates on managing the memory required to implement various synchronization protocols. In particular, the conservative

approach focuses on reducing the number of null messages, for example, the carrier-null mechanism [CAI90], the demand-driven method [BAIN88], and the flushing method [TEO94]. In the optimistic approach, the focus is placed on delimiting the optimism, thus constraining memory consumption, and on reclaiming memory before a simulator runs out of storage. Examples include the various state saving mechanisms [SOLI99], the use of event horizon in Breathing Time Bucket [STEI92], the adaptive Time Warp [BALL90], the message send-back [JEFF90], the artificial rollback [LIN91], and the adaptive memory management [DAS97].

There are also a number of studies which examine the minimum amount of memory required for various parallel simulation implementations under the shared-memory architecture (but not applicable to the distributed memory architecture [PREI95]). Their main objective is to design an efficient memory management algorithm which guarantees that the memory requirement of the parallel simulation is of the same order as sequential simulation. Jefferson refers to this algorithm as an *optimal memory management algorithm* [JEFF90]. Jefferson and Lin et al. proved that the CMB protocol is not optimal [JEFF90, LIN91]. Lin and Preiss analyzed the memory requirement of sequential simulation, the CMB protocol and the Time Warp protocol [LIN91]. Based on their characterization, they showed that the CMB protocol may require more or less memory than sequential simulation depending on the characteristics of the physical system. However, the Time Warp protocol always requires more memory than sequential simulation. Das and Fujimoto studied the effect of varying memory capacity on the performance of the Time Warp protocol [DAS97]. In particular, they studied the time performance of the Time Warp protocol as a function of the available memory space.

Wong and Hwang noted that space performance (i.e., memory requirement) has not been extensively studied [WONG95]. They proposed a critical path-like analyzer to predict the amount of memory consumed in a variant of the CMB protocol by measuring the number of events in the system. However, they did not give any analytical or empirical results. Based on their (unreported) preliminary result, they suggested that it is possible to predict the memory requirement of the CMB protocol from the execution of a sequential simulator.

The space performance becomes increasingly important as the simulation problem becomes more complex. Liljenstam et al. modeled the effect of a large scale Internet worm infestation [LILJ02]. They noted that the packet-level simulation uses a large amount of memory to model hosts and packets. They observed that the memory usage would exceed 6GB to model 300,000 hosts. A large scale multicast networks simulation also requires a significant amount of memory [XU03]. The memory requirement can be as high as 5.6GB for 2,000 stations. Zymanski et al. noted that with the emerging requirements of simulating larger and more complicated networks, the memory size becomes a bottleneck [ZYMA03].

3.1.2 Performance Metrics

As shown before, most frameworks focus on the time performance of a simulator. The common metrics used are:

1. *Speed-up* – it is defined as the ratio of the execution time of the best sequential simulator and the execution time of a target simulator [JHA96, BAJA99, BAGR99, SONG00, XU01].
2. *Event rate* – it measures the throughput of a simulator, i.e., the average number of useful events executed per unit time [BARR95, FERS97, BALA97].
3. *Execution time* – it measures the amount of (wall-clock) time that is required to complete a simulation [SOKO91, BAJA99, BAGR99].
4. *Efficiency* – it is defined as the ratio of the execution time of ISP to the execution time of the target protocol [JHA96]. This is different from the definition of efficiency in parallel computing, i.e., the ratio between speed-up and the number of processors.
5. *Blocking Time* – it is defined as the duration when an LP is waiting for a safe event to be executed [SONG01].
6. *Cost per simulation time unit* – it is the ratio of wall-clock time to simulation time [DICK96, LIU99].

Although, it has not been studied extensively, some researchers have indicated that the appropriate metrics for space performance are:

1. *Average memory usage* – it is defined as the average memory usage for every processor [YOUN99]. Young et al. studied the time and space performance of their proposed fossil collection algorithm. The average memory usage shows the memory utilization across all processors during simulation run.
2. *Peak memory usage* – it measures the maximum memory used for a simulation run. Zhang et al. defines it as the *maximum* of all machines' maximal memory usage [ZHANG01, LI04]. Young et al. used a different definition, i.e., the *average* of all machines' maximal memory usage [YOUN99].

3. *Maximum number of events* [JEFF90, LIN91].
4. *Null message ratio* – it is defined as the ratio of total number of null messages to total number of events. This metric is specific to the CMB protocol [BAIN88, CAI90, TEO94].

3.2 Proposed Framework

Given the many proposed frameworks, we feel that it is essential to have a complete and unified performance evaluation framework. The previous section has shown that most researchers characterize simulation performance as a function of protocol dependent factors and protocol independent factors [BARR95, JHA96, SONG01]. Bagrodia also included the partitioning related factors in addition to the protocol dependent factors and protocol independent factors [BAGR96]. Ferscha further noted that a performance evaluation framework should consider the six categories of performance influencing factors, namely, simulation model, simulation engine, optimization, partitioning, communication and target hardware [FERS96]. Later, Ferscha et al. simplified the classification into three categories, namely, simulation model, simulation strategy, and platform [FERS97]. Simulation model refers to the characteristics of a model, such as the probability distribution function of job arrivals. Simulation strategy refers to the characteristics of a protocol, such as state saving policy in the Time Warp protocol and null message optimizations in the CMB protocol. Platform refers to the characteristics of an execution platform, such as processor speed, communication latency and memory size. The same characterization is also suggested in [TEO99].

We propose to characterize simulation performance in three layers, i.e., *physical system*, *simulation model*, and *simulator* as shown in Figure 3.1. This thesis focuses on the physical systems that are formed by sets of interacting service centers. Hence, a physical system can be formalized as a directed graph where each vertex represents a service center and an edge from service center *i* to service center *j* shows that service center *i* may schedule an event to occur in service center *j*. The time used at the physical system layer is called physical time (see Chapter 1).

The second layer is the *simulation model layer*. In the virtual time paradigm [JEFF85], a simulation model is viewed as a set of interacting *logical processes* (LPs). Each LP models a physical process (service center) in the physical system. The interaction among physical processes in the physical system is modeled by exchanging events among LPs in the simulation model. Therefore, a simulation model can also be formalized as a directed graph where each vertex represents an LP, and an edge from LP *i* to LP *j* denotes that LP *i* may send an event to LP *j*. The time unit used at the simulation model layer is timestep. A *timestep* is defined as the time that is required for an LP to process an event.

A simulation model is implemented as a simulator, and it is executed on a computer consisting of one or more *physical processors* (PPs). In a sequential simulator, events are executed based on a total event order. In a parallel simulator, one or more LPs at the simulation model layer are mapped onto a PP. Therefore, the set of PPs also forms a directed graph where an edge from PP *i* to PP *j* denotes that PP *i* may send an event to PP *j*. The simulator constitutes the third layer.

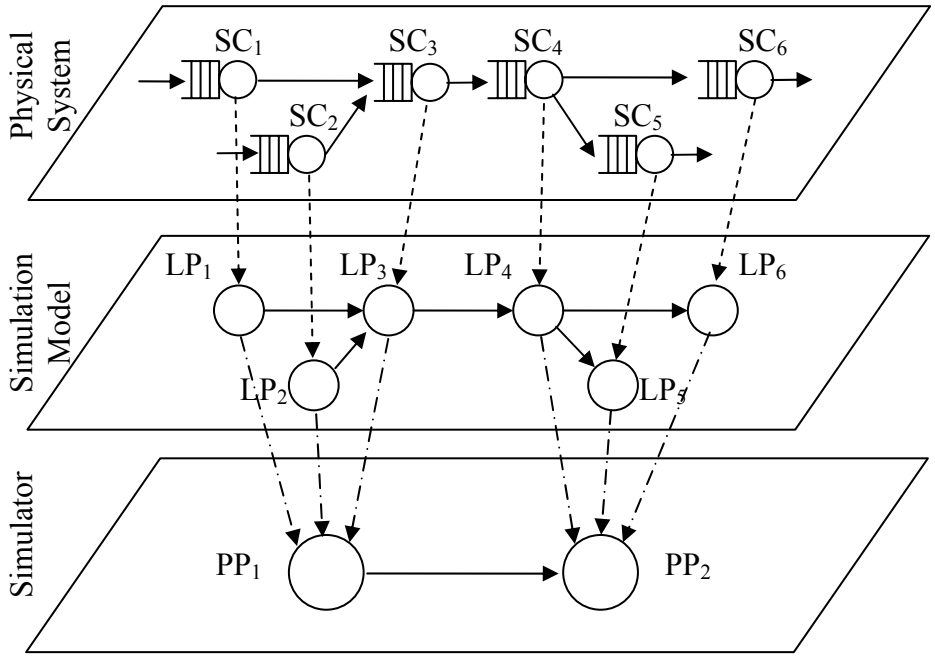


Figure 3.1: Three-Layer Performance Analysis Framework

Ideally, any analysis at the physical system layer should be independent of the simulation model and implementation. It should depend on the characteristics of the physical system only. Therefore, analysis can be conducted before building a simulation model (hence, its implementation). Similarly, any analysis at the simulation model layer should be implementation independent so that analysis can be conducted before implementation. Analysis at the simulator layer is implementation dependent.

In order to relate the analyses conducted at two different layers, we need a unifying concept. Bagrodia et al. introduced a unifying theory of simulation, and from the theory, they derived an algorithm called the space-time algorithm [BAGR91]. A simulator called Maisie was built to implement the space-time algorithm. A physical system can be modeled and simulated using Maisie. The performance of a simulator that is supported by the Maisie run-time system can be evaluated. Theoretically, Bagrodia et al. showed that sequential simulation, the CMB protocol, and the Time Warp protocol are instances

of the space-time algorithm. However, the relationship between different instances and their performance is not clear and they did not show the comparative results.

The idea of using a unifying concept where each simulator can be seen as an instance of the same abstraction motivates us to use the concept of event ordering introduced in Chapter 2 as the unifying concept. The reason is that event ordering exists at the three layers. Therefore, it is possible to use the concept of event ordering to relate analyses from the different layers. Based on the physical time, there is only one event order in any physical system (Definition 2.11). At the simulation model layer, the event order in a physical system can be modeled using different event orders to exploit different degrees of parallelism. In the implementation (simulator layer), synchronization overhead is incurred in maintaining event ordering at runtime. Similar to [BAGR91], where every simulator can be seen as an instance of the time-space algorithm, every simulator in our framework can be seen as an implementation of an event order.

3.3 Time Performance Analysis

Event parallelism is commonly used as a time performance measure [WAGN89, SHOR97, WANG00]. It is defined as the average number of events that occur (or are processed) per unit time. In this thesis, we choose event parallelism for two reasons. First, the underlying theory of our framework is event ordering. Second, in discrete-event simulation, events are atomic and are the lowest level of parallelism. This means that the code within an event is executed sequentially. Equation 3.1 defines *event parallelism*:

$$\Pi = \frac{\|E\|}{D} \tag{3.1}$$

where E is the set of events, $||E||$ is the number of events in E, and D denotes the measurement duration.

For the same physical system, the number of (real) events in the physical system, in the simulation model and in the simulator is the same. In a physical system, events may occur every minute, hour, and so on. In the simulation, these events can be executed at a different rate, depending on the characteristics of the execution platform (processor's speed, communication delay, etc.). We introduce the simulation model layer to allow analysis that is independent of the characteristics of the execution platform. Assuming that the time to execute an event is one timestep at the simulation model layer, the event parallelism can be expressed as the number of events executed per timestep. We refer to the event parallelism at the physical system, simulation model and simulator layers as Π^{prob}, Π^{ord} and Π^{sync}, respectively.

3.3.1 Physical System

A physical system has a certain degree of *inherent event parallelism* (Π^{prob}). The parallel simulation of a physical system may fail to deliver the desired improvement in performance if the physical system itself contains a low degree of inherent event parallelism [BAGR96]. The analysis at this layer can be used to compare the inherent event parallelism of different physical systems and to determine whether the problem is suitable for parallel simulation. The definition of Π^{prob} is given in Equation 3.2:

$$\Pi^{prob} = \frac{\|E\|}{D^{prob}} \tag{3.2}$$

where E is the set of events, $||E||$ is the number of events in E, and D^{prob} denotes the observation period (in physical time unit).

3.3.2 Simulation Model

At the simulation model layer, a less strict event order promotes more flexibility in executing events. An event order *R* selects a number of events that can be executed from a set of events *E*. The selected events are removed from *E* for execution. An event execution may schedule new events that will be added to *E*. This process repeats until a certain stopping condition is met. A less strict event order can potentially execute events at a faster rate since it has higher flexibility in executing events. Therefore, a less strict event order can potentially exploit more event parallelism (Π^{ord}) than a stricter event order. The analysis at this layer reveals the degree of event parallelism exploited by different event orders from the same physical system. This analysis can also be used to compare the time performance of two simulators, provided we know the event order that is implemented by each simulator. By comparing the event parallelism of the two event orders, we can evaluate the performance of the two protocols, independent of any implementation factors. The definition of Π^{ord} (model parallelism) is given in Equation 3.3:

$$\Pi^{ord} = \frac{\|E\|}{D^{ord}} \tag{3.3}$$

where *E* is the set of events, ||*E*|| is the number of events in *E*, and D^{ord} denotes the simulation duration (in timesteps).

3.3.3 Simulator

A simulator can be implemented as a sequential program or a parallel program. In a parallel simulator, a synchronization algorithm (or simulation protocol) is necessary for

maintaining a correct event ordering across processors. Enforcing event ordering at runtime incurs implementation overhead (such as null messages in the CMB protocol and rollback in the Time Warp protocol) that results in performance loss. The *effective parallelism* (Π^{sync}) is defined in Equation 3.4:

$$\Pi^{sync} = \frac{\|E\|}{D^{sync}} \tag{3.4}$$

where E is the set of real events (it does not include the overhead events such as null messages), $\|E\|$ is the number of events in E, and D^{sync} denotes the simulation execution time (in wall-clock time units).

Analysis at the simulator layer can be used to study the effect of different implementation factors on the performance of a simulator, i.e., the efficiency of implementations. Examples include execution platform [BARR95] and partitioning strategy [KIM96]. Since the same event order can be implemented differently, the analysis at this layer can also be used to compare the performance of two different implementations of the same event order. For examples, the performance comparison between the CMB protocol and the carrier null message protocol [CAI90], and the comparison of different state saving mechanisms in the Time Warp protocol [SOLI99].

3.3.4 Normalization of Event Parallelism

In the previous three subsections, we have analyzed event parallelism at each layer independent of the other layers. It is also useful to compare event parallelism across layers. For example, we can see how inherent event parallelism in a physical system is exploited by a particular event order at the simulation model layer, or we can analyze performance loss (the difference in event parallelism between the simulation model layer

and the simulator layer) due to overheads at the simulator layer. Since the time units used at different layers are different, the event parallelism across layers cannot be compared directly as shown in the following example.

We want to study the performance of simulating a physical system. During an observation period of 10,000 minutes, 200,000 events occur in the physical system. Hence, the inherent event parallelism (Π^{prob}) is 200,000/10,000 = 20 events per minute (from Equation 3.2). At the simulation model layer, we can execute these events using a different event order. The measurement shows that the same set of events is executed in 3,500 timesteps using the CMB event order. Hence, the model parallelism (Π^{ord}) is 200,000/3,500 = 57 events per timestep (from Equation 3.3). At the simulator layer, a CMB protocol uses null messages to maintain the event order at runtime. The measurement shows that the simulation completion time is 1,650 seconds on four processors. Therefore, the effective event parallelism (Π^{sync}) is 200,000/1,650 = 121 events per second.

We cannot compare Π^{prob} (20 events / minute), Π^{ord} (57 events / timestep), and Π^{sync} (121 events / second) directly. From the definition, event parallelism depends on the dependency among events and time. Therefore, to allow comparison across layers, we can either convert all time units to a common unit, or normalize event parallelism so that it becomes time-independent.

In the first approach, we convert all time units to a common unit; in this case, we choose second. At the physical system layer, the conversion from minute to second is straightforward. At the simulation model layer, one timestep is defined as the time to

execute one event at the simulation model layer. To convert the timestep into second, we need the real event execution time at the simulator layer. Let us assume that from measurement, the average time to execute an event at the simulator layer is 18ms. Hence, one timestep at the simulation model layer is equal to 18ms at the simulator layer. By converting the timestep into a wall-clock time unit, analysis at the simulation model layer can be viewed as an analysis at an ideal execution platform where communication delay is zero and the number of PPs is unlimited so that each LP can be mapped onto a PP. Now, we can compare event parallelism at the three layers.

Π^{prob}	= 20 events / minute	= 0.33 events / second
Π^{ord}	= 57 events / timestep	= 3,167 events / second
Π^{sync}		= 121 events / second

The results can be interpreted as follows. The simulator executes events in a faster rate than in the physical system (it is called *faster than real-time* simulation in [MART03]). It shows that the simulator can compress the time in the physical system. Theoretically, if the communication delay is zero and each LP is mapped onto a unique PP, the simulator should be able to achieve a parallelism of 3,167 events per second. Due to the overheads at the simulator layer and the limitation in the number of PPs, the simulator can only exploit a parallelism of 121 events per second.

In the second approach, we derive the normalized event parallelism from the dependency among events only. The dependency among events is governed by the event ordering used. In event ordering, events can be executed in parallel if they are not comparable (concurrent). Therefore, the normalized event parallelism (Π_{norm}^{prob}) is defined as the

average number of concurrent events. For example, the two physical systems shown in Figure 3.2 produce different Π^{prob} (1.5 events / minute and 1.5 events / hour, respectively). However, their normalized event parallelism is the same as shown in Figure 3.3. The links in Figure 3.2 and 3.3 are defined based on Definition 2.11 (event order at the physical system layer).

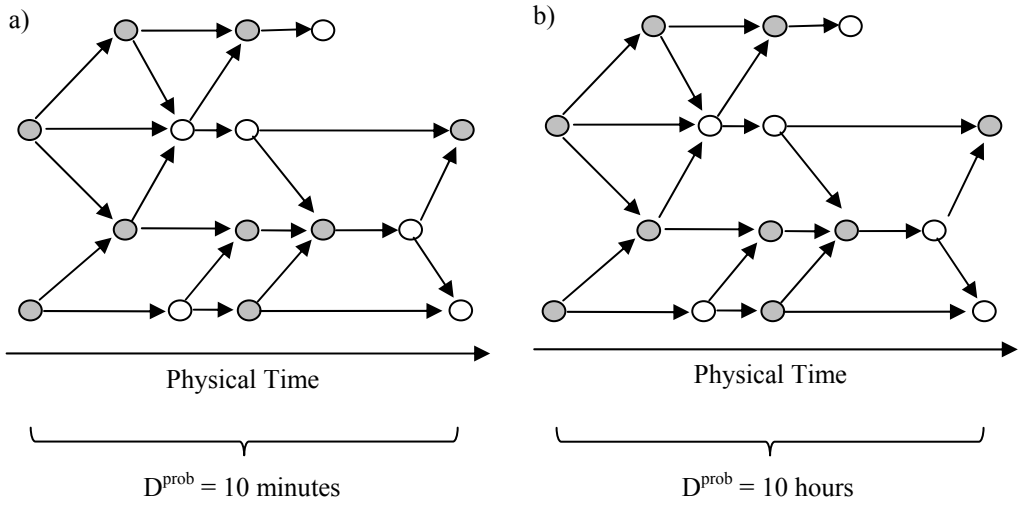


Figure 3.2: Two Cases of Event Execution at the Physical System Layer

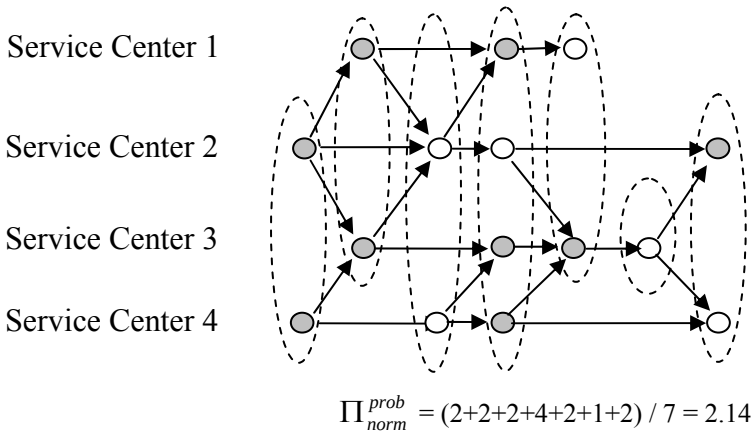


Figure 3.3: Normalized Event Parallelism at the Physical System Layer

The same normalization procedure is also applied to event orders at the simulation model layer. Figure 3.4 shows the event parallelism exploited by the partial event order at the simulation model layer (Definition 2.12).

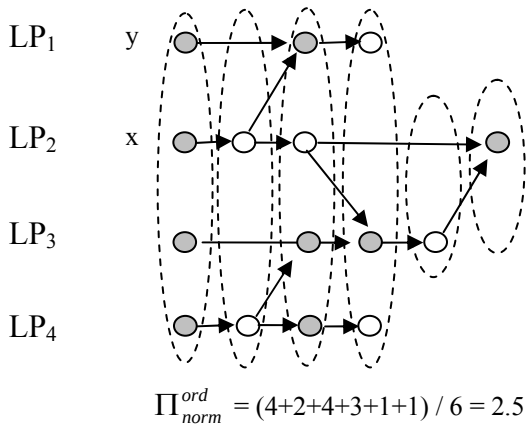


Figure 3.4: Event Execution at the Simulation Model Layer

At the simulator layer, the number of PPs is limited which affects the event ordering. For example, events *x* and *y* at the simulation model layer (Figure 3.4) are concurrent. At the simulator layer, LP_1 and LP_2 are mapped onto the same PP (Figure 3.5), hence only one of the two events can be executed at a time which means events *x* and *y* are comparable (decision on which event is executed first, depends on the scheduling policy used). Therefore, it is possible that two concurrent events at the simulation model layer are comparable at the simulator layer, due to the limitation in the number of processors.

After normalization, we can compare the normalized event parallelism at the three layers ($\Pi^{prob}_{norm} = 2.14$, $\Pi^{ord} = 2.5$, and $\Pi^{sync}_{norm} = 1.67$). It shows that event parallelism at the physical system can be exploited by the partial event order at the simulation model layer. Due to the limited number of processors, event parallelism at the simulator layer is less than the one at the simulation model layer.

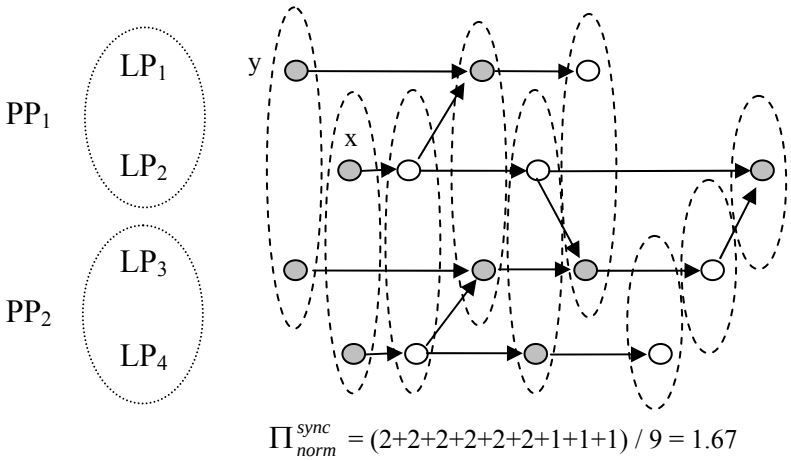


Figure 3.5: Normalized Event Parallelism at the Simulator Layer

3.3.5 Related Works

In this section, we show that a number of time performance analyses done by various researchers have been conducted at the three layers. Wagner and Lazowska noted that the presence of parallelism in the system being modeled does not imply the presence of the same degree of parallelism in the simulation of that system [WAGN89]. They clearly separated the parallelism at the physical system layer from the parallelism at the simulation model layer. They showed that the parallelism in the physical system (i.e., network of service center) is not the same as the parallelism in the simulation model (network of LPs) even if each service center is mapped onto a unique LP. One of the reasons is that the service time of an LP (i.e., the time required to execute an event) is different from the service time of a service center modeled by the LP (i.e., the time required to complete a service). Hence, the throughput of an LP at the simulation model layer and that of the service center modeled by the LP are different, which results in different upper bounds on parallelism.

Later, Shorey et al. built a comprehensive model for the upper bound on parallelism at the simulation model layer [SHOR97]. Further, Wang et al. showed that the causality constraint imposed at the simulation model layer also affects the parallelism of the simulation model [WANG00]. These works [WAGN89, SHOR97, WANG00] concentrate on the parallelism at the simulation model layer. The works may be extended to analyze parallelism at the other two layers by changing the unit of time.

Critical path analysis (CPA), introduced by Berry and Jefferson, is another widely known time performance analysis technique [BERR85]. Critical path time gives the lower bound on the completion time. Later, Jefferson showed that this is true only for conservative protocols [JEFF91]. Most researchers measure critical path time either at the simulation model layer where the event execution time is assumed to be constant [BERR85, LIN92] or at the simulator layer where the event execution time is measured directly from the simulator [JEFF91]. CPA may also be measured at the physical system layer where the event execution time in an LP reflects the service time at the service center that is modeled by the LP. Other time performance analyses measure speed-up [JHA96, BAJA99, BAGR99, SONG00, XU01, KIDD04], execution time [SOKO91, BAJA99, BAGR99, HUSS04], efficiency [JHA96], blocking time [SONG01], and wall-clock time per simulation time unit [DICK96, LIU99]. These metrics are commonly measured at the simulator layer.

3.4 Space Performance Analysis

Space performance refers to the amount of memory that is required to support a simulation. Memory is required when a simulation model is run on an execution

platform. Hence, the concept of memory requirement originates from the simulator layer. In this section, we attempt to extend the concept of memory requirement to the other two layers so that memory requirement at each layer can be analyzed independently. Further, time and space performance trade-off at each layer can also be studied.

We view a physical system as a queuing network of service centers. Therefore, we define the memory requirement at a physical system layer (M^{prob}) as the sum of the maximum queue size of each service center as shown in Equation 3.5:

$$M^{prob} = \sum_{i=1}^{m} \max_{0<t<D^{prob}} Q_i(t) \qquad (3.5)$$

where m is the number of service centers, $Q_i(t)$ is the queue size of service center i at physical time t; and D^{prob} is the observation period (in physical time units).

At the simulation model layer, based on the virtual time paradigm, we view a simulation model as a network of LPs and events are exchanged among LPs. Therefore, the memory requirement at this layer is defined as the sum of the maximum event list size of each LP as shown in Equation 3.6:

$$M^{ord} = \sum_{i=1}^{m} \max_{0<t<D^{ord}} L_i(t) \qquad (3.6)$$

where m is the number of LPs, $L_i(t)$ is the event list size of LP i at timestep t; and D^{ord} is the simulation duration (in timesteps).

At the simulator layer, we focus on the memory requirement that is used for synchronization purposes (M^{sync}). Since a sequential simulator does not need any global synchronization, its M^{sync} is zero. In a parallel simulator, LPs are mapped onto a number

of PPs. The types of memory overhead depend on the synchronization algorithm used. For examples, in the CMB protocol, M^{sync} is measured from the maximum size of the data structure used for storing null messages, and in the Time Warp protocol, M^{sync} is measured from the maximum size of the data structure used for storing past states, past messages, and anti-messages. The definition of M^{sync} is shown in Equation 3.7.

$$M^{sync} = \sum_{i=1}^{n} \max_{0<t<D^{sync}} B_i(t) \tag{3.7}$$

where n is the number of PPs, $B_i(t)$ is the size of the data structure used for storing memory overhead at PP i at wall-clock time t; and D^{sync} is the total execution time (in wall-clock time units).

3.4.1 Total Memory Requirement

In the previous section, we have looked at the memory requirement at three different layers independently. In this section, we study the total memory required to run a simulation.

In a physical system, when a job arrives at a busy service center, the job will join the queue at that service center. Similarly, in the simulation run, events will be stored in a data structure (such as a linked-list) to emulate the jobs waiting for resources in the physical system. The size of the data structure reflects the queue size of a service center in the physical system. Hence, the maximum queue size measured at the physical system layer and simulator layer is the same. Conclusively, M^{prob} is the same whether it is measured at the physical system layer or at the simulator layer.

In a simulator, future events are generated in advance to emulate the event occurrences in a physical system. These future events are stored and sorted in a data structure that is termed the future event list (FEL) in simulation. A parallel simulator may employ a set of event lists (ELs) instead of a global future event list. The event list size of an LP depends on the event incoming rate to the LP and the event execution rate of the LP [SHOR97]. Due to overheads at the simulator layer, the rates at the simulator layer are different from the simulation model layer. Therefore, M^{ord} may give a different value when it is measured at the simulation model layer and at the simulator layer.

Let M_{norm}^{ord} be the M^{ord} that is measured at the simulator layer. The total memory requirement (M^{tot}) can be defined as:

$$M^{tot} \approx M^{prob} + M_{norm}^{ord} + M^{sync} \tag{3.8}$$

Equation 3.8 is simple but less accurate. A more accurate model should include the memory architecture used to run a simulator because it affects the total memory requirement of the simulator. As Fujimoto noted, research in parallel simulation has focused on *shared memory* architecture and *distributed memory* architecture [FUJI00]. Preiss et al. [PREI95] and Fujimoto [FUJI00] noted that storage optimality is much more difficult to achieve on distributed memory architecture. In other words, the same simulator that is run on a distributed memory system requires significantly more memory than when it is run on a shared memory system.

Figure 3.6 shows the memory requirement for the shared memory architecture and the distributed memory architecture. In simulation, every service center is mapped onto a

unique LP and every k LPs are mapped onto one PP. Further, we assume that there are m service centers (hence m LPs) and n PPs, where $m = k \times n$.

Let:

- Q_i be the queue at service center i, and $||Q_i(t)||$ be the size of Q_i at wall-clock time t
- L_i be the event list at LP i, and $||L_i(t)||$ be the number of events in L_i at wall-clock time t
- B_i be the extra memory overhead at LP i, and $||B_i(t)||$ be the size of B_i at wall-clock time t.

The total memory required by a simulator running on a shared memory architecture (M^{shr}) and by one running on a distributed memory architecture (M^{dst}) are defined in Equations 3.9 and 3.10, respectively. D is the total execution time.

$$M^{shr} = \max_{0 \le t < D} \{ \sum_{i=1}^{m} (|| Q_i(t) || + || L_i(t) || + || B_i(t) ||) \} \quad (3.9)$$

$$M^{dst} = \sum_{i=1}^{n} \max_{0 \le t < D} \{ \sum_{j=1}^{k} (|| Q_{(i-1)k+j}(t) || + || L_{(i-1)k+j}(t) || + || B_{(i-1)k+j}(t) ||) \} \quad (3.10)$$

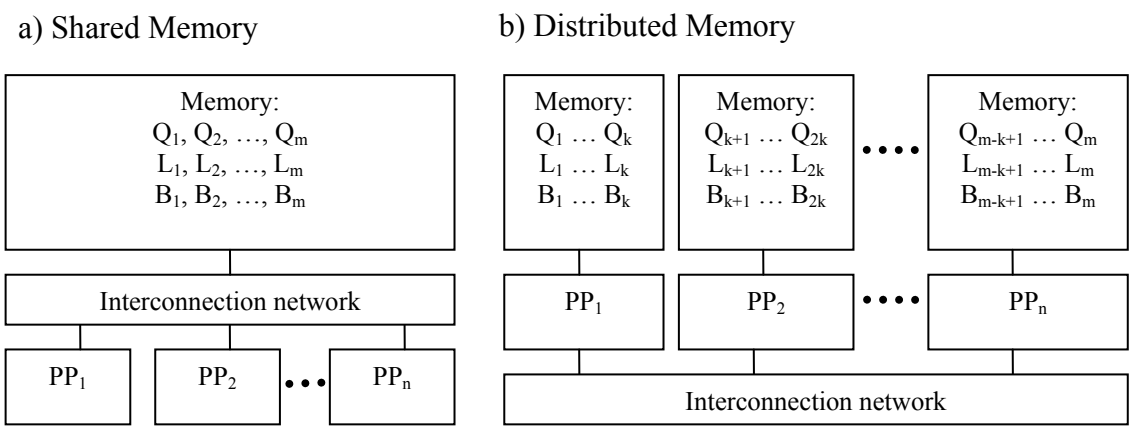


Figure 3.6: Shared Memory and Distributed Memory Architecture

As noted by Young et al., the average memory consumption shows how memory is utilized across processors in a simulation run [YOUN99]. A high average number of events in a system increases the probability of events that can be executed in parallel. The maximum value only shows the peak memory requirement. It captures the number of events at one point of time only; it does not tell about the expected number of events at any point of time. Hence, the average memory requirement can explain event parallelism better than the maximum memory requirement. However, if we want to know how much memory should be provided for the simulation, we should refer to the maximum memory requirement.

3.4.2 Related Works

We have shown in Section 3.1 that most works on space performance analysis concentrate on a specific protocol or memory architecture (at the simulator layer). They focus mainly on the memory management of various protocols.

The first comprehensive work on memory requirement comparison among different simulators was done by Lin and Preiss [LIN91]. They provided a thorough model for the memory requirement of a simulator running on shared memory architecture [LIN91]. The model covers the memory requirement of the sequential simulator, the CMB protocol and the Time Warp protocol. They focused more on the total memory requirement (for the shared memory architecture, i.e., M^{shr}). The memory for a sequential simulator consists of memory for state variables and an event list. The CMB protocol requires memory for state variables and event lists (they did not consider any memory

overhead). The Time Warp protocol requires memory for state variables (current and past), event lists, and anti-messages.

Preiss and Loucks noted that Lin's model is valid only for shared memory architecture [PREI95]. They noted that memory requirement for the distributed memory architecture should be based on the sum of LPs' maximum memory usage. The same definition is also suggested in [ZHAN01]. Further, Preiss and Loucks characterized the memory used in the Time Warp protocol into the following three components [PREI95]:

- *State Storage*: used to store various states (or state vectors).
- *Input Message Storage*: used to store the events that have been received by an LP.
- *Output Message Storage*: used to store copies of the events sent by the LP (for cancellation purposes).

The state storage here includes the present state and past states that have been saved in case an LP has to rollback. In our characterization, a past state is considered as memory overhead (M^{sync}). Output message storage also belongs to memory overhead. Input message storage refers to the memory that is allocated to store events in our characterization (M^{ord}). Li and Tropper [LI04] divided the memory consumed by Time Warp protocol into two: memory for state saving and memory for event lists. In our characterization the memory for event lists is M^{ord} and the memory for state saving is M^{sync}.

3.5 Strictness of Event Orderings

Different event orders impose different rules that regulate which events can be processed at any point of time. One event may have to be processed after another event. Therefore,

the degree of dependency among events is affected by the strictness of the ordering rules. We propose the relation *stricter* and a measure called *strictness* for comparing and quantifying the degree of event dependency of event orderings, respectively.

Event order R_1 is said to be stricter than event order R_2 if for any two events that have to be ordered one after another in R_2, they also have to be ordered one after another in R_1 but not vice versa. The relation stricter and the measure strictness depend only on the event dependency, i.e., whether two events have to be ordered one after another. It does not matter whether the first event occurs five minutes or five hours before the later event. Therefore, the relation stricter and the measure strictness are independent of time.

Events in a physical system are ordered based on their time of occurrences (see Definition 2.11). The same event order can be used at the simulation model layer such that if event *x* is ordered before event *y* at the physical system layer, event *x* is also ordered before event *y* at the simulation model layer. This implies that for the same set of events, both event orders have the same degree of event dependency, and therefore their strictness is the same even though they are from two different layers. Further, at the simulator layer, we can implement a simulator that executes events based on the same event order. Thus, if every LP is mapped onto one PP, the measured strictness will be the same as the strictness measured at the other two layers.

However, the number of PPs at the simulator can be less than the number of LPs. Therefore, it is possible that two concurrent events at the simulation model layer are not concurrent at the simulator layer as shown in Figure 3.4. In other words, an event order at the simulation model layer is less strict than its implementation at the simulator layer.

The same phenomenon is also reported in the memory consistency model [GHAR95]. The memory consistency model is less strict than its implementation in the real machine.

In the following subsections, we discuss the definition of stricter and strictness in more detail. It is followed by a strictness analysis to compare a number of event orders.

3.5.1 Definition of Strictness

To compare the degree of event dependencies among different event orders, we propose a relation *stricter*. The term stricter is borrowed from the memory consistency model [GHAR95, CULL99]. In the memory consistency model, the relation stricter is used to compare different models by considering the set of possible outcomes that is allowed by each model for a given set of instructions. In simulation event ordering, we consider the set of events that have to be executed one after another due to the ordering rules imposed by an event order for a given set of events. The definition of relation stricter is given in Definition 3.1; its properties are shown in Lemma 3.1.

Definition 3.1. *Let* (E, S_{R1}) *and* (E, S_{R2}) *be two event orderings on the same set of events E. Event order* R_1 *is* ***stricter*** *than* R_2 (*denoted by* $R_1 <_{\varsigma} R_2$) *if for any E,* $S_{R2} \subseteq S_{R1}$. *An event order* R_1 *is* ***incomparable*** *to event order* R_2 *if we can find two sets of events* E_1 *and* E_2, *such* $S_{R2} \subseteq S_{R1}$ *is true for* E_1 *but* $S_{R2} \subseteq S_{R1}$ *is not true for* E_2.

Lemma 3.1. *These are the properties of a stricter relation:*

1. *If* $R_1 <_{\varsigma} R_2$ *and* $R_2 <_{\varsigma} R_1$, *then* $R_1 = R_2$ (*anti-symmetric*).
2. *If* $R_1 <_{\varsigma} R_2$ *and* $R_2 <_{\varsigma} R_3$, *then* $R_1 <_{\varsigma} R_3$ (*transitive*).

Proof. The correctness of these properties can be proved from the definition of subset in set theory, i.e.:

1. If $S_{R1} \subseteq S_{R2}$ and $S_{R2} \subseteq S_{R1}$ then $S_{R1} = S_{R2}$. This implies that for any set of events E, R_1 and R_2 will produce the same set of comparable events. Therefore, R_1 and R_2 are the same event order.
2. If $S_{R1} \subseteq S_{R2}$ and $S_{R2} \subseteq S_{R3}$ then $S_{R2} \subseteq S_{R3}$. ❑

As the stricter relation is based on set inclusion, the strictness of an event order R (ς_R) can be quantified based on the number of elements in S_R as shown in Definition 3.2. For ease of comparison, $||S_R||$ is normalized with $||S_{tot}||$ because total event order or sequential order is the strictest event order (Lemma 3.4). This yields a value (ς_R) between zero when $||S_R||$ is zero and one when R is *total order.*

Definition 3.2. *The strictness of an event order R* (ς_R) *is defined as* $\frac{\|S_R\|}{\|S_{tot}\|}$ *where* $\|S_R\|$ *and* $\|S_{tot}\|$ *is the size of the set of comparable (or non-concurrent) events ordered by R and the total event order, respectively.*

Figure 3.7 shows a Hasse diagram of an event ordering with a set of events that occur between timestamp 10 and 16. The set of events E in Figure 3.7 is $\{ a_3^{10}, a_4^{11}, a_5^{11}, d_2^{12}, a_6^{13}, d_3^{13}, a_7^{14}, d_4^{15}, d_5^{15}, a_8^{16} \}$, hence $||E|| = 10$.

$|| S_{tot} || = ||E|| \times (||E||\text{-}1) / 2 = 45$

$S_R = \{(a_4^{11}, a_6^{13}), (a_6^{13}, d_4^{15}), (a_4^{11}, d_4^{15}), (a_3^{10}, d_2^{12}), (d_2^{12}, d_3^{13}), (d_3^{13}, a_8^{16}), (a_3^{10}, d_3^{13}),$
$(a_3^{10}, a_8^{16}), (d_2^{12}, a_8^{16}), (d_2^{12}, a_6^{13}), (a_3^{10}, a_6^{13}), (a_3^{10}, d_4^{15}), (d_2^{12}, d_4^{15}), (d_3^{13}, a_7^{14}),$

(a_3^{10}, a_7^{14}), (d_2^{12}, a_7^{14}), (a_3^{10}, d_5^{15}), (d_2^{12}, d_5^{15}), (d_3^{13}, d_5^{15}), (a_5^{11}, a_7^{14}), (a_7^{14}, d_5^{15}), (d_5^{15}, a_8^{16}), (a_5^{11}, d_5^{15}), (a_5^{11}, a_8^{16}), $(a_7^{14}, a_8^{16})\}$, hence $\| S_R \| = 25$

$\varsigma_R = \| S_R \| / \| S_{tot} \| = 0.56$

ς_R measures the ratio of the number of ordered pairs in R to the number of ordered pairs in total event order. Hence, $\varsigma_R = 0.56$ can be interpreted as: for a given set of events, the probability of two events are comparable is 0.56. Therefore, $\varsigma_R = 0$ implies that all events are concurrent and $\varsigma_R = 1$ implies that all events are comparable (sequential). In our experiment results presented in Chapter 4, ς_R is derived by summing up ς_R at every timestep and dividing it by the total number of timesteps.

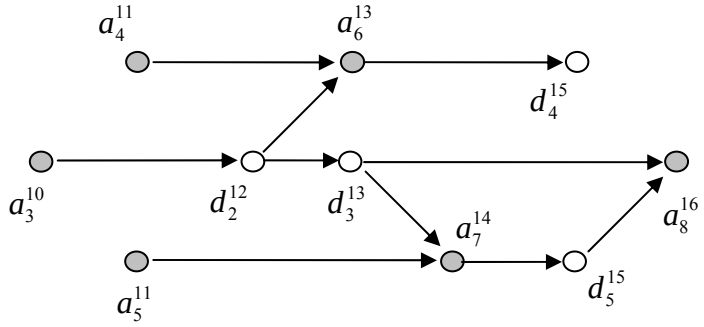


Figure 3.7: Hasse Diagram – Strictness

Theorem 3.1. *Let (E, S_{R1}) and (E, S_{R2}) be two event orderings, if event order R_1 is stricter than event order R_2 ($R_1 <_\varsigma R_2$), then $\varsigma_{R2} \leq \varsigma_{R1}$ for any set of events E, but not vice versa.*

Proof. If $R_1 <_\varsigma R_2$, then $S_{R2} \subseteq S_{R1}$ (from Definition 3.1). From set theory, we know that if $S_{R2} \subseteq S_{R1}$ then $\|S_{R2}\| \leq \|S_{R1}\|$. For a given E, $\|S_{tot}\|$ is constant, therefore $\frac{\|S_{R2}\|}{\|S_{tot}\|} \leq \frac{\|S_{R1}\|}{\|S_{tot}\|}$.

In other words, $\varsigma_{R2} \leq \varsigma_{R1}$ for any set of events E (from Definition 3.2).

Conversely, if $\varsigma_{R2} \leq \varsigma_{R1}$ for any set of events E, then $\frac{\|S_{R2}\|}{\|S_{tot}\|} \leq \frac{\|S_{R1}\|}{\|S_{tot}\|}$ (from Definition 3.2).

Since $\|S_{tot}\|$ is constant for a given E, then $\|S_{R2}\| \leq \|S_{R1}\|$. However, from set theory, $\|S_{R2}\| \leq \|S_{R1}\|$ does not imply that $S_{R2} \subseteq S_{R1}$. Hence, we cannot conclude that $R_1 <_{\varsigma} R_2$. Conclusively, $\varsigma_{R2} \leq \varsigma_{R1}$ is a necessary condition for $R_1 <_{\varsigma} R_2$, but it is not sufficient. The sufficient condition of $R_1 <_{\varsigma} R_2$ is $S_{R2} \subseteq S_{R1}$ as given in Definition 3.1. ❑

3.5.2 Strictness Analysis

Event order R_2 is stricter than event order R_1 implies that for any two distinct events x and y, if x is ordered before y in R_1, then x is also ordered before y in R_2, but not vice versa. Therefore, to prove whether an event order is stricter than another event order, we show that the ordering rule of one event order is a subset of the other event order. If the ordering rule of event order R_1 is a subset of event order R_2, then, definitely if x is ordered before y in R_1, x is also ordered before y in R_2. Using this approach, in the following theorems, we relate the event orders summarized in Figure 2.20.

Theorem 3.2. *Total event order is stricter than timestamp event order.*

Proof. Event x is ordered before event y if and only if $x.timestamp < y.timestamp$ for both event orders (Figure 2.20). However, if $x.timestamp = y.timestamp$, total event order will execute event x before y if and only if x has higher priority than y. In contrast, timestamp event order considers them incomparable. Therefore, it is possible that event x is ordered before event y in total event order but not in timestamp event order, showing that total event order is stricter than timestamp event order. ❑

Theorem 3.3. *Partial event order is stricter than the event ordering of the unsynchronized protocol.*

Proof. Figure 2.20 shows that the ordering rule of the unsynchronized protocol (i.e., event x is ordered before event y if $y.ante = x$) is a subset of the ordering rule of partial event order (i.e., event x is ordered before event y if $y.pred = x$ or $y.ante = x$). Consequently, partial event order is stricter than the event order of the unsynchronized protocol. ❑

The following two lemmas are used to prove the correctness of Theorem 3.4 to 3.8.

Lemma 3.2. $\{(x, y) \mid \forall x,y \in E \bullet y.ante = x\} \subseteq \{(x, y) \mid \forall x,y \in E \bullet x.lp \in SENDER(y.lp)$ and $x.timestamp + lookahead < y.timestamp\}$.

Proof. From the definition of the *SENDER* list and lookahead, if $y.ante = x$, then $x.lp$ must be in the *SENDER* list of $y.lp$ (i.e. $x.lp \in SENDER(y.lp)$) and the timestamp difference between x and y must be greater than the lookahead (i.e. $x.timestamp + lookahead < y.timestamp$). Therefore, $\{(x, y) \mid \forall x,y \in E \bullet y.ante = x\} \subseteq \{(x, y) \mid \forall x,y \in E \bullet x.lp \in SENDER(y.lp)$ and $x.timestamp + lookahead < y.timestamp\}$. However, the converse is not true. It is possible that $x.lp \in SENDER(y.lp)$ and $x.timestamp + lookahead < y.timestamp$ is true but $y.ante \neq x$. ❑

Lemma 3.3. $\{(x, y) \mid \forall x,y \in E \bullet x.timestamp + W < y.timestamp\} \subseteq \{(x, y) \mid \forall x,y \in E \bullet \lfloor x.timestamp/W \rfloor < \lfloor y.timestamp/W \rfloor\}$.

Proof. If $x.timestamp + W < y.timestamp$, then $\lfloor x.timestamp/W \rfloor < \lfloor y.timestamps/W \rfloor$, but if $\lfloor x.timestamp/W \rfloor < \lfloor y.timestamp/W \rfloor$, then $x.timestamp + W < y.timestamp$ may not be true. ❑

Theorem 3.4. *The event order of the CMB protocol is stricter than partial event order.*

Proof. Both have two ordering rules (Figure 2.20). The first rule is the same in both, i.e. event x is ordered before event y if $y.pred = x$. In the second rule, partial event order imposes event x to be ordered before event y if $y.ante = x$ whereas the CMB protocol imposes that event x be ordered before event y if $x.timestamp + lookahead < y.timestamp$ and $x.lp \in SENDER(y.lp)$. Lemma 3.2 shows that the second rule of partial event order is a subset of the second rule of the CMB protocol; therefore, the event order of the CMB protocol is stricter than partial event order. ❑

Theorem 3.5. *The event order of the BL protocol is stricter than that of the CMB protocol.*

Proof. The CMB protocol has two ordering rules and the BL protocol has three ordering rules (Figure 2.20). The first rule is the same in both protocols, i.e., event x is ordered before event y if $y.pred = x$. The second rule of the CMB protocol imposes that event x is ordered before event y if $x.timestamp + lookahead < y.timestamp$ and $x.lp \in SENDER(y.lp)$. Each LP in the BL protocol does not maintain a sender list; instead it maintains a lookahead list. Since $lookahead \neq \infty$ if $x.lp \in SENDER(y.lp)$ and $lookahead = \infty$ otherwise, its second rule, i.e., $x.timestamp + lookahead < y.timestamp$ imposes the same restriction as that of the CMB protocol. The BL protocol has one more rule, i.e.

event x is ordered before event y if $\lfloor x.timestamp/W \rfloor < \lfloor y.timestamp/W \rfloor$, which makes the event order of the BL protocol stricter than that of the CMB protocol. ❑

Theorem 3.6. *Timestamp event order is stricter than the event order of the BL protocol.*

Proof. Timestamp event order executes event x before event y if and only if $x.timestamp < y.timestamp$. On the other hand, the BL protocol executes event x before event y based on three rules: $y.pred = x$, $x.timestamp + lookahead < y.timestamp$, and $\lfloor x.timestamp/W \rfloor < \lfloor y.timestamp/W \rfloor$. These three rules can only be true if $x.timestamp < y.timestamp$. Therefore, if event x is ordered before event y in the BL protocol, event x is also ordered before event y in timestamp event order. However, if event x is ordered before event y in timestamp event order (i.e., $x.timestamp < y.timestamp$), they may be executed concurrently in the BL protocol – for instance, when they do not affect each other and are in the same time window. Hence, timestamp event order is stricter than the event order of the BL protocol. ❑

Theorem 3.7. *The event order of the BL protocol is stricter than that of the BTW protocol.*

Proof. Both protocols have three rules and their first two rules are the same, i.e., event x is ordered before event y if $y.pred = x$ or $\lfloor x.timestamp/W \rfloor < \lfloor y.timestamp/W \rfloor$ (Figure 2.20). The difference is in the third rule. The BL protocol imposes that event x be ordered before event y if $x.timestamp + lookahead < y.timestamp$ while the BTW protocol imposes that event x be ordered before y if $y.ante = x$. Lemma 3.3 shows that the latter is a subset of the former. Hence, the event order of the BL protocol is stricter than that of the BTW protocol. ❑

Theorem 3.8. *The event order of BTW protocol is stricter than partial event order.*

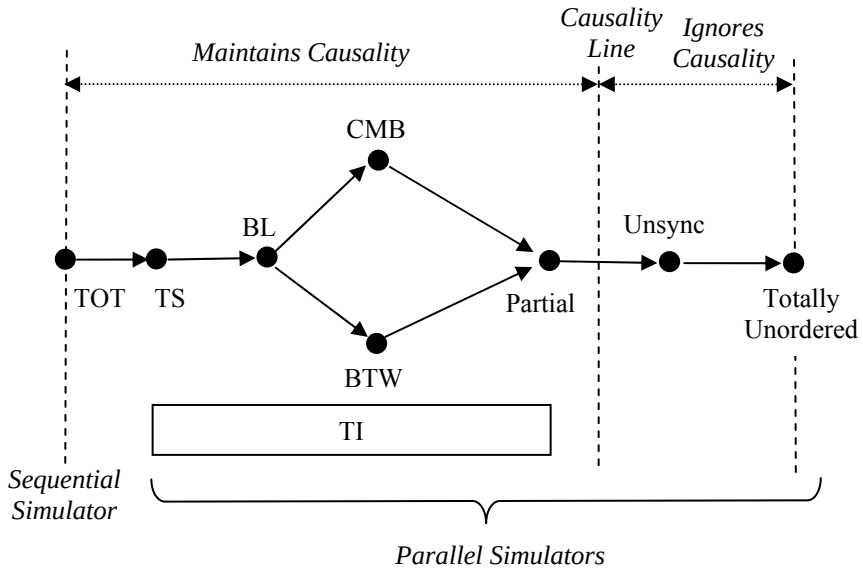
Proof. Figure 2.20 shows that the ordering rule of partial event order (i.e., event x is ordered before event y if $y.ante = x$ or $y.pred = x$) is a subset of the ordering rule of BTW protocol (i.e., event x is ordered before event y if $y.pred = x$ or $y.ante = x$ or $\lfloor x.timestamp/W \rfloor < \lfloor y.timestamp/W \rfloor$). Consequently, the event order of BTW protocol is stricter than partial event order. ❑

Lemma 3.4. *Total event order is the strictest event order.*

Proof. Definition 2.13 implies that in total event order, every two distinct events x and y from the given set E must be comparable (i.e., either x is ordered before y or y is ordered before x). Therefore, by definition, it is not possible to have an event order R where x and y are comparable in R but not in total event order. Consequently, total event order is the strictest event order. ❑

In summary, based on Theorems 3.2 – 3.8 and Lemma 3.4, we show the relationship among different event orders based on relation *stricter* in Figure 3.8. BL, BTW, CMB and Unsync refers to the event order of the BL protocol, the BTW protocol, the CMB protocol, and the unsynchronized protocol, respectively. An arrow from event order R_1 to another event order R_2 denotes that $R_1 <_{\varsigma} R_2$. The stricter relation is transitive, and the arrows can be traversed transitively. Total event order is the strictest, and to complete the spectrum of simulation event orders, we add one more point which corresponds to the totally unordered events. In contrast to total event order, all events in the totally unordered events are incomparable. Theoretically, totally unordered events can be

represented by event order (E, $\varnothing$). The strictness of time-interval event order varies, depending on the window size.



Tot	: Total	CMB	: Chandy-Misra-Bryant	Partial	: Partial
TS	: Timestamp	BL	: Bounded Lag	Unsync	: Unsynchronized
TI	: Time-interval	BTW	: Bounded Time Warp		

Figure 3.8: Strictness of Simulation Event Orders

Event orders on the left side of the causality line adhere to *lcc*. Hence, they produce correct simulation results. Some researchers have argued that it is acceptable to sacrifice correctness in favor of performance. This view is represented in the event orders located on the right side of the causality line. Sequential simulator implements total event order. The remaining event orders belong mainly to parallel simulators.

3.5.3 Strictness and Time Performance

Based on Dilworth's chain covering theorem (see Definition 2.4), an event ordering (E, S_R) can be decomposed into a number of smaller disjoint sets of events called *anti-chain* (shown as the dotted circles in Figure 3.9). Events in an anti-chain are incomparable /

concurrent. The average number of events in an anti-chain shows the average number of concurrent events (ψ). The number of anti-chains that form an event ordering (E, S_R) is called the *height* (denoted by H) of the event ordering (this term is borrowed from poset). Note that $\psi \times H = \| E \|$ for any event ordering (E, S_R).

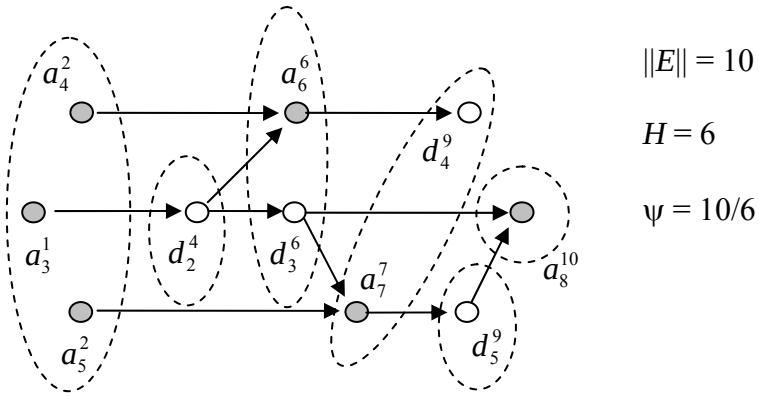


Figure 3.9: Event Ordering Formed by a Set of Anti-Chains

If the strictness of an event ordering is one, the implication is that events are executed sequentially in the event ordering. In other words, each of the anti-chains that form the event ordering comprises one event only (ψ = 1). The height of this event ordering is equal to the number of events in E ($H = \| E \|$). At the other extreme, if the strictness of an event ordering is zero, the implication is that all events are incomparable. Hence, this event ordering is formed by one anti-chain only (H = 1), and the number of events in this anti-chain is equal to the number of events in E ($\psi = \| E \|$). Theorem 3.9 shows the relation between the strictness (ς_R), the height (H_R), and the average number of concurrent events (ψ_R) of an event ordering R.

Theorem 3.9. *Let* (E, S_{R1}) *and* (E, S_{R2}) *be two event orderings, if* $\varsigma_{R1} < \varsigma_{R2}$, *then* $H_{R1} \leq H_{R2}$ *and* $\psi_{R1} \geq \psi_{R2}$.

Proof. Figure 3.10a and c show the Hasse Diagram of two event orderings. An event ordering (*E*, *R*) can also be represented as a graph, where the vertex represents an event, and an edge between event *x* and event *y* denotes that both are concurrent (see Figure 3.10b and d). This graph is termed concurrency graph [NEGG98]. Hence, the removal of an edge in a concurrency graph makes the event ordering stricter. In this graph an anti-chain forms a fully connected subgraph (i.e., all of its nodes are directly connected with an edge). If the edge belonging to a completely connected subgraph which is not connected to other subgraph is removed, then the removal of this edge will split the original anti-chain into two smaller anti-chains (for example, edge p-r in Figure 3.10b). Hence, it increases the number of anti-chains (*H*), and at the same time, reduces the average number of events in the anti-chains (ψ). Otherwise, the edge removal will not increase *H* (and hence, it will not decrease ψ) although the event ordering is stricter (for example edge r-s in Figure 3.10d). ❑

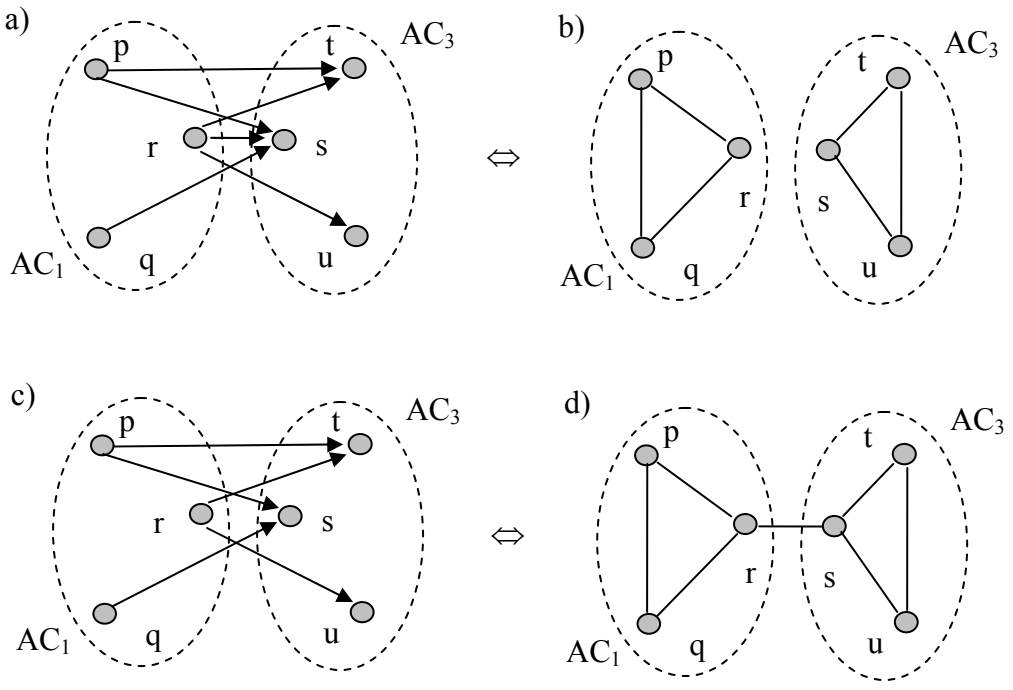


Figure 3.10: Strictness (ς_R) and Height (H_R)

Theorem 3.9 shows that the height of a stricter event ordering will never be less than that of a less strict event ordering. Similarly, the average number of concurrent events in a stricter event ordering will never be more than the average number of concurrent events in a less strict event ordering. A higher number of concurrent events implies that more events can be executed within the same duration. Conclusively, within the same duration, a stricter event order cannot execute more events than a less strict event order.

3.6 Summary

We have presented our framework for characterizing simulation performance that focuses on time and space performance. We characterize time (event parallelism) and space (memory requirement) performance into three layers: physical system, simulation model, and simulator as shown in Table 3.1. Event parallelism and memory requirement can be analyzed at each layer independent of the other layers. From its definition, event parallelism is dependent upon time and event dependency. Because the time units at different layers are different, event parallelism at different layers cannot be compared directly.

A physical system is viewed as a queuing network of service centers. It has a certain degree of inherent parallelism (Π^{prob}). At the simulation model layer, each service center is modeled as a logical process. Therefore, the physical system is modeled as a network of LPs and their interactions are modeled by exchanging events among LPs. Different event orderings can be used to exploit different degrees of parallelism (Π^{ord}). Enforcing event ordering at runtime incurs overhead which in turn results in performance loss. The

effective event parallelism at the simulator layer is called Π^{sync}. Π^{prob}, Π^{ord}, and Π^{sync} can be analyzed independently. A normalization process is required before we can compare the event parallelism at different layers.

Layers	Time	Space	Strictness
Physical System	$\Pi^{prob} = \frac{\|E\|}{D^{prob}}$	$M^{prob} = \sum_{i=1}^{m} \max_{0<t<D^{prob}} Q_i(t)$	$\varsigma = \frac{\|S_R\|}{\|S_{tot}\|}$
Simulation Model	$\Pi^{ord} = \frac{\|E\|}{D^{ord}}$	$M^{ord} = \sum_{i=1}^{m} \max_{0<t<D^{ord}} L_i(t)$	$<_{\varsigma}$
Simulator	$\Pi^{sync} = \frac{\|E\|}{D^{sync}}$	$M^{sync} = \sum_{i=1}^{n} \max_{0<t<D^{sync}} B_i(t)$ $M^{tot} \approx M^{prob} + M^{ord}_{norm} + M^{sync}$ $M^{shr} = \max_{0\le t<D} \{\sum_{i=1}^{m} (\| Q_{i,t} \| + \| L_{i,t} \| + \| B_{i,t} \|)\}$ $M^{dst} = \sum_{i=1}^{n} \max_{0\le t<D} \{\sum_{j=1}^{k} (\| Q_{(i-1)k+j,t} \| + \| L_{(i-1)k+j,t} \| + \| B_{(i-1)k+j,t} \|)\}$	

Table 3.1: Time and Space Performance Characterization

At the physical system layer, the memory requirement (M^{prob}) is derived from the maximum queue size at each service center. At the simulation model layer, the memory requirement (M^{ord}) is derived from the maximum event list size at each LP. At the simulator layer, the memory requirement (M^{sync}) is derived from the maximum size of the data structure used for storing overheads such as null messages in the CMB protocol. Finally, we define the total memory required to run a simulation as the sum of M^{prob},

M^{ord}_{norm} and M^{sync}. We have shown that measuring M^{prob} at the physical system and simulator layers will produce the same result. However, measuring M^{ord} at the simulation model and simulator layers may not produce the same result. More accurate measurement on total memory requirement should include the memory architecture used to run the simulation.

We have also proposed a relation called *stricter* ($<_{\varsigma}$) to compare the event dependency among different simulation event orderings. A new *strictness* (ς) measure has been proposed to quantify the degree of event dependency in an event ordering. Strictness is independent of time; therefore, it allows direct performance comparison across layers. We have shown that within the same duration, a stricter event order cannot execute more events than a less strict event order.

Chapter 4

Experimental Results

We have proposed a framework for characterizing simulation performance from the physical system layer to the simulator layer. In this chapter, we conduct a set of experiments to validate the framework and to demonstrate the usefulness of the framework in analyzing the performance of a simulation protocol.

To do experiment, first, we implement a set of measurement tools to measure the performance metrics at the three layers. Using these measurement tools, we test the framework. Then, we apply the framework to study the performance of Ethernet simulation.

Experiments that are used to measure performance metrics at the physical system layer and the simulation model layer are conducted on a single processor. Experiments using the SPaDES/Java parallel simulator (to measure performance metrics at the simulator layer) are conducted on a computer cluster of eight nodes connected via a Gigabit Ethernet. Each node is a dual 2.8GHz Intel Xeon with 2.5GB RAM.

The rest of this chapter is organized as follows. First, we discuss the measurement tools that we have developed for use in the experiments. Next, we test the proposed framework using an *open* and a *closed* system. After that, we discuss the application of the framework to study the performance of Ethernet simulation. We conclude this chapter with a summary.

4.1 Measurement Tools

To apply the proposed framework, we need tools to measure event parallelism, memory requirement, and event ordering strictness at the three different layers. We have developed two tools to measure these performance metrics as shown in Figure 4.1.

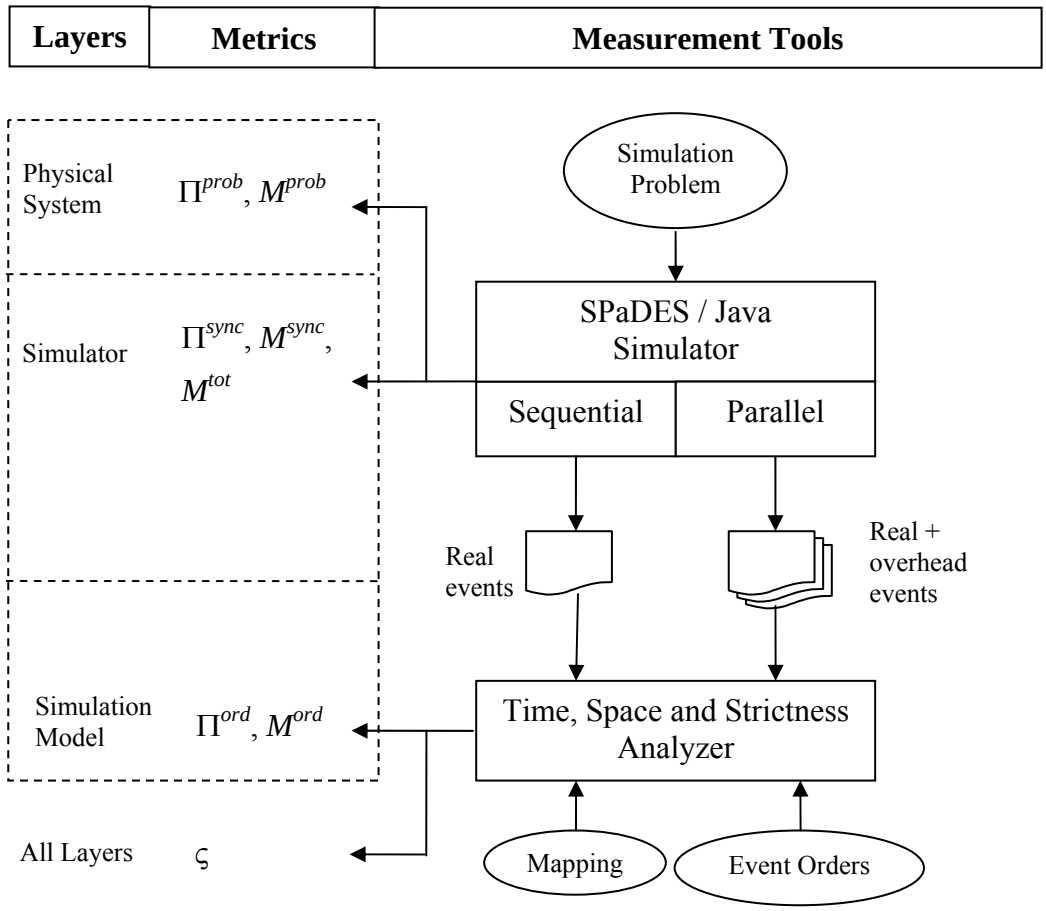


Figure 4.1: Measurement Tools

At the physical system layer, performance metrics (Π^{prob} and M^{prob}) are measured using the SPaDES/Java simulator. At the simulation model layer, the Time, Space and Strictness Analyzer (TSSA) is used to measure Π^{ord} and M^{ord}. The SPaDES/Java simulator is also used to measure performance metrics (Π^{sync}, M^{sync}, and M^{tot}) at the simulator layer. Depending on the inputs, TSSA can be used to measure event ordering strictness (ς) at the three layers. The details are discussed in the following sections.

4.1.1 SPaDES/Java Simulator

SPaDES/Java is a simulator library that supports a process-oriented worldview [TEO02A]. We extend the SPaDES/Java to support the event-oriented worldview and use this version in our experiments. The SPaDES/Java supports a sequential simulation and a parallel simulation based on the CMB protocol with demand-driven optimization [BAIN88].

The SPaDES/Java is used to simulate a simulation problem (physical system) and to measure event parallelism (Π^{prob}) and memory requirement (M^{prob}) at the physical system layer. Based on Equations 3.2 and 3.5, Π^{prob} and M^{prob} are derived from the number of events and maximum queue size, respectively. Therefore, instrumentation is inserted into the SPaDES/Java to measure the number of events and the maximum queue size of each service center.

The SPaDES/Java is also used to measure effective event parallelism (Π^{sync}), memory for overhead events (M^{sync}), and total memory requirement (M^{tot}) at the simulator layer.

Based on Equation 3.4, Π^{sync} is derived from the number of events and the simulation execution time. M^{sync} is derived from the size of the data structure used to store null messages (Equation 3.7). M^{tot} is derived from the size of the data structures that implement queues, event lists and buffers for storing null messages (Equation 3.8). Therefore, instrumentation is inserted into the SPaDES/Java simulator to measure the number of events, the simulation execution time, and the size of data structures that implement queues, event lists, and buffers for storing null messages.

The sequential execution of the SPaDES/Java produces a log file containing information on the sequence of event execution that will be used by TSSA to measure time and space performance at the simulation model layer as well as the strictness of different event orderings at the physical system and simulation model layers. The parallel execution of the SPaDES/Java produces a set of log files (one for every PP). Each log file contains information on the sequence of event execution (real and overhead) in a PP. These log files will be used by TSSA to measure the strictness of event ordering at the simulator layer.

4.1.2 Time, Space and Strictness Analyzer

We have developed the Time, Space and Strictness Analyzer (TSSA) to simulate different event orderings, to measure event parallelism (Π^{ord}) and memory requirement (M^{ord}) at the simulation model layer, and to measure event ordering strictness (ς) at the three layers.

To measure Π^{ord} and M^{ord}, TSSA needs two inputs, i.e., the log file generated by the sequential execution of the SPaDES/Java and the event order to be simulated. Every event executed by the SPaDES/Java is stored in a record in the log file, and the record number indicates the sequence when the SPaDES/Java executes the event. Each record also contains information on event dependency. Based on a given event ordering, TSSA simulates the execution of events and measures Π^{ord} and M^{ord}. Based on Equation 3.3, Π^{ord} is derived from the number of events and the simulation execution time (in timesteps). M^{ord} is derived from the maximum event list size of each LP. Therefore, TSSA is equipped with an instrumentation to measure the simulation execution time and the maximum event list size of each LP.

To measure the strictness of event ordering (ς) at the physical system layer and the simulation model layer, TSSA also needs the same inputs listed in the previous paragraph. At every iteration, TSSA reads a fixed number of events from the log file, and measures the strictness of the given event order based on Definition 3.2. This method is used because to measure the strictness of an event ordering with a large number of events is computationally expensive. Event ordering strictness is then derived by summing up the strictness at every iteration, and dividing it by the number of iterations.

To measure the strictness of event ordering (ς) at the simulator layer, TSSA requires the log files generated by the parallel execution of the SPaDES/Java simulator. Every event executed by the SPaDES/Java on a PP is stored in a record of a log file associated with the PP. This includes real events as well as overhead events (i.e., null messages). From

these log files, TSSA deduces the dependency among events and uses the same method as in the previous paragraph to measure event ordering strictness at the simulator layer.

4.2 Framework Validation

The objective of the experiments in this section is to validate our framework using an open system called Multistage Interconnected Network (MIN) and a closed system called PHOLD as the benchmarks. First, we validate each measurement tool that analyzes the performance at a single layer. The results are validated against analytical results. The validated tools are used to measure time and space performance at each layer independent of other layers. Next, we compare the time performance across layers in support of our theory on the relationship among the time performance at the three layers. Next, we analyze the total memory requirement. Finally, we measure the strictness of a number of event orderings in support of our strictness analysis in Chapter 3.

4.2.1 Benchmarks

We use two benchmarks:

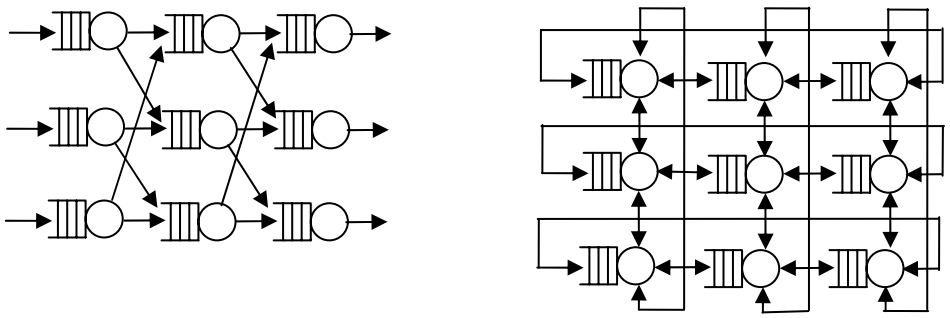
1. **Multistage Interconnected Network (MIN)**

   MIN is commonly used in a high speed switching system and it is modeled as an open system [TEO95]. MIN is formed by a set of stages; each stage is formed by the same number of switches. Each switch in a stage is connected to two switches in the next stage (Figure 4.2a). Each switch (except at the last stage) may send signals to one of its neighbors with equal probability. We model each switch as a service

center. MIN is parameterized by the number of switches ($n \times n$) and traffic intensity (ρ) which is the ratio between the arrival rate (λ) and the service rate (μ).

2. Parallel Hold (PHOLD)

PHOLD is commonly used in parallel simulation to study and represent a closed system with multiple feedbacks [FUJI90]. Each service center is connected to its four neighbors as shown in Figure 4.2b. PHOLD is parameterized with the number of service centers ($n \times n$) and job density (m). Initially, jobs are distributed equally among the service centers, i.e., m jobs for each service center. Subsequently, when a job has been served at a service center, it can move to one of the four neighbors with an equal probability.



a) MIN (3×3, ρ) b) PHOLD (3×3, m)

Figure 4.2: Benchmarks

Table 4.1 shows the total number of events that occur during an observation period of 10,000 minutes for both physical systems. All service centers in both MIN and PHOLD have the same service rates. The table shows that for MIN, the total number of events depends on the problem size and traffic intensity. From Little's law [JAIN91], at steady state condition, the number of jobs that arrive at a service center is equal to the job

arrival rate (λ) multiplied by the observation period (D). Since each job in MIN and PHOLD generates two events (arrival and departure), the number of events ($||E||$) at a service center is $||E|| = 2 \times \lambda \times D$. Since $\rho = \lambda / \mu$, $||E|| = 2 \times \rho \times \mu \times D$, where μ is the service rate of each service center. Therefore, for $n \times n$ service centers, the number of events can be modeled as:

$$||E|| = 2 \times \rho \times \mu \times D \times n \times n \tag{4.1}$$

Problem size	**MIN**		**PHOLD**	
	ρ	**Number of events**	***m***	**Number of events**
8×8	0.2	52,156	1	132,437
	0.4	103,981	4	222,384
	0.6	161,376	8	249,584
	0.8	220,431	12	261,675
16×16	0.2	205,964	1	525,411
	0.4	427,924	4	886,191
	0.6	640,067	8	999,156
	0.8	868,465	12	1,045,912
24×24	0.2	468,002	1	1,176,686
	0.4	946,792	4	1,991,927
	0.6	1,455,067	8	2,246,027
	0.8	1,941,903	12	2,351,078
32×32	0.2	824,529	1	2,093,555
	0.4	1,679,004	4	3,541,933
	0.6	2,536,016	8	4,004,896
	0.8	3,405,761	12	4,178,760

Table 4.1: Characteristics of the Physical System

The table also shows that the total number of events for PHOLD depends on the problem size and message density. All service centers in both MIN and PHOLD have the same service rates. Based on forced flow law, the arrival rate of a closed system is equal to its throughput [JAIN91]. Further, based on interactive response time law [JAIN91], the throughput of a closed system is a function of message density (m). Appendix C shows that message density has a logarithmic effect on traffic intensity in PHOLD. Hence, for

PHOLD, Equation 4.1 can be rewritten as the following equation where c_1 and c_2 are constants.

$$||E|| = 2 \times (c_1 \times \log (c_2 + m)) \times \mu \times D \times n \times n \quad (4.2)$$

4.2.2 Physical System Layer

The objective of this experiment is to measure time and space performance at the physical system layer (Π^{prob} and M^{prob}). First, we validate the SPaDES/Java simulator that is used to measure Π^{prob} and M^{prob}. We run the SPaDES/Java simulator to obtain the throughput and average queue size of the two physical systems (i.e., MIN and PHOLD). The results are validated against analytical results based on queuing theory and mean value analysis. The validation results show that there is no significant difference between the simulation results and the analytical results. The detail validation process can be seen from Appendix B. Next, we use the validated SPaDES/Java simulator to measure Π^{prob} and M^{prob} of the two physical systems. Figure 4.3 and Figure 4.4 show the event parallelism (Π^{prob}) of MIN and PHOLD, respectively. The detail experimental results in this chapter can be found in Appendix C.

Figure 4.3 shows that the event parallelism (Π^{prob}) of MIN varies with problem size ($n \times n$) and traffic intensity (ρ). The result confirms that a bigger problem size (more service centers) and higher traffic intensity increase the number of events per time unit (Equation 4.1). Figure 4.4 shows the effect of a varying problem size ($n \times n$) and message intensity (m) on the event parallelism (Π^{prob}) of PHOLD. The result confirms that a bigger problem size and higher message density increase the number of events that occur per unit of time (Equation 4.2).

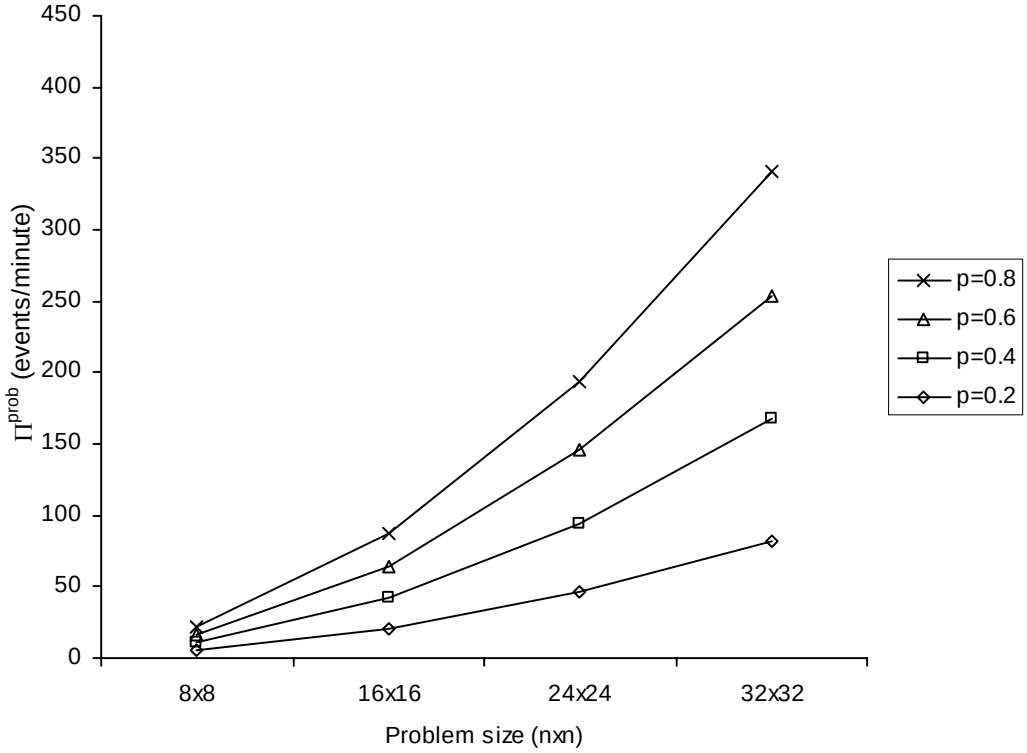


Figure 4.3: Π^{prob} – MIN ($n \times n$, ρ)

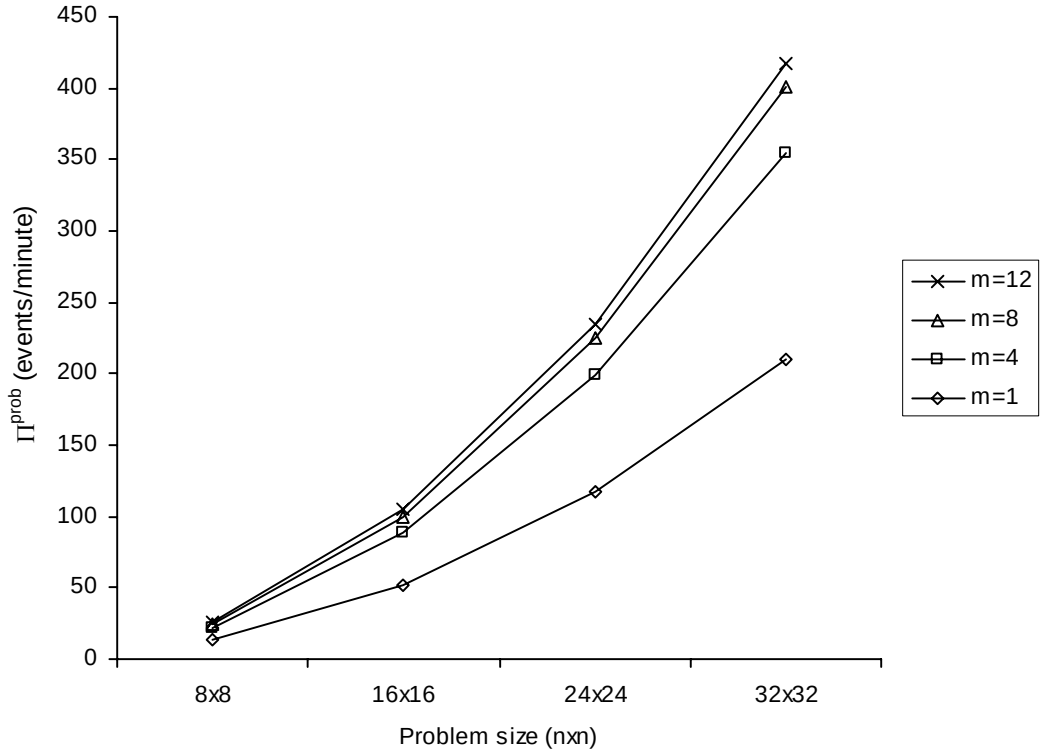


Figure 4.4: Π^{prob} – PHOLD ($n \times n$, m)

The memory requirement of the physical system MIN (M^{prob}) under a varying problem size ($n \times n$) and traffic intensity (ρ) is shown in Figure 4.5. The figure suggests that M^{prob} depends on problem size and traffic intensity. As shown in Chapter 3, we derive M^{prob} from the queue size at each service center. Hence, an increase in the number of service centers (problem size) increases M^{prob}. The same observation can also be made at PHOLD (Figure 4.6).

In MIN, high traffic intensity means that the service centers have to cope with many jobs. Similarly, in PHOLD, high message density indicates that the system has more jobs to execute. Consequently, a physical system with higher traffic intensity or message density requires more memory because the size of its queues is longer.

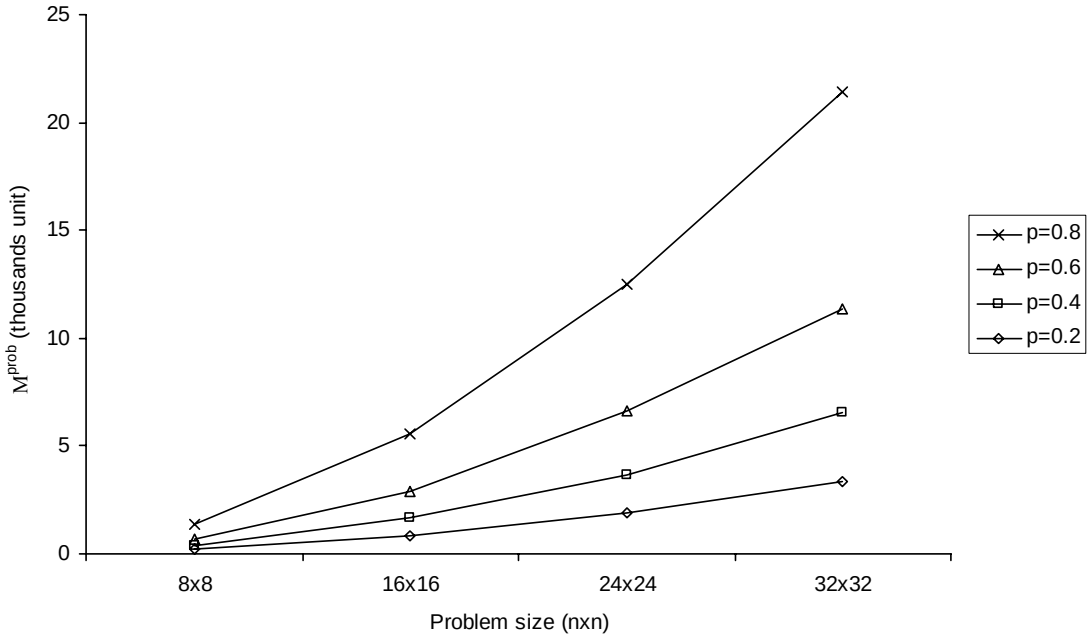


Figure 4.5: M^{prob} – MIN ($n \times n$, ρ)

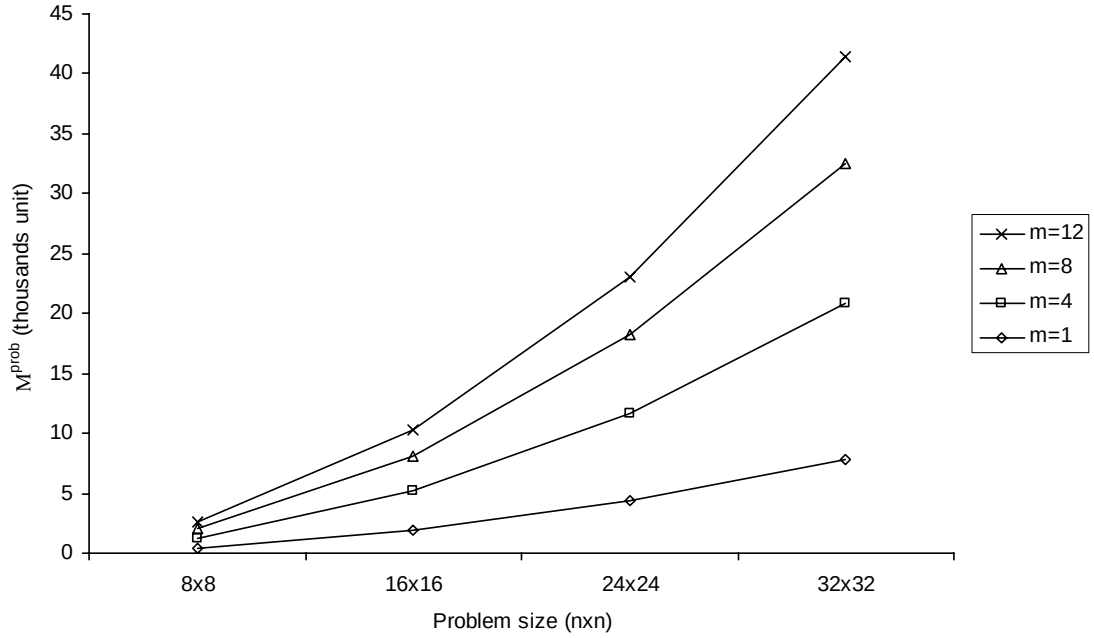


Figure 4.6: M^{prob} – PHOLD ($n \times n$, m)

4.2.3 Simulation Model Layer

The objective of this experiment is to measure the time and space performance of different event orderings at the simulation model layer (Π^{ord} and M^{ord}). First, we validate TSSA, and then use the validated TSSA to measure event parallelism exploited by different event orders (Π^{ord}) and their memory requirement (M^{ord}).

Wang et al. developed an algorithm to predict the upper bound of model parallelism (or Π^{ord} in our framework) [WANG00]. Therefore, we validate the parallelism of partial event ordering produced by our TSSA against the result of the algorithm. The results show that the algorithm gives an upper bound on Π^{ord} produced by TSSA. The detail is given in Appendix A.

Next, we use the validated TSSA to measure Π^{ord} and M^{ord}. Figure 4.7 and Figure 4.8 show that Π^{ord} depends on problem size ($n \times n$), traffic intensity (ρ), and the event order used.

A physical system with a bigger problem size and higher traffic intensity would have to handle more events within the same duration than a physical system with a smaller problem size and lower traffic intensity. Hence, more events can potentially be processed at the same time. At the same time, different event orders impose different ordering rules which also affect the number of events that can be executed at the same time. The result confirms that for the same duration, a stricter event order will never execute more events than the less strict event order (see Theorem 3.9). In this open system example, the partial event order and the CMB event order exploit almost the same amount of parallelism; therefore, only one line can be seen from Figure 4.7 and Figure 4.8.

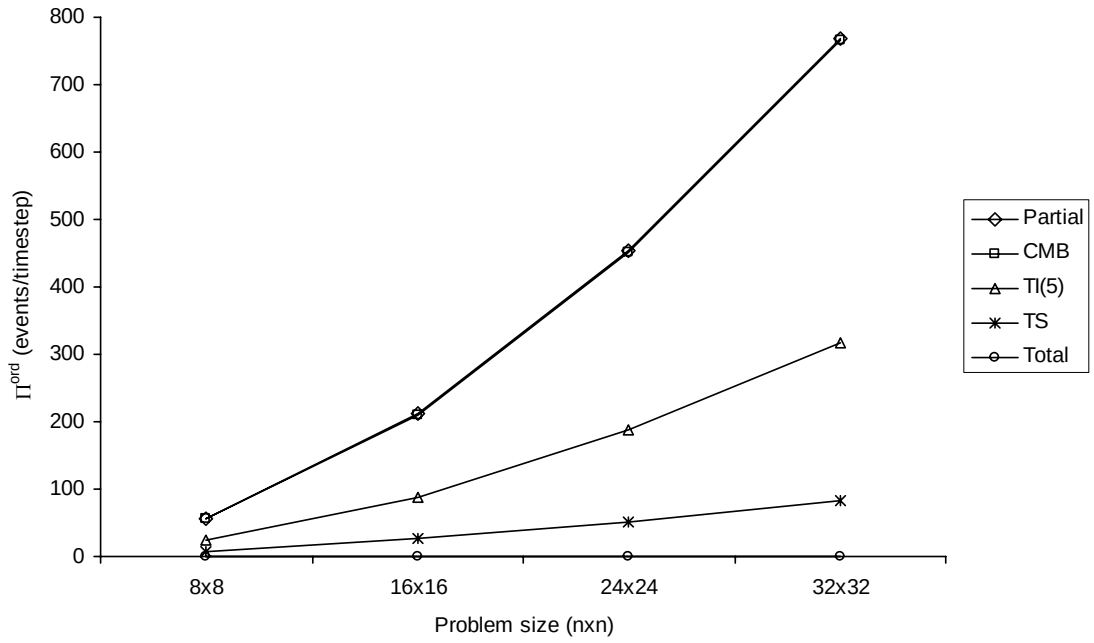


Figure 4.7: Π^{ord} – MIN ($n \times n$, 0.8)

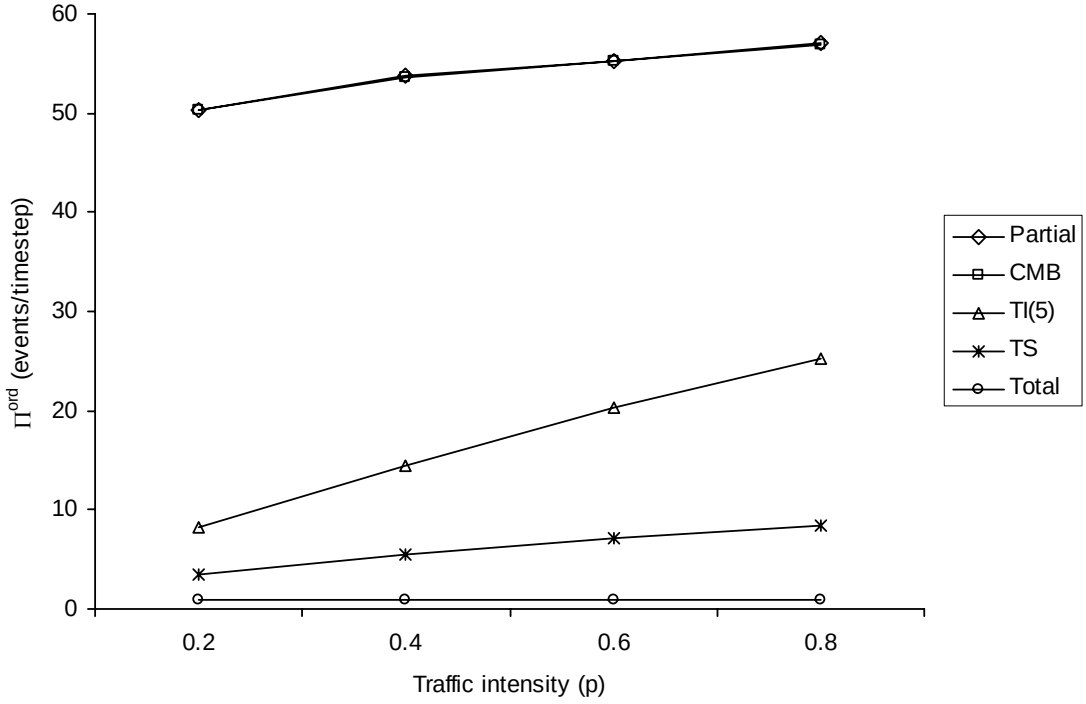


Figure 4.8: Π^{ord} – MIN (8×8, ρ)

Figure 4.9 and Figure 4.10 show the event parallelism (Π^{ord}) of different event orders in simulating PHOLD. The result shows that Π^{ord} varies with the problem size ($n \times n$), message density (m), and event order. The problem size and event order affect Π^{ord} for the same reason as in the open system example. An increase in message density (m) improves parallelism (Π^{ord}). This is because high message density increases the probability that each LP has some events to process at any given time. The improvement levels off eventually when each LP has an event to process at all times. The result also confirms that for the same duration, a stricter event order will never execute more events than a less strict event order (Theorem 3.9).

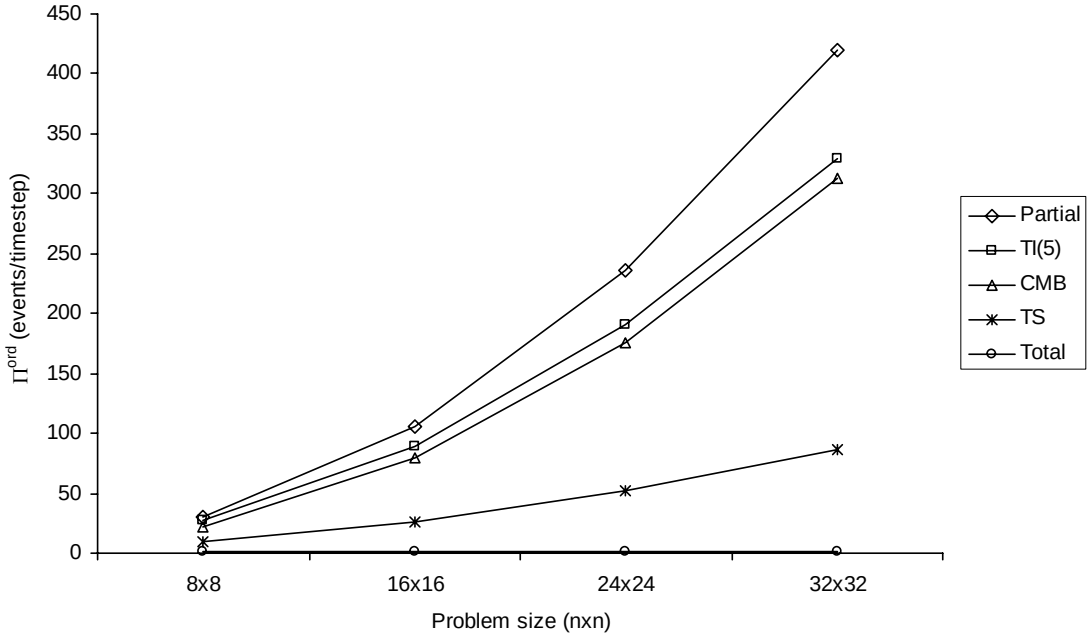


Figure 4.9: Π^{ord} – PHOLD ($n \times n$, 4)

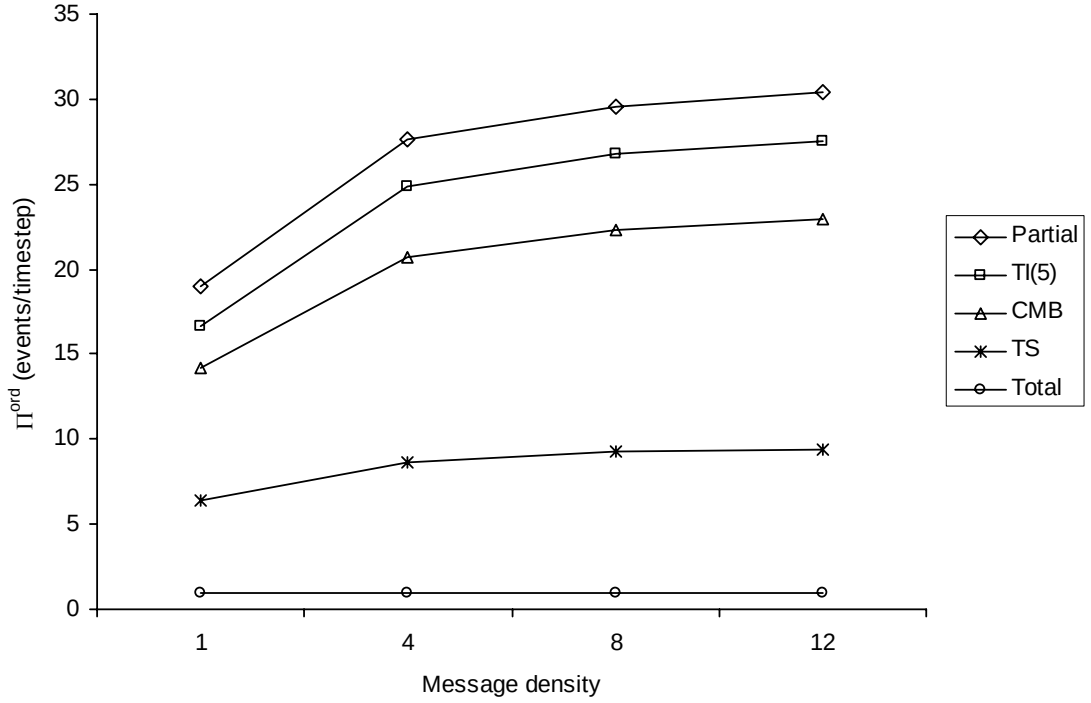


Figure 4.10: Π^{ord} – PHOLD (8×8, m)

We can observe from Figure 4.8 and Figure 4.9 that the event parallelism of CMB is better than that of TI(5) for the MIN problem, but the event parallelism of TI(5) is better

than that of CMB for the PHOLD problem. This is because time-interval event order is not comparable to the event order in CMB protocol as shown in Figure 3.8. Therefore, it is possible that time-interval event order can exploit more parallelism than the event order of CMB protocol at some problems but exploiting less parallelism at other problems.

We can also observe that the same event order may exploit different degrees of Π^{ord} from two different physical systems with the same Π^{prob}. Figure 4.3 and Figure 4.4 show that for the same problem size, the inherent event parallelism (Π^{prob}) of MIN with $\rho = 0.8$ is not significantly different from the inherent event parallelism of PHOLD with $m = 4$ (this is also supported by the analytical results shown in Appendix C). However, the same event order exploits more event parallelism at the simulation model layer (Π^{ord}) when it is used in MIN than when it is used in PHOLD (compare Figure 4.7 and Figure 4.9). This is caused by the difference in the topology of the two physical systems. At the simulation model layer, we can execute events at different LPs in parallel as long as they are independent. MIN generates less dependent events than PHOLD because of the multiple feedbacks in PHOLD. Therefore, at the simulation model layer, the same event order can exploit more parallelism (Π^{ord}) from MIN than PHOLD.

Table 4.2 shows the (maximum) memory requirement (M^{ord}) of different event orders in simulating MIN, and Table 4.3 shows the respective average memory requirement. As in Π^{ord}, the memory requirement (M^{ord}) varies with the problem size ($n \times n$), traffic intensity (ρ), and the event order used. More events occur within the same duration in a system with a bigger problem size and higher traffic intensity. Hence, more memory is required

to store these events. A less strict event order also tends to exploit more parallelism (Π^{ord}) than a stricter one.

Event Order	Problem size (n×n)			
	8×8	16×16	24×24	32×32
Partial	2,145	8,151	16,516	27,722
CMB	2,165	8,198	16,642	27,909
TI(5)	302	1,192	2,646	4,652
TS	92	301	656	1,102
Total	88	296	640	1,093

a) ρ = 0.8

Event Order	Traffic intensity (ρ)			
	0.2	0.4	0.6	0.8
Partial	1,073	1,551	1,964	2,145
CMB	1,076	1,556	1,970	2,165
TI(5)	185	226	259	302
TS	37	56	75	92
Total	35	53	73	88

b) Problem Size = 8×8

Table 4.2: M^{ord} – MIN

Event Order	Problem size (n×n)			
	8×8	16×16	24×24	32×32
Partial	879	2,822	5,164	7,766
CMB	888	2,845	5,234	7,903
TI(5)	73	268	589	1,016
TS	68	253	561	975
Total	64	245	376	958

a) ρ = 0.8

Event Order	Traffic intensity (ρ)			
	0.2	0.4	0.6	0.8
Partial	371	625	756	879
CMB	375	629	763	888
TI(5)	23	38	55	73
TS	22	36	52	68
Total	21	36	50	64

b) Problem Size = 8×8

Table 4.3: Average Memory Requirement – MIN

The (maximum) memory requirement (M^{ord}) and average memory requirement of different event orders in simulating PHOLD are shown in Table 4.4 and Table 4.5, respectively.

Table 4.4 shows that as the message density gets higher ($m \geq 8$), the value of M^{ord} tends to converge to the same value. The explanation is as follows. From the extreme values theory, the probability that a maximum number of events will exceed a threshold depends on the value of the threshold, the average number of events, and the standard deviation [COLE01]. A high threshold value, low average number of events, and narrow standard deviation result in a smaller probability. In PHOLD, initially, $n \times n \times m$ events are

distributed evenly among $n \times n$ LPs (m events per LP). This sets a threshold value of m. Therefore, the probability of the maximum number of events in each LP exceeding m depends on the average number of events and its standard deviation. For partial event ordering with a large m = 8 and 12, the average number of events per LP is only 1.5 (97/64) and 1.6 (103/64), respectively. The standard deviation is only 0.16 and 0.65, respectively. Therefore, statistically, as we increase m, it becomes more unlikely that the maximum number of events per LP will exceed m. It is even less likely for the less strict event orderings.

Event Order	Problem size (n×n)			
	8×8	16×16	24×24	32×32
Partial	468	1,878	4,245	7,584
CMB	305	1,237	2,836	4,892
TI(5)	321	1,299	2,736	4,952
TS	256	1,024	2,304	4,096
Total	256	1,024	2,304	4,096

a) m = 4

Event Order	Traffic intensity (ρ)			
	1	4	8	12
Partial	359	468	532	770
CMB	252	305	514	770
TI(5)	261	321	515	770
TS	64	256	512	768
Total	64	256	512	768

b) Problem Size = 8×8

Table 4.4: M^{ord} – PHOLD

Event Order	Problem size (n×n)			
	8×8	16×16	24×24	32×32
Partial	84	338	761	1354
CMB	66	260	580	1026
TI(5)	67	265	596	1060
TS	62	245	551	979
Total	58	229	512	912

a) m = 4

Event Order	Traffic intensity (ρ)			
	1	4	8	12
Partial	45	84	97	103
CMB	39	66	75	78
TI(5)	39	67	76	80
TS	37	62	69	73
Total	34	58	65	67

b) Problem Size = 8×8

Table 4.5: Average Memory Requirement – PHOLD

Table 4.5 shows that the average memory requirement depends on the problem size ($n \times n$) and event order for the same reason as in the MIN example. Message density (m)

also affects the average memory requirement because a higher message density implies that more events are generated.

4.2.4 Simulator Layer

In this section, we measure performance metrics (Π^{sync} and M^{sync}) at the simulator layer. We use the SPaDES/Java simulator in this experiment. As discussed in Chapter 1, many factors affect the performance of a simulator at runtime. In this experiment, we do not attempt to study all factors that affect Π^{sync} and M^{sync}, but we demonstrate how performance is measured at the simulator layer so as to complete our three layered performance characterization.

We map a number of service centers (each is modeled as a logical process) onto a physical processor (PP). To reduce the null message overhead, logical processes (LPs) that are mapped onto the same PP communicate via shared memory. Java RMI is used for inter-processor communication among LPs that are mapped onto different processors. We run our SPaDES/Java parallel simulator on four and eight PPs. The results are shown in Figure 4.11 and Figure 4.12.

Figure 4.11 shows that effective event parallelism (Π^{sync}) is affected by the number of LPs. For the same number of PP, the result shows that an increase in the number of LPs increases the exploited parallelism. This can be explained by comparing Figure 4.3 and Figure 4.12a. Both figures show that an increase in the number of LPs increases the number of useful events and null messages at different rates. The rate of increase in the useful events is higher than that of the null messages. Therefore, the proportion of time

that is spent by the processors to execute useful events increases as the number of LPs increases. Consequently, it increases the exploited parallelism.

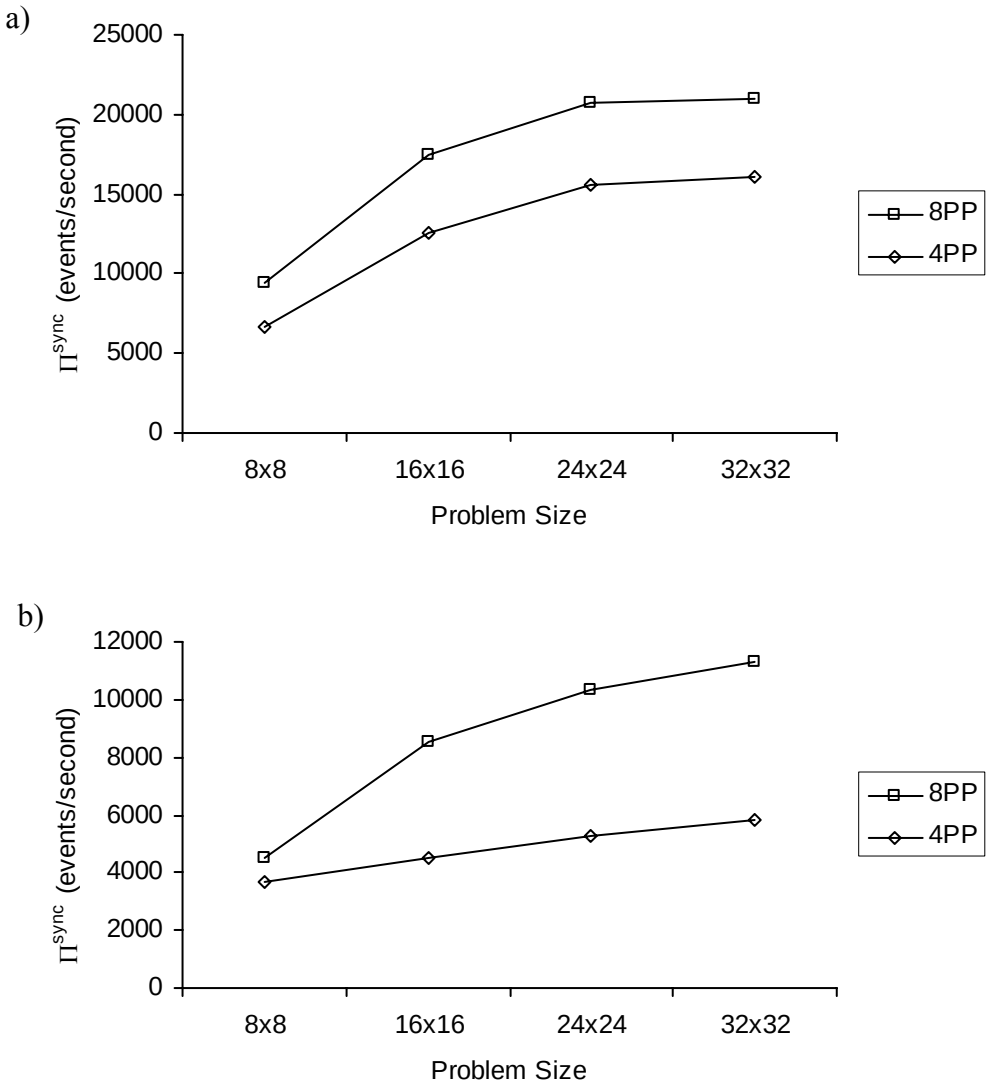


Figure 4.11: Π^{sync} – a) MIN ($n \times n$, 0.8) and b) PHOLD ($n \times n$, 4)

The experimental also shows that Π^{sync} is affected by the number of PPs. An increase in the number of PPs increases computing power so that less time is spent in executing useful events. At the same time, it increases the number of null messages to synchronize more PPs. The result shows that an increase from four PPs to eight PPs improves the parallelism because the reduction of time for executing useful events are higher than the

additional time to execute extra null messages. Since the number of null messages increases exponentially with the number of PPs (Figure 12), further increase in the number of PPs will eventually decrease the exploited event parallelism.

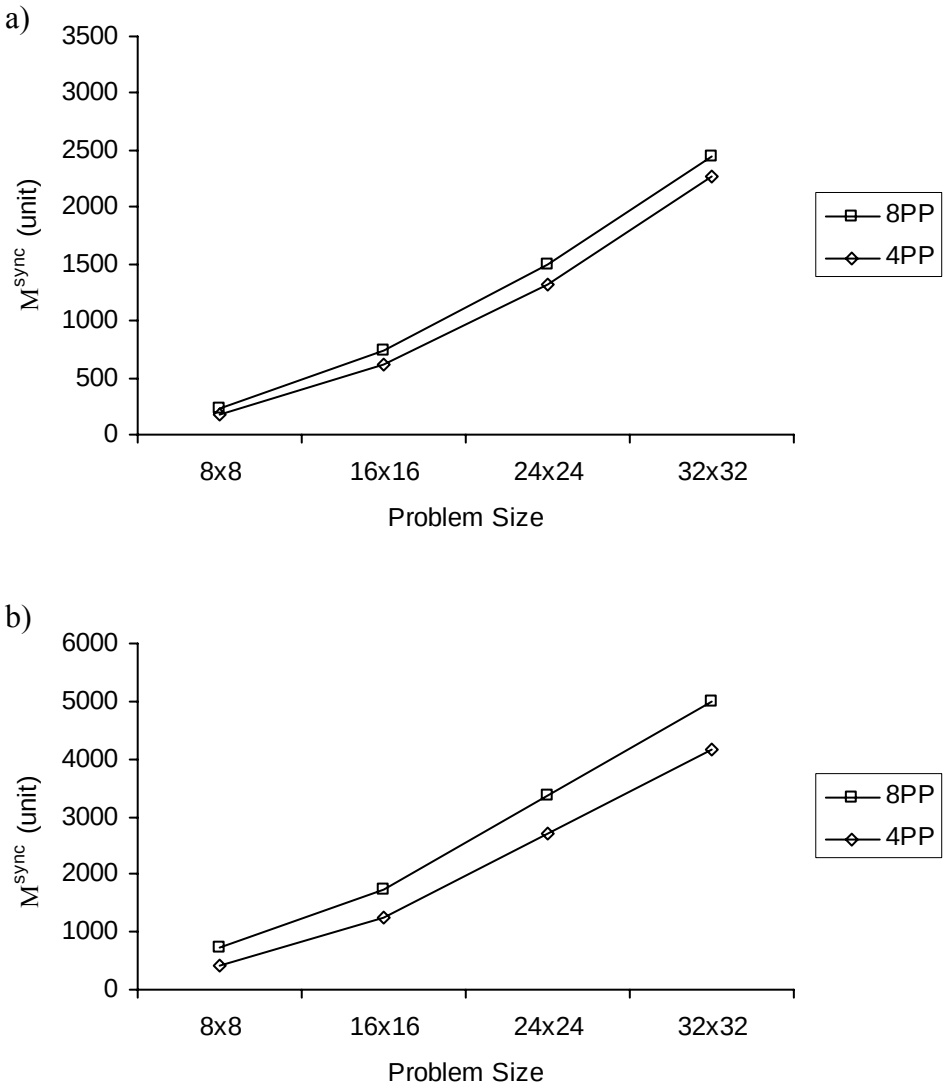


Figure 4.12: M^{sync} – a) MIN ($n \times n$, 0.8) and b) PHOLD ($n \times n$, 4)

In the CMB protocol, M^{sync} is derived from the maximum number of null messages for all PPs. For the same problem size, an increase in the number of PPs increases the number of null messages because more overhead is needed to synchronize more PPs. For the same number of PPs, an increase in the problem size (i.e., the number of LPs)

may result in more communication channels among LPs (at different PPs). This increases the number of null messages that must be sent. The result shows that more null messages are generated in PHOLD than in MIN because of the feedback in PHOLD.

4.2.5 Parallelism Analysis

In this section, we first show the parallelism profile exploited by the SPaDES/Java for the two benchmarks. Next, we normalize event parallelism using the method that has been explained in Chapter 3, and compare the normalized event parallelism.

The parallelism profile of MIN and PHOLD are shown in Figure 4.13 and Figure 4.14, respectively. The horizontal axis is the wall-clock time, and the vertical axis is event parallelism (the number of events executed per second). The profiles show that MIN has more parallelism than PHOLD.

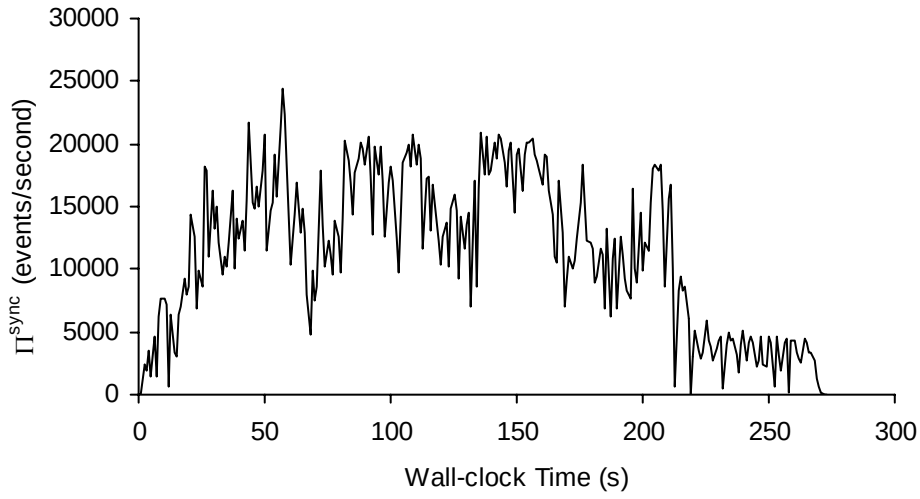


Figure 4.13: Parallelism Profile – MIN (32×32, 0.8)

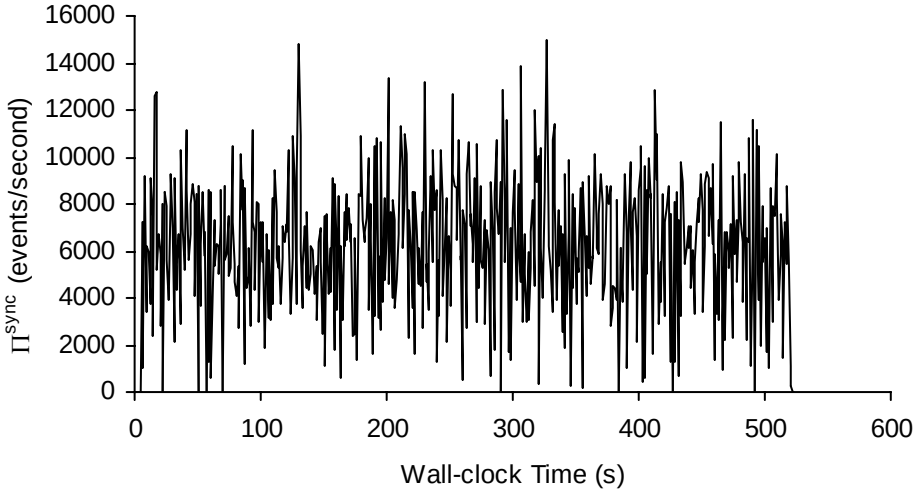


Figure 4.14: Parallelism Profile – PHOLD (32×32, 4)

Next, we normalize and compare event parallelism at the three layers. The results are shown in Figure 4.15 and Figure 4.16. The normalization results are consistent with the previous results for each layer. First, the normalized event parallelism increases as the problem size increases. Second, at the simulation model layer, for the same level of Π^{ord}, CMB event order exploits more parallelism from MIN than PHOLD. Third, at the simulator layer, the SPaDES/Java exploits more parallelism when it is used to simulate MIN than when it is used to simulate PHOLD.

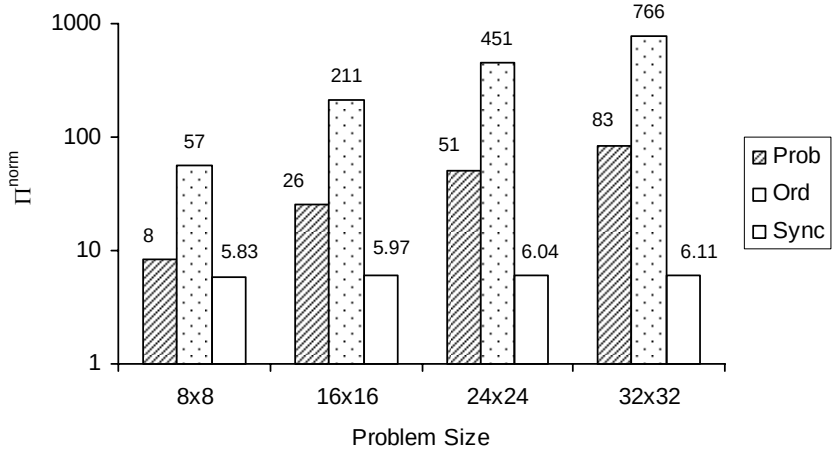


Figure 4.15: Normalized Parallelism– MIN (n×n, 0.8)

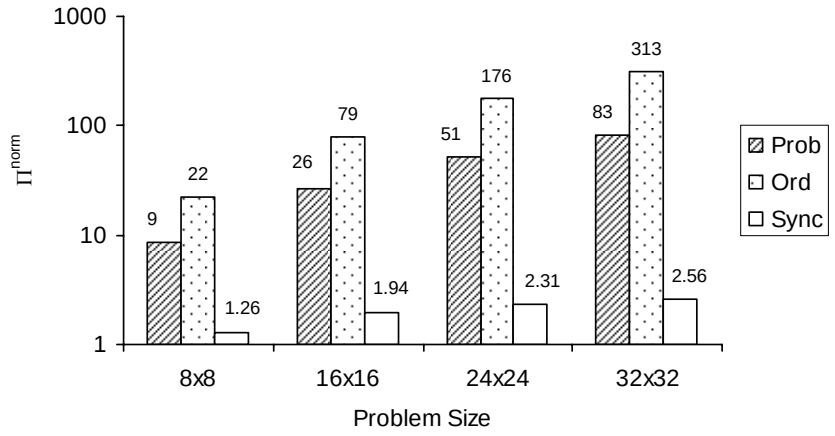


Figure 4.16: Normalized Parallelism– PHOLD ($n \times n$, 4)

The comparison results show that event parallelism at the physical system layer can be exploited by the CMB event order at the simulation model layer. However, the implementation overheads reduce the event parallelism at the simulator layer.

4.2.6 Total Memory Requirement

In this section, we analyze the total memory required by the SPaDES/Java to simulate the two benchmarks. First, we show the memory profiles for simulating MIN and PHOLD in Figure 4.17 and Figure 4.18, respectively. The horizontal axis shows the wall-clock time in 100ms (logarithmic scale). We choose 100ms so that we can observe the fluctuation more accurately. The vertical axis shows the required memory in unit (i.e., the number of entries in the data structures that implement queues, event lists and buffers for storing null messages). The profiles show that the queues (that are used to derive M^{prob}) are initially empty and increase until a certain level. In MIN, an event is scheduled for each LP in the first stage (leftmost column in Figure 4.2a). Therefore, initially, for $n \times n$ MIN, n events are in the event lists (that are used to derive M^{ord}). The profile shows that the total number of events in the event lists increases until a certain

level. In PHOLD, m events are scheduled for each LP before the simulation starts. Therefore, initially, for $n \times n$ PHOLD, more events ($n \times n \times m$) are in the event lists than in MIN. The profile shows that the total number of events in the event lists decreases until a certain level. This confirms the memory profile reported in [TEO01]. The profiles show that the null message population (that are used to derive M^{sync}) in PHOLD is higher than that in MIN.

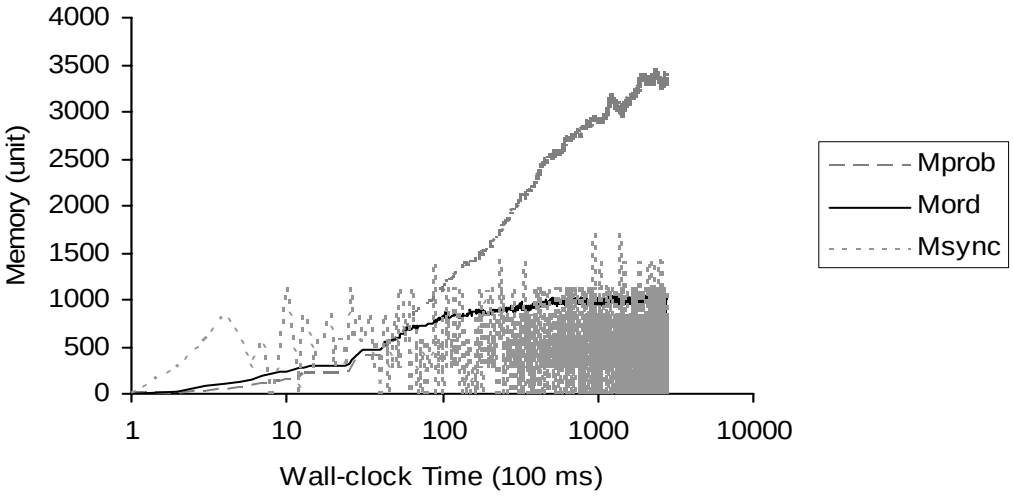


Figure 4.17: Memory Profile – MIN (32×32, 0.8)

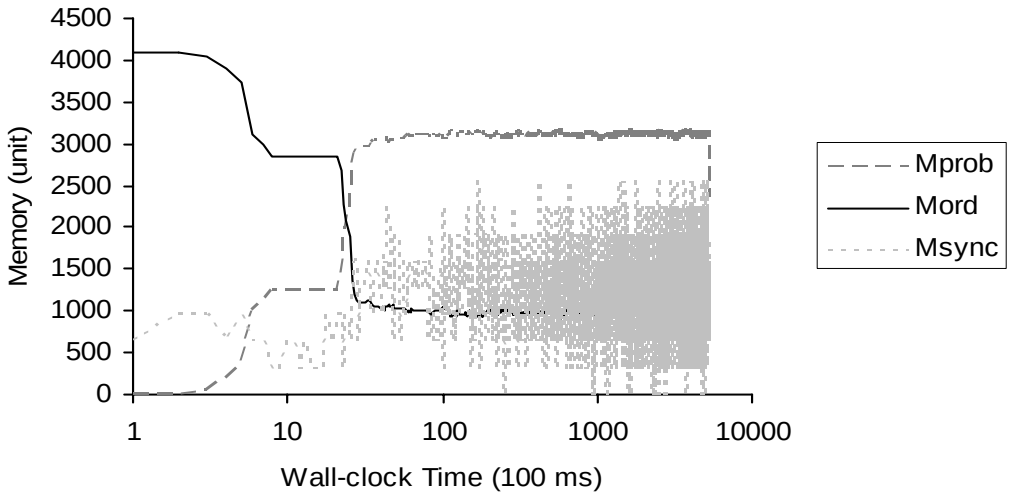


Figure 4.18: Memory Profile – PHOLD (32×32, 4)

Figure 4.19 and Figure 4.20 shows the total memory requirement of MIN and PHOLD, respectively. They show that each memory component (and hence the total memory requirement) increases as the problem size increases. For the same problem size, PHOLD requires more memory than MIN.

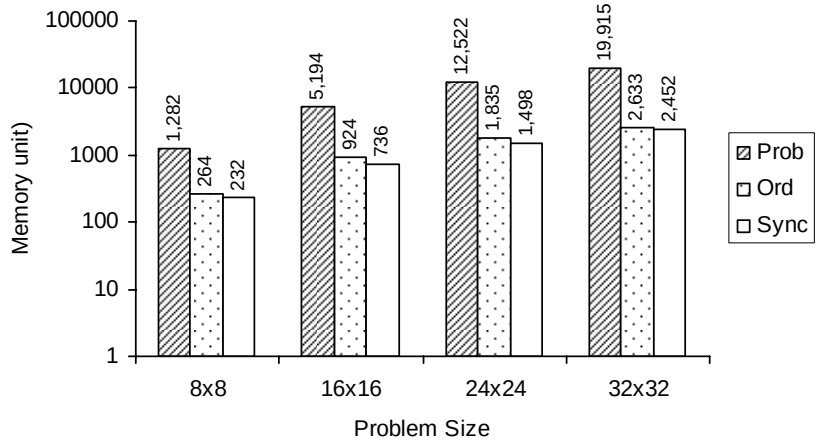


Figure 4.19: M^{tot} – MIN ($n \times n$, 0.8)

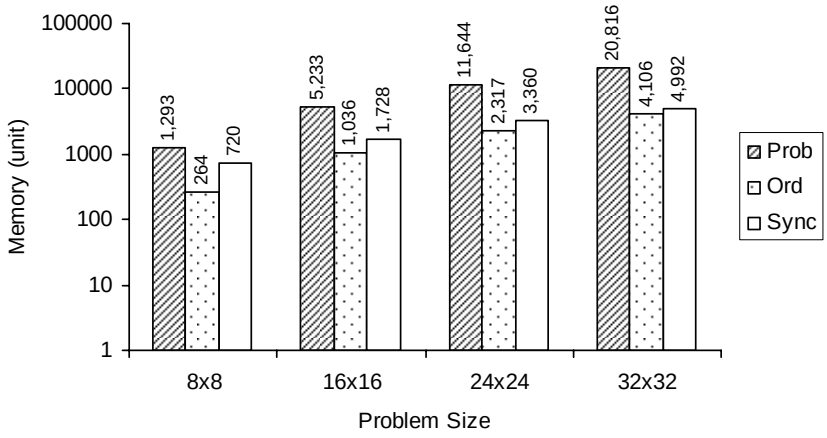


Figure 4.20: M^{tot} – PHOLD ($n \times n$, 4)

4.2.7 Strictness Analysis

Unlike event parallelism, strictness is time independent. It can be used to compare the performance of event orderings at different layers directly. In this section, we show the

strictness of a number of event orderings at the three layers. The results for MIN and PHOLD are shown in Figure 4.21 and Figure 4.22, respectively. The leftmost bar shows the strictness of event order at the physical system (denoted by PS). The subsequent four bars show the strictness of four event orders at the simulation model layer. The rightmost bar shows the strictness of event order maintained by the CMB protocol measured at the simulator layer (using four PPs).

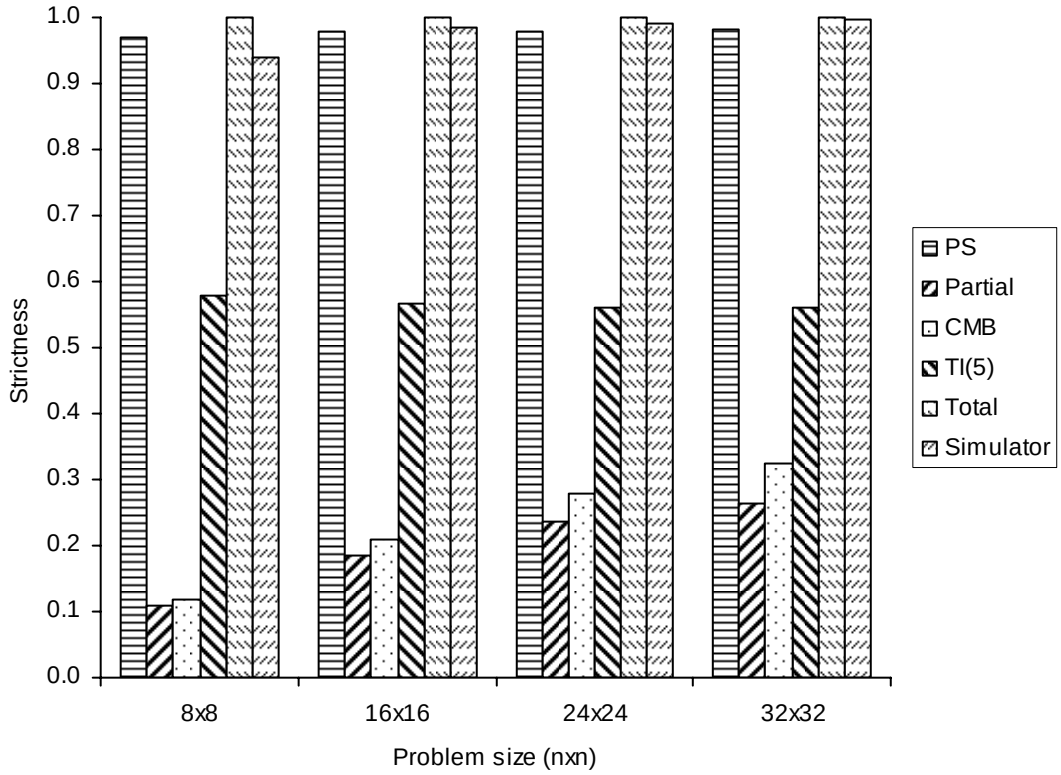


Figure 4.21: Strictness (ς) – MIN ($n \times n$, 0.8)

Both figures consistently show that partial event order is the least strict event order and total event order is the strictest. The difference is that in MIN, the CMB event order is less strict than the time-interval event order with window size of five. In PHOLD, however, time-interval event order with window size of five is less strict than the CMB event order. In Chapter 3, we have shown that the time-interval order is neither stricter

nor less strict than the CMB event order (see Figure 3.8). Therefore, it is possible that time-interval event order is stricter than CMB event order for one physical system, but less strict for other physical system. The strictness of partial, CMB, and time interval event orderings in PHOLD shows a significantly higher degree of event dependency than the strictness value of the same event orders in MIN. This is due to the higher degree of event dependency in the PHOLD physical system.

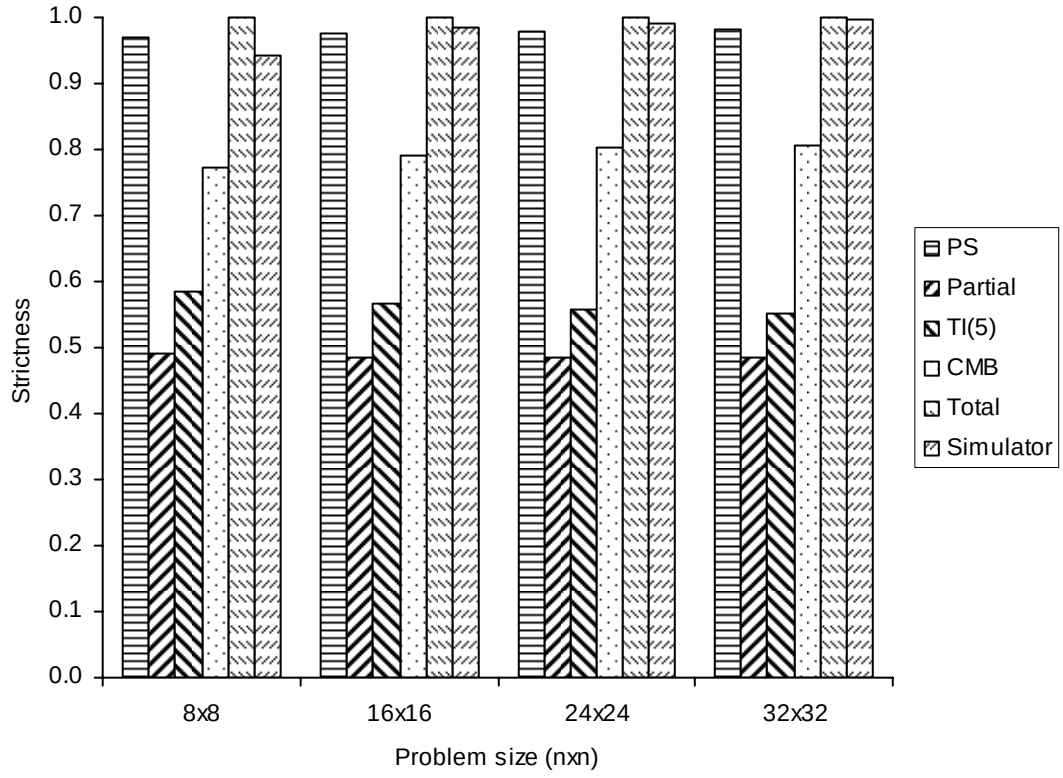


Figure 4.22: Strictness (ς) – PHOLD ($n \times n$, 4)

From the result, we can compare the strictness of CMB event order at the simulation model layer and its implementation at the simulator layer. At the simulator layer, the CMB protocol maintains its event order by sending null-messages so that an LP may know whether its event is safe to execute. We have shown in Chapter 3 that null messages increase the strictness of event order. Furthermore, at the simulator layer, the

number of processors is limited so that a number of LPs are mapped onto the same processor. Therefore, two independent events at two LPs will have to be processed sequentially at the simulator layer if both LPs are mapped onto the same processor. This also increases the strictness of the event order.

4.3 Performance Analysis of Ethernet Simulation

The objective of this experiment is to apply the framework to analyze the suitability of implementing an Ethernet simulation using a CMB protocol with demand driven optimization implemented in the SPaDES/Java. First, we analyze the time and space performance from the physical system layer to the simulator layer. After that, we analyze the scalability of the simulation.

The most commonly used medium access control (MAC) protocol for Ethernet is Carrier Sense Multiple Access with Collision Detection (CSMA/CD) [STAL04]. Under this protocol, a station that attempts to transmit must listen to the medium first to determine whether the medium is in use or not. If the medium is in use, then the station must wait; otherwise, it may transmit.

It is possible that two or more stations transmit at almost the same time so that all of them sense that the medium is idle. If this happens, there will be a collision, and the frame being sent will be garbled. Therefore, it is important for a station to be able to detect a collision. To provide for this, during transmission, a station has to listen to the medium to ascertain whether one or more stations are transmitting their frames for up to two propagation delay time. If collision is detected during the transmission, the station

will transmit a brief jamming signal to ensure that all stations know that there has been a collision and to cease the transmission. After transmitting the jamming signal, the stations which are involved in the collision must wait for a random amount of time before attempting to retransmit their frames (back off).

4.3.1 Model and Assumptions

Two types of simulation model have been used in Ethernet simulation [WANG99]. In the first type, the communication channel is modeled as a service center that serves a number of stations. This model is inherently sequential since there is only one service center in the model. We choose the second type where each station is modeled as a service center and frames are sent from one station to other stations. Therefore, this model can produce more parallelism by mapping each station onto one LP. Events are exchanged among LPs to model the frames moving from one station to other stations. There are six events used in the model:

a. *Frame arrival* occurs when a frame arrives at the MAC layer of a station. If the channel is idle, the station will transmit the frame. Otherwise, the frame will be put in the buffer.
b. *Begin transmit data* occurs when a station transmits the first bit of its frame to its neighbors.
c. *End transmit data* occurs when a station transmits the last bit of its frame to its neighbors.
d. *Begin receive data* occurs when a station receives the first bit of a frame sent by its neighbor. If the station is receiving or transmitting a frame, then collision occurs.

e. *End receive data* occurs when a station receives the last bit of a frame sent by its neighbor.

f. *End back off* occurs when a back off period has lapsed.

Figure 4.23 shows the state diagram of our model. A station can be in one of the four states represented by rectangles in Figure 4.23. Initially a station is idle. Upon receiving event *begin transmit data*, the station moves to a sending state. When event *end transmit data* is received, the station moves back to an idle state. When an idle station receives event *begin receive data*, it will change its state to receiving. Later, when the station receives event *end receive data*, the station moves to an idle state again.

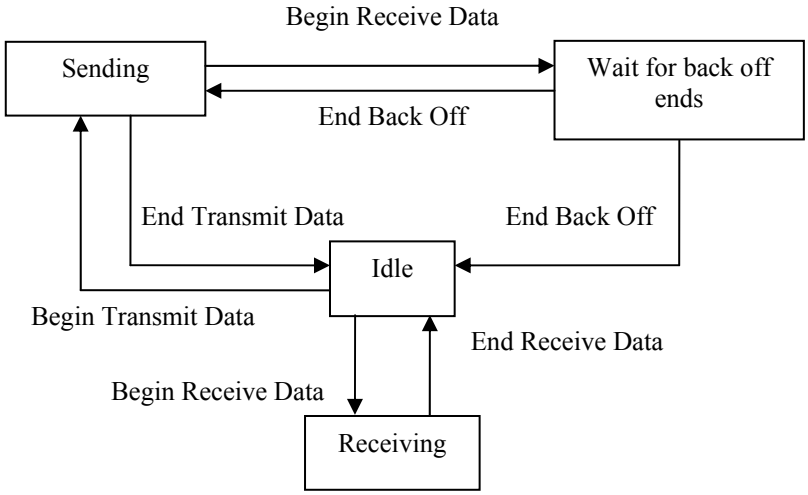


Figure 4.23: State Diagram of Ethernet Simulation

A collision occurs when a station is in a sending state and receives event *begin receive data*. The station will change its state to wait for back off ends. When the back off period lapses, the station moves to an idle state and tries to retransmit its corrupted frame (until the maximum retransmission is reached and the frame is dropped).

We choose the 100BASE-TX Ethernet specification and its parameters are shown in Table 4.6. We validate our simulation model with the analytical model developed by Chuan and Zukerman [CHUA01]. The validation is detailed in Appendix B. The result shows that there is no significant difference between our simulation result and their analytical result. We used the validated model in our experiments.

Parameters / Characteristics	**Values**
Simulation duration	1 second
Number of segments	1
Propagation delay	5.12 μs
LAN Speed	100 Mbps
Maximum frame size	1518 bytes
Minimum frame size	64 bytes
Jamming signal size	32 bytes
Inter-frame gap	12 bytes
Offered load	100%
Buffer size at each station	8 frames

Table 4.6: 100BASE-TX Physical Layer Medium Parameters

4.3.2 Performance Analysis

In this section, we study the performance of Ethernet simulation from the simulation problem to its simulation implementation using the SPaDES/Java parallel simulator. Figure 4.24 shows the number of events executed per second at the physical system layer (Π^{prob}). Π^{ord} depends on the number of stations (n) and frame size (F). In Ethernet, a sending station will send a frame to all stations within its collision domain, although only the intended recipient will pass the frame to the higher network layer. Hence, an increase in the number of stations implies more events are generated since each frame will be sent

to all stations. A larger frame size implies that within the same duration, the number of frame sent is less. Hence, the number of events is less for larger frame size.

The buffer at each station can hold up to eight frames, hence the $M^{prob} \leq 8n$. Since, the Ethernet operates at 100% workload (i.e., each station uses $1/n$ of the LAN bandwidth), M^{prob} reaches the maximum value of $8n$.

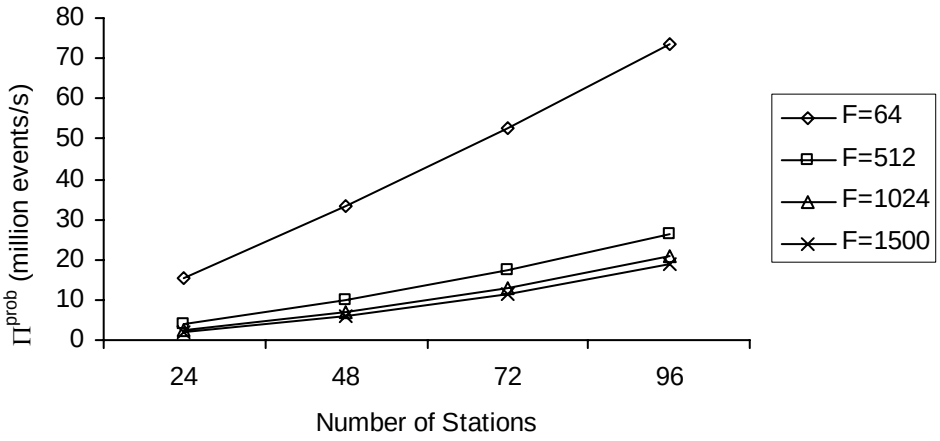


Figure 4.24: Π^{prob} – Ethernet (n, F)

The time (Π^{ord}) and space (M^{ord}) performance of the Ethernet simulation using CMB event order at the simulation model layer are shown in Figure 4.25 and Figure 4.26, respectively. The parallelism exploited by CMB event order depends on the number of stations and frame size for the same reason as in the physical system layer. Further, the performance of CMB event order depends on the lookahead.

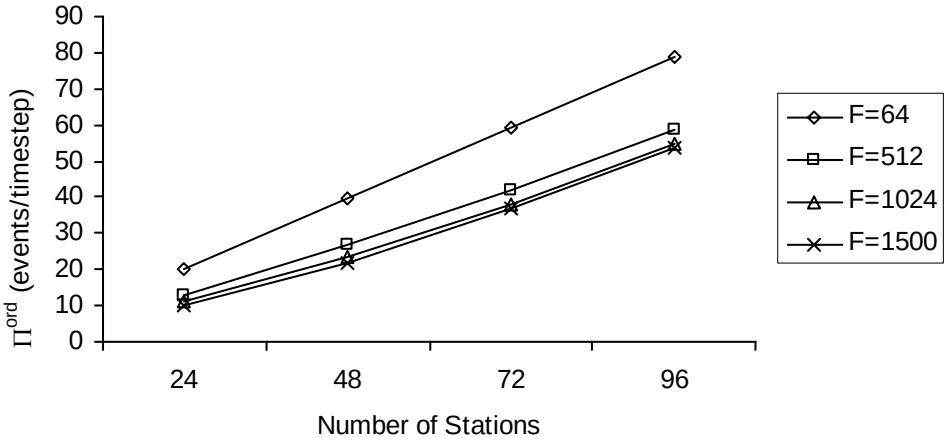


Figure 4.25: Π^{ord} – Ethernet (n, F) using CMB Event Order

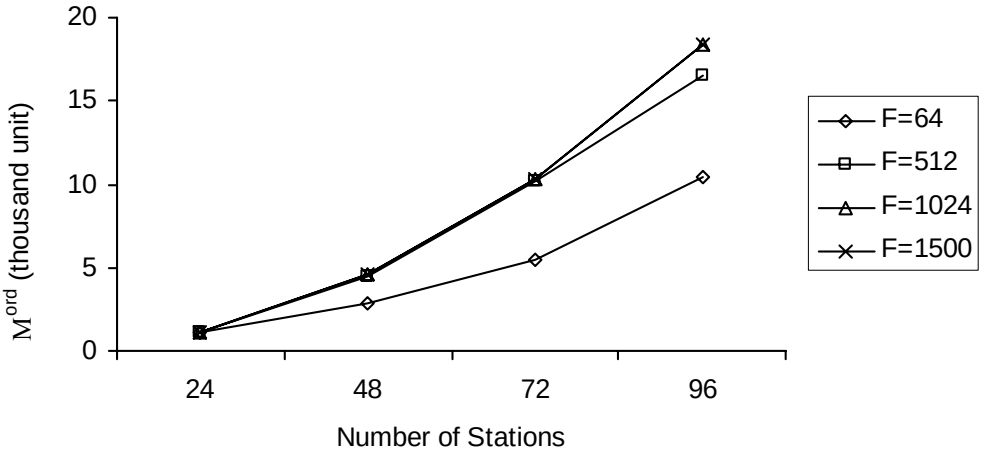


Figure 4.26: M^{ord} – Ethernet (n, F) using CMB Event Order

The lookahead in Ethernet is the propagation delay, i.e., 5.12μs. A larger frame size implies that the transmission time (the time to complete frame transmission) is longer, e.g., 5.12μs for 64 bytes and 120μs for 1,500 bytes. Therefore, when a station has executed event *begin transmit data*, the station can immediately execute event *end transmit data* for the frame size of 64 bytes. However, if the frame size is 1,500 bytes,

the station cannot only execute event *end transmit data*. It is because the event is scheduled at 120μs later while the lookahead is only 5.12μs (see Figure 4.27).

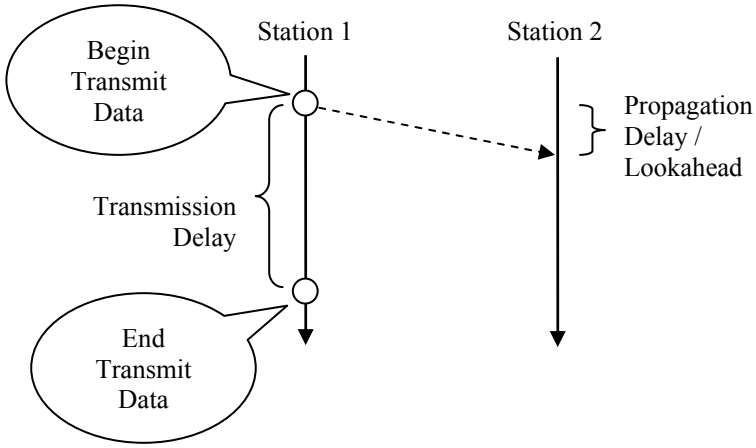


Figure 4.27: Frame Size and Lookahead

In our model, when a station transmits a frame, it generates *n* events (i.e., the arrival at each station excluding itself and an event *end transmit data*). Lin analytically proves that for large *n*, this type of event scheduling may produce a phenomenon whereby a simulation that exploits less parallelism requires more memory than a simulation that exploits more parallelism [LIN91]. Our experiment shows the same result for large number of stations (Figure 4.26)

We run our simulation on 2, 4, 6, and 8 PPs, and the time and space performance of the simulator is shown in Figure 4.28 and Figure 4.29, respectively. For the same number of PPs, effective event parallelism increases as we increase the number of stations. This is because more stations are mapped onto each PP which improves the utilization of each PP.

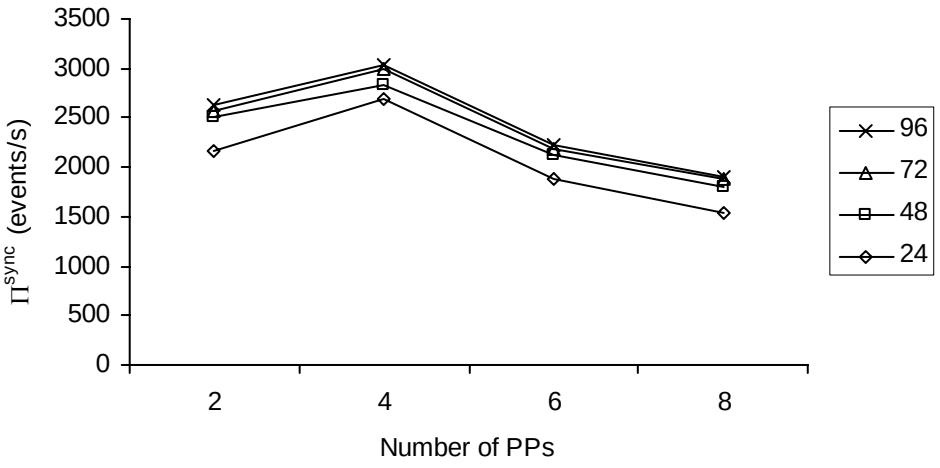


Figure 4.28: Π^{sync} – Ethernet(n, 64 bytes) on 2, 4, 6, and 8 PPs

For the same number of stations, an increase in the number of PPs increases computing power, but at the same time more synchronization overheads are required. The results show that an increase from two PPs to four PPs improves the parallelism. However, further increase in the number of PPs decreases the exploited parallelism.

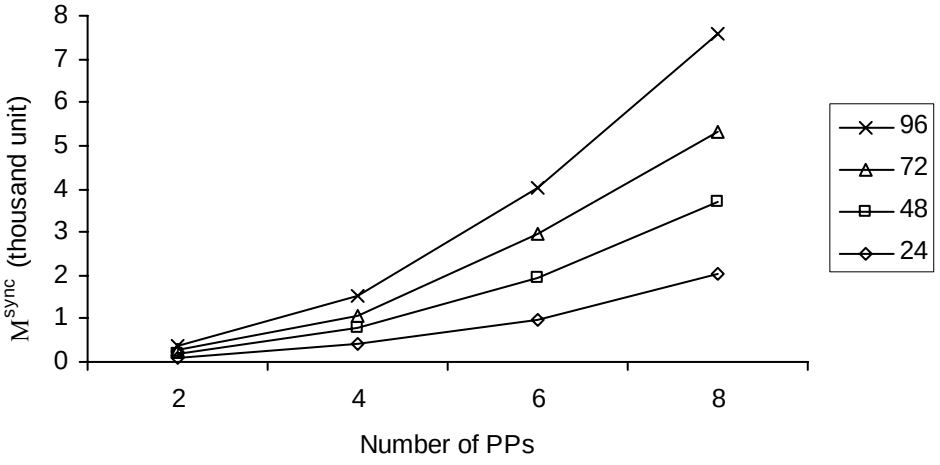


Figure 4.29: M^{sync} – Ethernet(n, 64 bytes) on 2, 4, 6, and 8 PPs

Figure 4.30 shows the effective parallelism as we increase both the number of stations and the number of PPs by the same ratio. The result shows that our Ethernet simulation is not scalable because further increase in both the number of stations and PPs does not increase the effective parallelism.

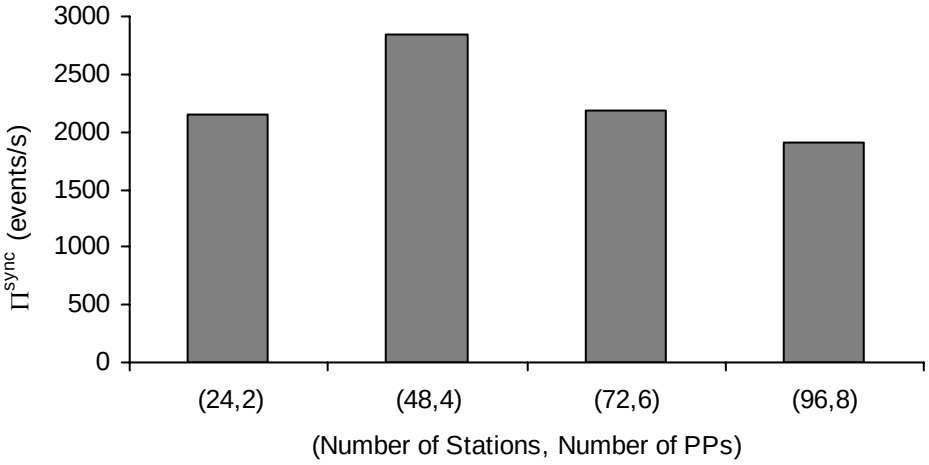


Figure 4.30: Π^{sync} – Ethernet(n, 64 bytes) on 2, 4, 6, and 8 PPs

4.4 Summary

We have conducted two sets of experiments. In the first set, we tested the proposed framework using an *open system* called Multistage Interconnection Network (MIN) and a *closed system* called PHOLD. In the second set, we discussed the application of our framework in studying the performance of Ethernet simulation. In the first set, we measured event parallelism and memory requirement at each layer. The results are summarized in Table 4.7.

At the physical system layer, Π^{prob} and M^{prob} vary with problem size ($n \times n$) and workload of the system (ρ in MIN or m in PHOLD). Based on Equations 4.1 and 4.2, Π^{prob} of MIN and PHOLD can be modeled as Equations 4.4 and 4.5, respectively (c_i is a constant):

$$\Pi^{prob} = \frac{\| E \|}{D} = \frac{2 \times \rho \times \mu \times D \times n \times n}{D} = 2 \times \rho \times \mu \times n \times n \quad (4.4)$$

$$\Pi^{prob} = \frac{\| E \|}{D} = \frac{2 \times c_1 \times \log(c_2 + m) \times \mu \times D \times n \times n}{D} = 2 \times c_1 \times \log(c_2 + m) \times \mu \times n \times n \quad (4.5)$$

Layers	MIN		PHOLD	
	Event Parallelism	**Memory**	**Event Parallelism**	**Memory**
Problem	f(n×n, ρ)	f(n×n, ρ)	f(n×n, m)	f(n×n, m)
Simulation Model	f(n×n, ρ, R)	f(n×n, ρ, R)	f(n×n, m, R)	f(n×n, m, R)
Simulator	f(n×n, ρ, R, …)	f(n×n, ρ, R, …)	f(n×n, m, R, …)	f(n×n, m, R, …)

Table 4.7: Time and Space Performance Summary

The experimental result shows that traffic intensity (ρ) has an exponential effect on M^{prob}, and problem size (n×n) has a linear effect on M^{prob}. Similarly, in PHOLD, problem size has a linear effect on M^{prob}. However, message density has a logarithmic effect on M^{prob}. Based on this observation, we develop a first order model with interaction [MEND95] for MIN and PHOLD as shown in Equations 4.6 and 4.7, respectively:

$$M^{prob} = c_1 \times n \times n + c_2 \times e^{\rho} + c_3 \times n \times n \times e^{\rho} + \varepsilon \quad (4.6)$$

$$M^{prob} = c_1 \times n \times n + c_2 \times \ln m + c_3 \times n \times n \times \ln m + \varepsilon \quad (4.7)$$

At the simulation model layer, Π^{ord} and M^{ord} vary with problem size, workload of the system, and the event order used (R). Based on the experimental results, a first order model with interaction is used to model Π^{ord} and M^{ord} as shown in Equations 4.8 to 4.11, where different event orders use different constants c_1, c_2, and c_3:

$$\Pi^{ord} = c_1 \times n \times n + c_2 \times \ln \rho + c_3 \times n \times n \times \ln \rho + \varepsilon \text{ (MIN)} \quad (4.8)$$

$$\Pi^{ord} = c_1 \times n \times n + c_2 \times \ln m + c_3 \times n \times n \times \ln m + \varepsilon \text{ (PHOLD)} \quad (4.9)$$

$$M^{ord} = c_1 \times n \times n + c_2 \times \rho + c_3 \times n \times n \times \rho + \varepsilon \text{ (MIN)} \quad (4.10)$$

$$M^{ord} = c_1 \times n \times n + c_2 \times e^m + c_3 \times n \times n \times e^m + \varepsilon \text{ (PHOLD)} \quad (4.11)$$

At the simulator layer, Π^{sync} and M^{sync} depend on problem size, system workload, event order maintained at runtime, protocol specific factors, and execution platform factors.

Next, we normalized event parallelism to allow performance comparison across layers. The comparison in event parallelism between the physical system layer and the simulation model layer reveals that we can get more parallelism at the simulation model layer when we use a less strict event order (than the one used at the physical system layer). The comparison between the simulation model layer and the simulator layer reveals that parallelism at the simulator layer cannot be more than the parallelism at the simulation model layer. This is due to the implementation overhead. Subsequently, we measured total memory requirement. The result shows that to simulate PHOLD requires more memory than MIN because PHOLD generates more null-messages due to its topology.

The last experiment in the first set measures the strictness of different event orders. The results show that for the same event order, the strictness value in PHOLD is higher than in MIN. This suggests that PHOLD generates more dependent events than MIN. The results support our analytical results on the relationship among different event orders based on the stricter relation given in Chapter 3. Next, we see that factors at the

simulator layer such as null messages and the number of processors increase the strictness of an event order.

In the second set of experiments, we applied our framework to study the performance of Ethernet simulation from the simulation problem to its simulation implementation. Further, we have shown scalability analysis of the Ethernet simulation. First, we fixed the problem size (i.e., the number of stations) and increase the number of PPs. The result shows that initially, the effective event parallelism increases. However, further increase in the number of PPs decreases the exploited parallelism. Next, we increase the problem size and the number of PPs by the same ratio (fixed-time analysis). The result shows a similar result with the fixed-size analysis. Therefore, Ethernet simulation using CMB parallel simulation protocol is not scalable.

Chapter 5

Conclusions

5.1 Summary

This thesis has proposed a framework for the formalization and characterization of simulation performance. The proposed framework provides a basis for understanding and analyzing simulation performance from the simulation problem to the simulation implementation.

Figure 5.1 summarizes the proposed framework. We formalize simulation event orderings that provide the theoretical foundation for simulation performance analysis based on the partially ordered set (poset). We characterize simulation performance in three layers: physical system, simulation model and simulator. Our analysis focuses on *time* (event parallelism) and *space* (memory requirement) performance. Because of the different definitions of time used at different layers, a normalization process is required to allow time performance comparison across layers. Event parallelism depends on event dependency and time. We have proposed a time independent measure called *strictness* for quantifying the degree of event dependency of a simulation event ordering,

and a relation called *stricter* to compare the degree of event dependency of simulation event orderings.

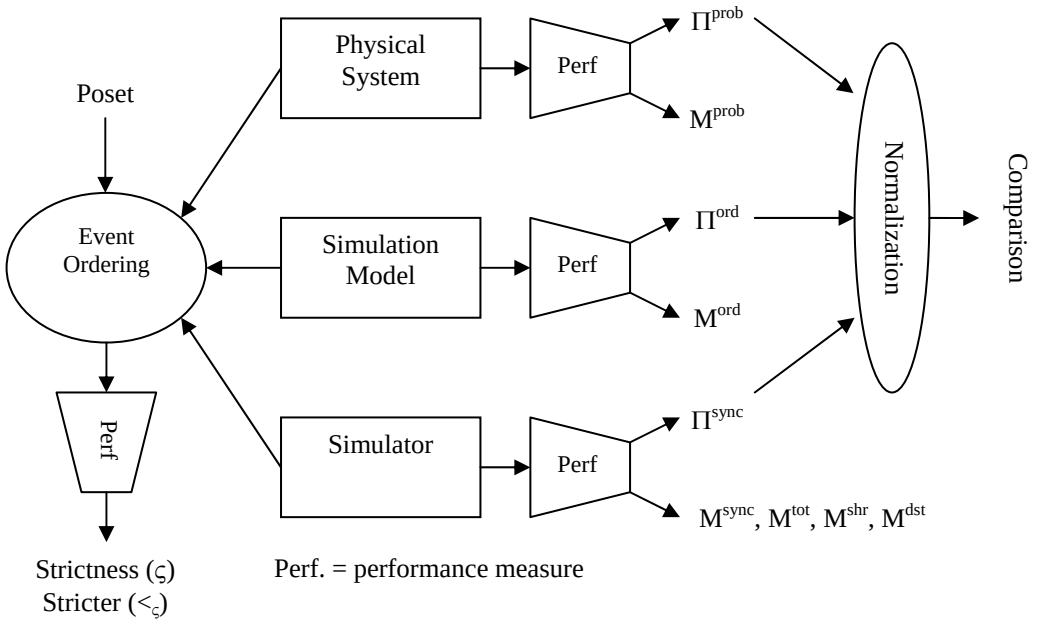


Figure 5.1: Framework for Formalization and Characterization of Simulation Performance

In summary, this thesis offers *three* main contributions: formalization of event orderings, time and space performance characterization, and strictness measurement.

5.1.1 Formalization of Simulation Event Orderings

We have proposed the formalization of simulation event orderings based on poset. Poset is a branch of discrete mathematics that studies the ordering of elements in a set. In event ordering, we study the ordering of events in a given set of events. At the physical system, events are ordered based on their physical time of occurrence. A number of event orderings can be used at the simulation model layer. A simulator maintains a specific event ordering at runtime. We have extracted and formalized the event orderings

maintained by a number of simulators (sequential and parallel). This formalization provides a theoretical foundation for carrying out performance analysis of simulation. The existing theories in poset such as Dilworth's chain covering theorem can be applied to explain the performance of different simulation event orderings. The proposed formalization can also be extended to other computer systems that involve ordering in their operations such as memory operation orderings in the memory consistency model [CULL99], instruction orderings in processor design [HENN03], and job orderings in grid computing [THAI03].

5.1.2 Time and Space Performance Characterization

We have proposed characterizing simulation performance along the three natural boundaries in simulation, i.e., *physical system*, *simulation model* and *simulator* (implementation). This thesis focuses on the simulation of physical systems that can be modeled as queuing networks. Therefore, at the physical system layer, we view a physical system as a queuing network. At the simulation model layer, based on the virtual time paradigm [JEFF85], we view a simulation model as a set of logical processes (LPs). Every LP models a physical process (service center) in the physical system. The interaction among physical processes in the physical system is modeled by exchanging events among LPs in the simulation model. At the simulator layer, a simulator that implements a simulation model is executed on a platform consisting of one or more physical processors (PPs).

We have formalized event parallelism and memory requirement at each of the layers. Event parallelism is defined as the number of events executed per unit of time. At the

physical system layer, the physical time unit is used. At the simulation model layer, the time unit is timestep. At the simulator layer, the wall-clock time unit is used. Analysis at each layer can be done independently. To compare event parallelism at different layers, normalization is necessary because the time units used at the three layers differ.

The memory requirement at the physical system layer is derived from the maximum queue size of each service center in the system. At the simulation model layer, the memory requirement is derived from the maximum event list size of each LP. At the simulator layer, the memory requirement is derived from the maximum size of the data structure that is used for synchronization purposes. The memory requirement at each layer can be analyzed independently.

The total memory requirement is measured at the simulator layer. At runtime, in addition to the data structure that is used for synchronization purposes, other data structures (such as a linked list) are also used to implement queues and event lists. These data structures constitute the total memory requirement of simulation.

The proposed three-layer performance characterization can be extended to other areas. In shared-memory computer architecture, for example, a program that is going to be executed constitutes the *physical system layer*, a memory consistency model that specifies the ordering of memory operations independent of implementation constitutes the *model layer*, and an implementation of the memory consistency model constitutes the *implementation layer*. With this characterization, we can analyze the performance from the physical system layer to the implementation layer. Similarly, in processor design (to exploit instruction level parallelism), a program that is going to be executed constitutes

the *physical system layer*, the different instruction orderings analyzed using an idealized processor model constitutes the *model layer*, and an implementation of the instruction orderings on a real processor constitutes the *implementation layer*.

5.1.3 Strictness of Event Orderings

We have proposed a relation called *stricter* and a measure called *strictness* to compare and quantify the degree of event dependency of event orderings respectively. An event order R_1 is stricter than R_2 if event x is ordered before event y in R_2 then x is also ordered before y in R_1 but not vice versa. For any two events, they are either comparable or non-comparable (concurrent). We have shown that if event order R_1 is stricter than R_2, then R_1 will never execute more events than R_2 for the same measurement duration.

The strictness of an event ordering is measured from the number of comparable event pairs which is divided by the number of comparable event pairs in the total event ordering. A strictness value of x means that the probability of two events being comparable is x. If all events are comparable (non comparable) in an event ordering, then the strictness of the event ordering is one (zero).

The relation *stricter* has been used in the memory consistency model [GHAR95]. It can be extended to compare the degree of dependency of different orderings in areas such as instruction orderings in the processor design [HENN03]. The measure *strictness* can also be extended to measure the degree of dependency in an ordering such as memory operation orderings in the memory consistency model [CULL99] and instruction orderings in processor design [HENN03].

5.2 Future Works

We highlight three possible research areas that can be further explored:

a. Hardware Implementation.

The formalization of event orderings is motivated by research in the memory consistency model. A number of hardware implementations have been built based on the existing memory consistency models [CULL99, GHAR95]. Learning from research in the memory consistency model, it may be possible to explore the hardware implementation of a simulation event ordering. This may improve simulation performance.

b. Empirical Study on Other Protocols.

This thesis has used only a CMB protocol with demand driven optimization in the experiment. We have formalized a number of event orders and each event order can be implemented in many ways, resulting in different simulation protocols. Therefore, empirical study of the various protocols will be useful in supporting the proposed framework. Further, the proposed framework also provides the foundation for studying future event orderings and new protocol implementations.

c. Extreme Value Analysis.

We have formalized the memory requirement based on the total of a maximum value, such as maximum queue size at the physical system layer, maximum event list size at the simulation model layer, and maximum buffer size at the simulator layer. The stochastic behavior of a maximum value is the focus of research in the extreme value theory [COLE01]. Therefore, the extreme value theory may provide some analytic model that can explain the stochastic behavior of the memory requirement.

References

AFEK89 Afek, Y., Brown, J. and Merritt, M. A Lazy Cache Algorithm. *Symposium on Parallel Algorithms and Architectures*, pp. 209-222, 1989.

AKYI93 Akyildiz, I.F., Chen L., Das, R., Fujimoto, R.M. and Serfozo, R.F. The Effect of Memory Capacity on Time Warp Performance. *Journal of Parallel and Distributed Computing*, 18 (4), pp. 411-422, 1993.

ATTI98 Attiya, H. and Welch, J. Distributed Computing: Fundamentals, Simulations and Advanced Topics. McGraw-Hill, 1998.

AYAN87 Ayani, R. Performance of Priority-queue Implementations on Shared Memory Multi Processor Computer Systems. *TR TRITA-CS-8705, Dept. of Telecommunications and Computer Systems, Royal Institute of Technology,* Sweden, 1987.

AYAN92 Ayani, R. and Rajaei, H. Parallel Simulation using Conservative Time Windows. *Proceedings of Winter Simulation Conference*, pp. 709-717, December, 1992.

BAIL94 Bailey, M.L. and Pagels, M.A. Empirical Measurements of Overheads in Conservative Asynchronous Simulations. *ACM Transaction on Modeling and Computer Simulation*, 4 (4), pp. 350-367, 1994.

BAIN88 Bain, W.L. and Scott, D.S. An Algorithm for Time Synchronization in Distributed Discrete Event Simulation. *Proceedings of the SCS Multi-conference on Distributed Simulation*, 19 (3), pp. 30-33, 1988.

BAJA99 Bajaj, L., Bagrodia, R. and Meyer, R. Case Study: Parallelizing a Sequential Simulation Model. *Proceedings of the 13th Workshop on Parallel and Distributed Simulation*, pp. 29-36, 1999.

BAGR91 Bagrodia, R., Chandy, K.M. and Liao, W.T. A Unifying Framework for Distributed Simulation. *ACM Transactions on Modeling and Computer Simulation*, 1 (4), pp. 348-385, 1991.

BAGR96 Bagrodia, R. Perils and Pitfalls of Parallel Discrete-event Simulation. *Proceedings of the 1996 Winter Simulation Conference*, pp. 136-143, 1996.

BAGR99 Bagrodia, R., Deelman, E., Docy, S. and Phan T. Performance Prediction of Large Parallel Applications Using Parallel Simulation. *Symposium on Principles & Practice of Parallel Programming*, pp. 151-162, 1999.

BALA97 Balakrishnan, V., Frey, P., Abu-Ghazaleh, N.B and Wilsey, P.A. A Framework for Performance Analysis of Parallel Discrete Event Simulators. *Proceedings of the Winter Simulation Conference*, pp. 429-436, 1997.

BANK00 Banks J., Carson, J.S., Nelson, B.L. and Nicol, D.M. Discrete-Event System Simulation, 3rd edition. Prentice Hall, 2000.

BANK03 Banks, J., Hugan, J.C., Lendermann, P., McClean, C., Page, E.H., Pegden, C.D., Ulgen, O. and Wilson, J.R. The Future of the Simulation Industry. *Proceedings of the 2003 Winter Simulation Conference*, pp. 2033-2043, 2003.

BARR95 Barriga, L., Ronngren, R. and Ayani, R. Benchmarking Parallel Simulation Algorihtms. *Proceedings of the IEEE First International Conference on Algorithms and Architectures for Parallel Processing*, pp. 611-620, 1995.

BERR85 Berry O. and Jefferson D. Critical Path Analysis of Distributed Simulation. *Proceedings of SCS Multiconference on Distributed Simulation*, pp. 57-60, 1985.

BIRM87 Birman, K.P. and Joseph, T.A. Reliable Communication in the Presence of Failure. *ACM Transaction on Computer Systems*, 5 (1), pp. 47-76, 1987.

BODO03 Bodoh, D.J. and Wieland, F. Performance Experiments with the High Level Architecture and the Total Airport and Airspace Model (TAAM). *Proceedings of the 17th Workshop on Parallel and Distributed Simulation*, pp. 31-39, 2003.

BOGA95 Bogart, K.P., Fishburn, P., Isaac G. and Langley, L. Proper and Unit Tolerance Graphs. *Discrete Application Mathematics*, 60, pp. 37-51, 1995.

BOLC98 Bolch, G., Greiner, S., de Meer, H. and Trivedi, K.S. Queuing Networks and Markov Chains: Modeling and Performance Evaluation with Computer Science Applications. John Wiley & Sons, Inc., 1998.

BRYA84 Bryant, R.E. A Switch-Level Model and Simulator for MOS Digital Systems. *IEEE Transaction on Computers*, 33 (2), pp. 160-177, 1984.

BUXT62 Buxton, J.N. and Laski, J.G. Control and Simulation Language. *Journal of Computer*, 5, 1962.

CAI90 Cai, W. and Turner, S.J., An Algorithm for Distributed Discrete Event Simulation – The Carrier Null Message Approach. *Proceedings of the SCS Multiconference on Distributed Simulation*, pp. 3-8, 1990.

CARO95 Carothers, C.D., Fujimoto, R.M. and Lin, Y.-B. A Case Study in Simulating PCS Networks Using Time Warp. *Proceedings of the 9th Workshop on Parallel and Distributed Simulation*, pp. 87-94, 1995

CARO00 Carothers, C., Perumalla, K. and Fujimoto, R.M. Efficient Optimistic Parallel Simulations using Reverse Computation. *ACM Transaction on Modelling and Computer Simulation*, 9 (3), pp. 224-253, 1999.

CHAN79 Chandy, K.M. and Misra J. Distributed Simulation: a Case Study in Design and Verification of Distributed Programs. *IEEE Transaction on Software Engineering*, 5, (5), pp. 440-452, 1979.

CHEN03 Chen, G. and Zymanski, B.K. Four Types of Lookback. *Proceedings of the 17th Workshop on Parallel and Distributed Simulation*, pp. 3-10, 2003.

CHIO95 Chiola G. and Ferscha A. Performance Comparable Design of Efficient Synchronization Protocols for Distributed Simulation. *Proceedings of the 3rd International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, pp. 59-65, 1995.

CHUA01 Chuan, H.F and Zukerman, M. Performance Comparison of CSMA/RI and CSMA/CD with BEB. *Proceedings of the IEEE International Conference on Communications*, pp. 2670-2675, 2001.

CLEA94 Cleary, J., Gomes, F., Unger, B., Zhonge, X. and Thudt R. Cost of State Saving and Rollback. *Proceedings of the 8th Workshop on Parallel and Distributed Simulation*, pp. 94-101, 1994.

COLE01 Coles, S. An introduction to statistical modeling of extreme values. Springer, 2001.

CULL99 Culler, D.E., Singh, J.P. and Gupta, A. Parallel Computer Architecture: A Hardware/Software Approach. Morgan Kaufmann, 1999.

DICK96 Dickens, P.M., Nicol, D.M., Reynolds, P.M. and Duva, J.M. Analysis of Bounded Time Warp and Comparison with YAWNS. *ACM Transactions on Modeling and Computer Simulation*, 6 (4), pp. 297-320, 1996.

DAS94 Das, S.R. and Fujimoto, R.M. An Adaptive Memory Management Protocol for Time Warp Parallel Simulation. *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pp. 201-210, 1994.

DAS97 Das, S.R. and Fujimoto, R.M. Adaptive Memory Management and Optimism Control in Time Warp. *ACM Transactions on Modeling and Computer Simulation*, 7 (2), pp. 239-271, 1997.

DUBO90 Dubois, M., Scheurich, S. and Briggs F. Memory Access Buffering in Multiprocessors. *Proceedings of the 13th Annual International Symposium on Computer Architecture*, pp. 434-442, June, 1986.

DUSH41 Dushnik B. and Miller, E.W. Partially Ordered Sets. *American Journal of Mathematics*, 63, pp. 600-610, 1941.

FELD91 Felderman, R.E. and Kleinrock, L. Two Processor Time Warp Analysis: Some Results on Unifying Approach. *Proceedings of the 5th Workshop on Parallel and Distributed Simulation*, pp. 3-10, 1991.

FERS96 Ferscha A. Parallel and Distributed Simulation of Discrete-event System. *Hawaii International Conference on System Sciences*, Tutorial, 1996.

FERS97 Ferscha, A., Johnson J. and Turner, S.J. Early Prediction of Parallel Simulation Protocols. Research Report R367, University of Exeter, 1997.

FISH88 Fishburn, P.C. Interval Orders and Circle Orders. *Order* 5, pp. 225-234, 1988.

FISH89 Fishburn, P.C. Circle Orders and Angle Orders. *Order* 6, pp. 39-47, 1989.

FISH99 Fishburn, P.C. and Trotter, W.T. Split Semiorders. *Discrete Mathematics*, 195, pp. 111-126, 1999.

FRAN97 Franks, S., Gomes, F., Unger, B. and Cleary, J. State Saving for Interactive Optimistic Simulation. *Proceedings of the 11th Workshop on Parallel and Distributed Simulation*, pp. 72-79, 1997.

FUJI89 Fujimoto, R. Time Warp on a Shared Memory Multiprocessor. *Transaction on Society Computer Simulation*, 6 (3), pp. 211-239, 1989.

FUJI90 Fujimoto, R.M. Performance of Time Warp under Synthetic Workloads. *Proceedings of SCS Multiconference on Distributed Simulation*, 22 (1), pp. 23-28, 1990.

FUJI96 Fujimoto, R.M. and Weatherly, R.M. Time Management in the DoD High Level Architecture. *Proceedings of the 10th Workshop on Parallel and Distributed Simulation*, pp. 60-67, 1996.

FUJI99 Fujimoto, R.M. Exploiting Temporal Uncertainty in Parallel and Distributed Simulations. *Proceedings of the 13th Workshop on Parallel and Distributed Simulation*, pp. 46-53, 1999.

FUJI00 Fujimoto, R.M. Parallel and Distributed Simulation Systems. John Wiley & Sons, Inc., 2000.

GAFN88 Gafni, A. Rollback Mechanisms for Optimistic Distributed Simulation Systems. *Proceedings of the SCS Multiconference on Distributed Simulation*, 19 (3), pp. 61-67, 1990.

GHAM00 Gambhire, P. and Kshemkalyani, A.D. Evaluation of the Optimal Causal Message Ordering Algorithm. *Proceedings of the High Performance Computing*, Lecture Notes on Computer Science no. 1970, Springler-Verlag, pp. 83-95, 2000.

GHAR90 Gharachorloo, K., Lenoski, D., Laudon, J., Gibbons, P., Gupta, A. and Henessy J. Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors. *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pp. 15-26, 1990.

GHAR91 Gharachorloo, K., Gupta, A. and Hennesy, J. Two Techniques to Enhance the Performance of Memory Consistency Model. *Proceedings of the International Conference on Parallel Processing*, pp. 355-364, 1991.

GHAR95 Gharachorloo, K. Memory Consistency Models for Shared-Memory Multiprocessors. Research Report 95/9, Western Research Laboratory, 1995.

GONS85 Gonsalves, T.A. Performance Characteristics of 2 Ethernets: an Experimental Study. *ACM SIGMETRICS Performance Evaluation Review*, 13 (2), pp. 78-86, 1985.

GUPT91 Gupta, A., Akyildiz, I.F. and Fujimoto, R.M. Performance Analysis of Time Warp with Multiple Homogenous Processors. *IEEE Transaction on Software Engineering*, 17 (10), pp. 1013-1027, 1991.

HADZ93 Hadzilacos, V. and Toueg S. Fault-Tolerant Broadcasts and Related Problems. A chapter in Distributed Systems, Mullender, S. (ed), Addison-Wesley, 2nd edition, 1993.

HENN03 Hennessy, J.L. and Patterson, D.A. Computer Architecture: A Quantitative Approach. Morgan Kaufmann, 3rd edition, 2003.

HU97 Hu, L. and Gorton, I. Performance Evaluation of Parallel Systems: A Survey. UNSW-CSE-TR-9707, Department of Computer Systems, University of New South Wales, 1997.

HUSS04 Hussain A., Kapoor, A. and Heidemann J. The Effect of Detail on Ethernet Simulation. *Proceedings of the 18th Workshop on Parallel and Distributed Simulation*, pp. 97-104, 2004.

JAIN91 Jain, R. The Art of Computer Systems Performance Analysis. John Wiley & Sons, Inc., 1991.

JEFF85 Jefferson, D.A. Virtual Time. *ACM Transaction on Programming Language System*, 7 (3), pp. 404-425, 1985.

JEFF87 Jefferson, D., Beckman B, Wieland, F., Blume, L. and Diloreto M. Distributed Simulation and the Time Warp Operating System. *ACM Operating Systems Review*, 21 (5), pp. 77-93, 1987.

JEFF90 Jefferson D. Virtual Time II: Storage Management in Distributed Simulation. *Proceedings of the 9th ACM symposium on Principles of Distributed Computing*, pp. 75-89, 1990.

JEFF91 Jefferson D. Supercritical Speedup. *Proceedings of the 24th Annual Simulation Symposium*, pp. 159-168, 1991.

JHA96 Jha, V. and Bagrodia, R. A Performance Evaluation Methodology for Parallel Simulation Protocols. *Proceedings of the 10th Workshop on Parallel and Distributed Simulation*, pp. 180-1985, 1996.

KIDD04 Kiddle, C., Simmonds, R. and Unger B. Performance of a Mixed Shared/Distributed Memory Parallel Network Simulator. *Proceedings of the 18th Workshop on Parallel and Distributed Simulation*, pp. 17-25, 2004.

KUEK92 Kuek, S.S. and Wong, W.C. Ordered Dynamic Channel Assignment Scheme with Reassignment in Highway Microcells. *IEEE Transaction on Vehicle Technology*, 41 (3), pp. 271-277, 1992.

LAMP78 Lamport, L. Time, Clocks, and the Ordering of Events in a Distributed System. *Communication ACM*, 21 (7), pp. 558-565, 1978.

LAMP79 Lamport, L. How to Make a Multiprocessor that Correctly Executes Multiprocess Program. *IEEE Transactions on Computers*, 28 (9), pp.690-691, 1979.

LAND91 Landin, A., Hagerstein, E. and Haridi, S. Race-free Interconnection Networks and Multiprocessor Consistency. *Proceedings of the 18th Annual International Symposium on Computer Architecture*, pp. 27-30, 1991.

LAVE83 Lavenberg, S., Muntz, R. and Samadi, B. Performance Analysis of a Rollback Method for Distributed Simulation. *Performance 83*, pp. 117-132, 1983.

LAW84 Law, A.M and Larmey, C.S. Introduction to Simulation using SIMSCRIPT II.5. CACI Products Company, La Jolla, California, 1984.

LI04 Li, L. and Tropper, C. Event Reconstruction in Time Warp. *Proceedings of the 18th Workshop on Parallel and Distributed Simulation*, pp. 37-44, 2004.

LILJ02 Liljenstam, M., Yuan, Y., Premore, B.J. and Nicol, D. A Mixed Abstraction Level Simulation Model of Large-Scale Internet Worm Infestations. *Proceedings of the 10th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems*, pp. 109-116, 2002.

LIN90 Lin, Y.B. and Lazowska, E.D. Optimality Considerations for Time Warp Parallel Simulation. *Proceedings of the SCS Multiconference on Distributed Simulation*, pp. 29-34, 1990.

LIN91 Lin, Y.B. and Preiss, B.R. Optimal Memory Management for Time Warp Parallel Simulation. *ACM Transactions on Modeling and Computer Simulation*, 1 (4), pp. 283-307, 1991.

LIN92 Lin, Y.B. Parallelism Analyzers for Parallel Discrete Event Simulation. *ACM Transaction on Modelling and Computer Simulation*, 2 (3), pp. 239-264, 1992.

LIN93 Lin, Y.B. Will Parallel Simulation Research Survive? *ORSA Journal of Computing*, 5 (3), pp. 236-238, 1993.

LIPT90 Lipton R. and Mizel D. Time Warp vs. Chandy-Misra: a Worst Case Comparison. *Proceedings of the SCS Multiconference on Distributed Simulation*, pp. 137-143, 1990.

LIU99 Liu, J., Nicol, D., Premore, B. and Poplawski, A. Performance Prediction of a Parallel Simulator. *Proceedings of the 13th Workshop on Parallel and Distributed Simulation*, pp. 156-164, 1999.

LIVN85 Livny, M. A Study of Parallelism in Distributed Simulation. *Proceedings of SCS Multiconference on Distributed Simulation*, pp. 94-98, 1985.

LOPE00 Loper, M.L. and Fujimoto, R.M. Pre-sampling as an Approach for Exploiting Temporal Uncertainty. *Proceedings of the 14th Workshop on Parallel and Distributed Simulation*, pp.157-164, 2000.

LUBA89 Lubachesky, B.D. Efficient Distributed Event-driven Simulations of Multiple-loop Networks. *Communications of the ACM*, 32 (1), pp. 111-123, 1989.

MART97 Martini, P., Rumekasten, M. and Tolle, J. Tolerant Synchronization for Distributed Simulations of Interconnected Computer Networks. *Proceedings of the 11th Workshop on Parallel and Distributed Simulation*, pp. 138-141, 1997.

MART03 Martin, D.E., Wilsey, P.A., Hoekstra, R.J, Keiter, E.R., Hutchinson, S.A., Russo, T.V. and Waters, L.J. Redesigning the WARPED Simulation Kernel for Analysis and Application Development. *Proceedings of the 36th Annual Simulation Symposium*, pp. 216-223, 2003.

MEHL92 Mehl H. A Deterministic Tie-breaking Scheme for Sequential and Distributed Simulation. *Proceedings of the 6th Workshop on Parallel and Distributed Simulation*, pp. 199-200, 1992.

MEND95 Mendenhall W. and Sincich T. Statistics for Engineering and the Sciences, 4th edition. Prentice-Hall, 1995.

MEYE99 Meyer, R.A. and Bagrodia, R.L. Path Lookahead: A Data Flow View of PDES Models. *Proceedings of the 13th Workshop on Parallel and Distributed Simulation*, pp. 12-19, 1999.

MEYE00 Meyer, R.A., Martin, J.M. and Bagrodia, R.L. Slow Memory: the Rising Cost of Optimism. *Proceedings of the 14th Workshop on Parallel and Distributed Simulation*, pp. 45-52, 2000.

MISR86 Misra, J. Distributed Discrete-Event Simulation. *ACM Computing Surveys*, 18 (1), pp. 39-65, 1986.

NICO91 Nicol, D.M. Performance Bounds on Parallel Self-initiating Discrete Event Simulations. *ACM Transactions on Modeling and Computer Simulation*, 1 (1), pp. 24-50, January, 1991.

NICO93 Nicol, D.M. The Cost of Conservative Synchronization in Parallel Discrete Event Simulation. *Journal of ACM*, 40 (2), pp. 304-333, 1993.

NICO02 Nicol, D. Analysis of Composite Synchronization. *Proceedings of the 16th Workshop on Parallel and Distributed Simulation*, pp. 115-124, 2002

NEGG98 Neggers, J. and Kim, H.S. Basic Posets. World Scientific Publishing, 1998.

ONGG02 Onggo, B.S.S. and Teo, Y.M. Performance Trade-off in Distributed Simulation. *Proceedings of the 6th IEEE International Workshop on Distributed Simulation and Real Time Applications*, IEEE Computer Society Press, Fort Worth, Texas, USA, pp. 77-84, October, 2002.

PARK04 Park, A., Fujimoto, R. and Perumalla, K. Conservative Synchronization of Large-Scale Network Simulations. *Proceedings of the 18th Workshop on Parallel and Distributed Simulation*, pp. 153-161, 2004.

PEAC79 Peacock, J.K., Wong, J.W. and Manning, E.G. Distributed Simulation using a Network of Processors. *Computer Networks*, 3 (1), pp. 44-56, 1979.

PIRL97 Pirlot, M. and Vinck, P. Semiorders. Kluwer Academic Publishers, Dordrecht, 1997.

PREI90 Preiss, B.R. Performance of a Discrete-event Simulation on a Multiprocessor Using Optimistic and Conservative Synchronization. *International Conference on Parallel Processing*, pp. 218-222, 1990.

PREI95 Preiss, B.R. and Loucks, W.M. Memory Management Techniques for Time Warp on a Distributed Memory Machine. *Proceedings of the 9th Workshop on Parallel and Distributed Simulation*, pp. 30-39, 1995.

QUAG99 Quaglia, F. Combining Period and Probabilistic Checkpointing in Optimistic Simulation. *Proceedings of the 13th Workshop on Parallel and Distributed Simulation*, pp. 109-116, May 1-4, 1999.

RAO98 Rao D.M., Thondugulam, N.V., Radhakrishnan, R. and Wilsey, P.A. Unsynchronized Parallel Discrete Event Simulation. *Proceedings of the Winter Simulation Conference*, vol: 2, pp. 1563–1570, 1998.

RAYN91 Raynal, M., Schiper, A. and Toueg, S. The Causal Ordering Abstraction and a Simple Way to Implement It. *Information Processing Letter*, 39 (6), pp. 343-350, 1991.

RONN94 Ronngren, R. and Ayani, R. Adaptive Checkpointing in time warp. *Proceedings of the 8th Workshop on Parallel and Distributed Simulation*, 24 (1), pp. 110-117, 1994.

RONN95 Ronngren, R. Performance Issues in Discrete Event Simulation. PhD Thesis, Department of Teleinformatics, Kungl Tekniska Hogskolan, Sweeden, 1995.

RONN99 Ronngren, R. and Liljenstam, M. On Event Ordering in Parallel Discrete Event Simulation, *Proceedings of 13th Workshop on Parallel and Distributed Simulation*, pp. 38-45, 1999.

ROSE99 Rosen, K.H. Discrete Mathematics and Its Applications, 4th edition. McGraw-Hill, 1999.

RUSS87 Russel, E.C. SIMSCRIPT II.5 Programming Language. CACI, La Jolla, California, 1987.

SCHI89 Schiper, A., Eggli, J. and Sandoz, A. A New Algorithm to Implement Causal Ordering. *Proceedings of the 3rd International Workshop on Distributed Algorithms*, Lecture Notes in Computer Science no. 392, Springer-Verlag, pp. 219-232, 1989.

SCHW94 Schwarz, R. and Mattern, F. Detecting Causal Relationships in Distributed Computations: In Search of the Holy Grail. *Distributed Computing*, 7 (3), pp. 149-174, 1994.

SHAS88 Shasha, D. and Snir, M. Efficient and Correct Execution of Parallel Programs that Share Memory. *ACM Transaction on Programming Languages and Operating Systems*, 10 (2), pp. 282-312, 1988.

SHOR97 Shorey, R., Kumar, A. and Rege, K.M. Instability and Performance Limits of Distributed Simulators of Feedforward Queueing Networks. *ACM Transactions on Modeling and Computer Simulation*, 7 (2), pp. 210-238, 1997.

SKOL96 Skold, S. and Ayani, R. Event sensitive state saving in Time Warp Parallel Discrete event simulation. *Proceedings of Winter Simulation Conference*, pp. 653-660, 1996.

SOKO88 Sokol, L.M. Briscoe, D. and Wieland, A. MTW: A Strategy for Scheduling Discrete Simulation Events for Concurrent Execution. *Proceedings of the SCS Multiconference on Distributed Simulation*, 1988.

SOKO91 Sokol, L.M., Weisman, J.B. and Mutchler, P.A. MTW: An Empirical Performance Study. *Proceedings of the 1991 Winter Simulation Conference*, pp. 557-563, 1991.

SOLI99 Soliman, H.M. On the Selection of the State Saving Strategy in Time Warp Parallel Simulations. *Transaction of the Society for Computer Simulation International*, 16 (1), pp. 32-36, 1999.

SONG00 Song, H.Y., Meyer, R.A. and Bagrodia, R. An Empirical Study of Conservative Scheduling. *Proceedings of the 14th Workshop on Parallel and Distributed Simulation*, pp. 165-172, 2000.

SONG01 Song, H.Y. A Probabilistic Performance Model for Conservative Simulation Protocol. *Proceedings of the 15th Workshop on Parallel and Distributed Simulation*, pp. 200-207, 2001.

STAL04 Stallings, W. Data and Computer Communications, 7th edition. Prentice Hall, 2004.

STEI92 Steinman, J.S. SPEEDES: A Multiple Synchronization Environment for Parallel Discrete-Event Simulation. *International Journal in Computer Simulation*, vol 2, pp. 251-286, 1992.

STEI93 Steinman, J.S. Breathing Time Warp. *Proceedings of the 7th Workshop on Parallel and Distributed Simulation*, 23 (1), pp. 109-118, 1993.

STEI94 Steinman, J.S. Discrete Event Simulation and the Event Horizon. *Proceedings of the 8th Workshop on Parallel and Distributed Simulation*, 24 (1), pp. 39-49, 1994.

TEO94 Teo Y.M. and Tay, S.C. Efficient Algorithms for Conservative Parallel Simulation of Interconnection Networks. *Proceedings of the International Symposium on Parallel Architectures, Algorithms and Networks*, pp. 286-293, 1994.

TEO95 Teo, Y.M. and Tay, S.C. Modeling an Efficient Distributed Simulation of Multistage Interconnection Network. *Proceedings of International Conference on Algorithms and Architectures for Parallel Processing*, pp. 83-92, 1995.

TEO99 Teo, Y.M., Wang, H., and Tay S.C. A Framework of Analyzing Parallel Simulation Performance. *Proceedings of the 32nd Annual Simulation Symposium*, IEEE Computer Society Press, San Diego, USA, pp. 102-109, April 1999.

TEO01 Teo, Y.M., Onggo, B.S.S., and Tay, S.C. Effect of Event Orderings on Memory Requirement in Parallel Simulation, *Proceedings of the 9th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, pp. 41-48, IEEE Computer Society Press, USA, 2001.

TEO02A Teo, Y.M and Ng, Y.K. SPaDES/Java: Object-Oriented Parallel Discrete-Event Simulation. *Proceedings of the 35th Annual Simulation Symposium*, pp. 222-229, IEEE Computer Society Press, San Diego, USA, April 2002.

TEO02B Teo, Y.M., Ng, Y.K. and Onggo, B.S.S. Conservative Simulation Using Distributed Shared Memory. *Proceedings of the 16th ACM/IEEE/SCS Workshop on Parallel and Distributed Simulation*, pp. 3-10, IEEE Computer Society Press, Washington D.C., USA, May 2002.

TEO04 Teo, Y.M. and Onggo, B.S.S. Formalization and Strictness of Simulation Event Orderings. *Proceedings of the 18th IEEE/ACM/SCS Workshop on Parallel and Distributed Simulation*, pp. 89-96, IEEE Computer Society Press, Kufstein, Austria, May 16-19, 2004 (nominated for best paper award and to appear in the special edition of the Transaction of Society for Modeling and Simulation).

THAI03 Thain, D., Tannenbaum, T. and Livny M. Condor and the Grid, a chapter in Grid Computing Making the Global Infrastructure a Reality, Berman F., Fox, G.C. and Hey, A.J.G (editors). John Wiley & Sons, Inc., 2003.

THON99 Thondugulam, N.V., Rao, D.M., Radhakrishnan, R. and Wilsey, P.A. Relaxing Causal Constraints in PDES. *Proceedings of the 13th International and 10th Symposium on Parallel and Distributed Processing*, pp. 696 –700, 1999.

TRIV02 Trivedi, K.S. Probability and Statistics with Reliability, Queuing and Computer Science Applications, 2nd edition. John Wiley & Sons, Inc., 2002.

TROP02 Tropper, C. Parallel Discrete-Event Simulation Applications. *Journal of Parallel and Distributed Computing*, 62 (3), pp. 327-335, 2002.

TROT92 Trotter, W.T. Combinatorics and Partially Ordered Sets: Dimension Theory. Johns Hopkins University Press, 1992.

TURN92 Turner, S. and Xu, M. Performance Evaluation of the Bounded Time Warp Algorithm. *Proceedings of the 6th Workshop on Parallel and Distributed Simulation*, pp. 117-126, 1992.

UNGE01 Unger, B., Xiao, Z, Cleary, J., Tsai, J.J. and Williamson, C. Parallel Shared-Memory Simulator Performance for Large ATM Networks. *ACM Transaction on Modeling and Computer Simulation*, 10 (4), pp. 358-391, 2001.

WAGN89 Wagner, D.B. and Lazowska, E.D. Parallel Simulation of Queuing Networks: Limitations and Potentials. *ACM Performance Evaluation Review*, 17 (1), pp.146-155, 1989.

WANG99 Wang, J. and Keshav S. Efficient and Accurate Ethernet Simulation. *Proceedings of the 24th Conference on Local Computer Networks*, pp. 182-201, 1999.

WANG00 Wang, H., Teo, Y.M. and Tay, S.C. An Analytic Method for Predicting Simulation Parallelism. *Proceedings of the 33rd Annual Simulation Symposium*, pp. pp. 211-218, 2000.

WEAV94 Weaver, D. and Germond T. The SPARC Architecture Manual. SPARC International, version 9. Prentice Hall, 1994.

WEST96 West D. and Panesar K. Automatic Incremental State Saving. *Proceedings of the 10th Workshop on Parallel and Distributed Simulation*, pp. 78-85, 1996.

WIEL97 Wieland, F. The Threshold of Event Simultaneity. *Proceedings of the 11th Workshop on Parallel and Distributed Simulation*, pp. 56-59, 1997.

WONG95 Wong, Y.C. and Hwang, S.Y. Prediction of Memory Consumption in Conservative Parallel Simulation. *Proceedings 9th Workshop on Parallel and Distributed Simulation*, pp. 199-202, 1995.

WONN96 Wonnacott, P and David, B. The APOSTLE Simulation Language: Granularity Control and Performance Data, *Proceedings 10th Workshop on Parallel and Distributed Simulation*, pp. 114-123, 1996.

WOOD94 Wood, K.R. and Turner, S.J. A Generalized Carrier-Null Method for Conservative Parallel Simulation. *Proceedings of the 8th Workshop on Parallel and Distributed Simulation*, IEEE Computer Society Press, pp. 50-57, 1994.

XIAO99 Xiao, Z. and Unger, B. Scheduling Critical Channels in Conservative Parallel Simulation. *Proceedings of the 13th Workshop on Parallel and Distributed Simulation*, pp. 20-28, 1999.

XU01 Xu, J.S. and Chung M.J. Predicting the Performance of Synchronous Discrete Event Simulation System. *International Conference on Computer-Aided Design*, pp. 18-23, 2001.

XU03 Xu, D., Riley, G.F., Ammar, M.H. and Fujimoto, R. Enabling Large-Scale Multicast Simulation by Reducing Memory Requirements. *Proceedings of the 17th Workshop on Parallel and Distributed Simulation*, pp. 69-76, 2003.

YOUN99 Young, C.H., Radhakrishnan and Wilsey, P. Optimism: Not Just For Event Execution Anymore. *Proceedings of the 13th Workshop on Parallel and Distributed Simulation* , pp. 136-143, 1999.

ZENG04 Zeng, Y., Cai, W. and Turner, S.J. Batch Based Cancellation: a Rollback Optimal Cancellation Scheme in Time Warp Simulations. *Proceedings of the 18th Workshop on Parallel and Distributed Simulation*, pp. 78-86, 2004.

ZHAN89 Zhang, M and Yum, T.-S. Comparisons of Channel Assignment Strategies in Cellular Mobile Telephone Systems. *IEEE Transaction on Vehicle Technology*, 38 (4), pp. 211-215, 1989.

ZHAN01 Zhang, J.L. and Tropper, C. The Dependence List in Time Warp. *Proceedings of the 15th Workshop on Parallel and Distributed Simulation*, pp. 35-45, 2001.

ZHOU02 Zhou, S.P., Cai, W.T., Turner, S.J. and Lee, B.S. Critical Causality in Distributed Environment. *Proceedings of the 16th Workshop on Parallel and Distributed Simulation*, pp. 53-59, 2002.

ZYMA03 Szymanski, B.K, Liu, Y. and Gupta, R. Parallel Network Simulation under Distributed Genesis. *Proceedings of the 17th Workshop on Parallel and Distributed Simulation*, pp. 61-68, 2003.

Appendix A

Time, Space and Strictness Analyzer

Time, Space and Strictness Analyzer (TSSA) is a measurement tool that we have developed. It is used to measure:

1. the strictness (ς) of event orderings at the physical system layer, the simulation model layer, and the simulator layer;
2. the time (Π^{ord}) and space (M^{ord}) performance at the simulation model layer.

A.1 Algorithm

Unlike the strictness of event orderings at the physical system and simulation model layers, the strictness of event orderings at the simulator is affected by overhead events. Because of this, TSSA operates in two modes.

In the first mode, TSSA simulates event orderings at the physical layer and at the simulation model layer. In the simulation, TSSA measures the strictness of the given event ordering. For event ordering at the simulation model layer, Π^{ord} and M^{ord} are also measured. It requires two inputs: a log file produced by a sequential simulator and the event order *R*. The algorithm is shown in Figure A.1. First, TSSA reads the log file and

creates a set of events *E*. Lines 3 – 6 distribute events to a number of lists according to their LPs (one list for each LP). Next, TSSA retrieves a set of events where each of them has the smallest timestamp from each list, and simulates the event execution based on the given event order *R* (lines 7 – 17). TSSA, then, updates the necessary accounting variables (lines 18 – 27). These steps are repeated until all events have been processed. Finally, TSSA computes Π^{ord}, M^{ord} and ς (lines 29 – 31).

```
TIME, SPACE AND STRICNESS ANALYZER
 1. E ← Log File
 2. while (E ≠ ∅) {
 3.    while (∃i L[i] = ∅) {
 4.        E ← E − {e}
 5.        L[e.lp] ← L[e.lp] ∪ e
 6.    }
 7.    events_executed ← 0; S ← ∅
 8.    for each L[i] {
 9.       h ← head(L[i])
10.       if (h can be processed based on event order R){
11.          L[i] ← L[i] − {h}
12.          ELsize[i]--
13.          for all e ∈ h.NE { ELsize[e.lp]++ }
14.          events_executed++;
15.       }
16.       S ← S ∪ {h}
17.    }
18.    for each EL[i] { maxEL[i] ← max(maxEL[i], ‖ELsize[i]‖) }
19.    tot_events ← tot_events + events_executed
20.    comparable ← 0
21.    for each x∈S {
22.       for each y∈S, x≠y {
23.          if (y is dependent on x) comparable++
24.       }
25.    }
26.    strict += comparable / (‖S‖×(‖S‖-1)/2)
27.    time_step++
28. }
29. M^ord ← sum of maxEL[i]
30. Π^ord ← tot_events/time_step
31. ς ← strict / time_step;
```

Figure A.1: Time, Space and Strictness Analyzer – Mode 1

In the second mode, TSSA is used to measure the strictness of event orderings at the simulator layer. It requires two inputs: a set of log files produced by SPaDES/Java (one log file for each PP), and the mapping between LPs and PPs. The detail algorithm is shown in Figure A.2.

```
TIME, SPACE AND STRICNESS ANALYZER
 1. Merge Log Files to produce E
 2. while (E ≠ ∅) {
 3.    do {
 4.       E ← E - {e}
 5.       L[e.lp] ← L[e.lp] ∪ e
 6.    } until each L[i] contains at least one real event
 7.    S ← ∅
 8.    for each L[i] {
 9.       h ← head(L[i])
10.       S ← S ∪ {h}
11.       if (h can be processed) L[i] ← L[i] - {h}
12.    }
13.    comparable ← 0
14.    for each x∈S {
15.       for each y∈S, x≠y {
16.          if (y is dependent on x) comparable++
17.       }
18.    }
19.    strict += comparable / (‖S‖×(‖S‖-1)/2)
20.    time_step++
21. }
22. ς ← strict / time_step;
```

Figure A.2: Time, Space and Strictness Analyzer – Mode 2

First, a set of events *E* is constructed from the log files. In this mode, *E* comprises real events and overhead events. Next, events are grouped according to their LPs (lines 3 – 6). Events occurring at the same LP are stored in the same list. Next, TSSA prepares a set of events comprises one event for each LP (lines 7 – 12). Lines 13 – 20 measure the strictness of event ordering at the current timestep. These steps are repeated until all real events have been processed. Finally, the event ordering strictness is derived from the average strictness at every timestep.

Mapping information is used to decide whether two events are executed at the same PP. If real events *x* and *y* are executed at the same PP, then *x* is said to be comparable to *y* at the simulation layer. This is because either *x* must be executed before *y* or vice versa. At the simulator layer, real event *x* must also be executed before real event *y*, if *x* is executed before an overhead event *z*, and *z* is executed before *y*. Therefore, the number of PPs and overhead events affect the event ordering strictness at the simulator layer. Two concurrent events at the simulation model layer can be comparable at the simulator layer.

A.2 Validation

Wang et al. has developed an algorithm that predicts the upper bound on model parallelism (Π^{ord}) [WANG00]. We compare the results of their algorithm with the Π^{ord} of partial event order measured using TSSA. Figure A.3 and Figure A.4 show that the TSSA results are bounded by the results of the algorithm.

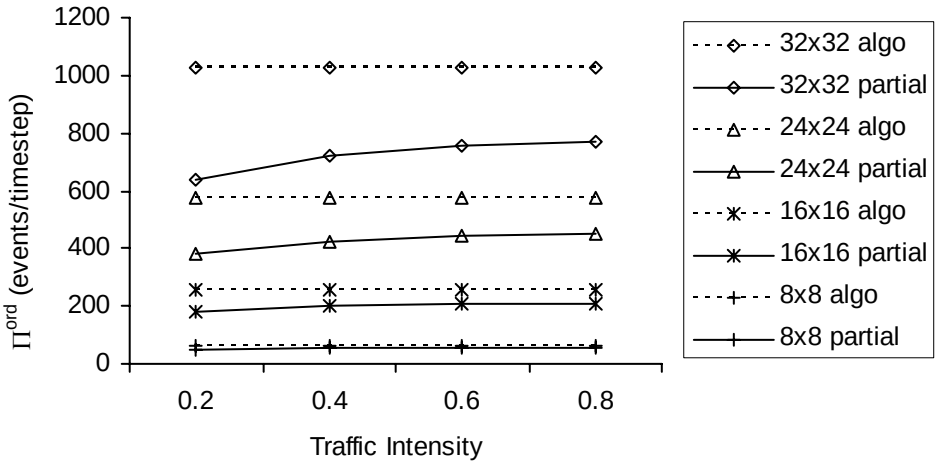


Figure A.3: TSSA Validation – MIN

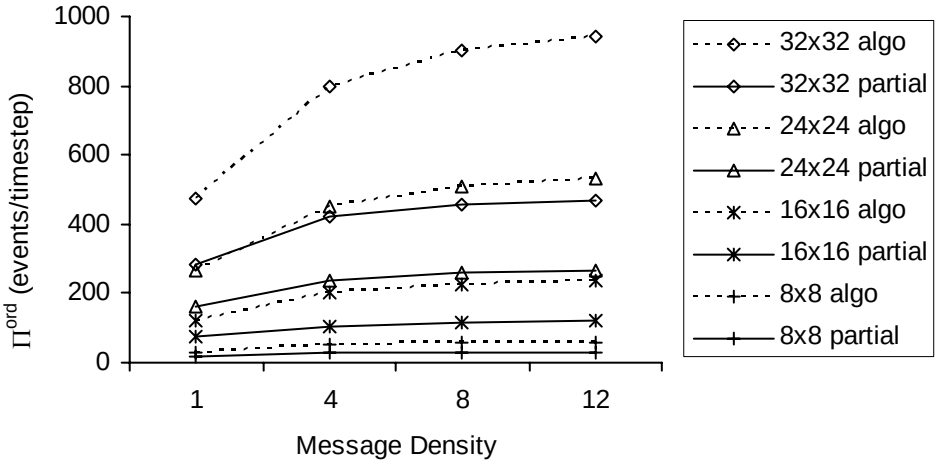


Figure A.4: TSSA Validation –PHOLD

The *stricter* relation among event orders that have been proved analytically in Theorems 3.1 – 3.6 (summarized in Figure 3.8) are validated against the *strictness* measurement results shown in Figures 4.21 and 4.22. The measurement results confirm the analytical results.

Appendix B

SPaDES/Java Simulation Library

The first version of SPaDES/Java supports the process-oriented world-view [TEO02A]. In this thesis, we extend it to support the event-oriented world-view. A programming template is provided to help a user in writing a program using SPaDES/Java parallel simulator. The parallelization is transparent to the user. A user only needs to modify the content of six main sections, i.e., parameters, state variables, simulator, kernel, event handlers, and a configuration file.

B.1 Parameters

User specifies the simulation parameters in class *parameter* that is derived from class *SpadesParam*. The template for class parameter is shown in Figure B.1.

```
1. import spades.*;
2. public class <name> extends SpadesParam
3. {
4.    <variable lists>
5. }
```

Figure B.1: SPaDES/Java – Parameters

In MIN simulation, for example, the parameters include problem size, mean inter-arrival time and mean service time.

B.2 State Variables

The state variables are specified in class *StateVariable* that is derived from class *SpadesStateVar*. The template for class *StateVariable* is shown in Figure B.2.

```
1. import spades.*;
2. public class <name> extends SpadesStateVar
3. {
4.    <variable lists>
5. }
```

Figure B.2: SPaDES/Java – State Variables

Note that in parallel simulation, each LP must have a disjoint set of state variables. SPaDES/Java library only supports a simulation whereby each LP has the same type of state variables. In MIN simulation, for example, each LP maintains state variables such as a queue and the status of server (busy or idle).

B.3 Simulator

The class *Simulator* handles all simulation initialization. First, a user must provide all information required by the initialization process such as simulation parameters, state variables, and communication channels (line 8). The underlying protocol is CMB protocol, therefore, static communication channels must be specified before the simulation begins (line 10). In SPaDES/Java, this is done by specifying a matrix called *channelMatrix* where *channelMatrix[i][j]* is true if LP_i may send an event to LP_j. LPs

and their communication channels are initialized in line 11. Next, the start events are sent to their respective LPs to be executed upon LPs' activation (line 12). After completing the initialization process, class *Simulator* activates all LPs (line 13). The execution of start events triggers new events, and subsequently, the execution of the new events generate another set of events, and so on. Class *Simulator* is derived from class *SpadesSimulator* as shown in Figure B.3.

```
 1. import java.io.*;
 2. import java.text.*;
 3. import spades.*;

 4. public class <name> extends SpadesSimulator {
 5.   public MINParam params;
 6.   public MINStateVar state;

 7.   public static void main(String[] args) {
 8.     // add information for initialization here
 9.     <name> simulator = new <name>;
10.     simulator.createCommChannelMatrix();
11.     simulator.initialize();
12.     simulator.initStartEvents();
13.     simulator.startSimulation();
14.   }
15. }
```

Figure B.3: SPaDES/Java – Simulator

B.4 Kernel

The main task of a kernel (or simulation engine) is to execute events. A kernel is activated on each PP and it serves one or more LPs that are mapped onto the PP. A user only needs to define procedure *printReport()* to show the simulation results and performance metrics. The class *Kernel* is derived from class *SpadesKernel* and its template is shown in Figure B.4.

```
 1. import java.io.*;
 2. import java.text.*;
 3. import spades.*;

 4. public class <name> extends SpadesKernel {
 5.   public MINParam params;
 6.   public MINStateVar state;

 7.   public static void main(String[] args) {
 8.     <name> kernel = new <name>;
 9.     kernel.startSimulation();
10.     while (!kernel.simulationCompleted()) {}
11.     kernel.printReport();
12.     System.exit(0);
13.   }
14. }
```

Figure B.4: SPaDES/Java – Kernel

B.5 Event Handler

SPaDES/Java simulator adopts an event-oriented world view; therefore, a user must define a handler for each event. The handler is derived from class *SpadesEvent*. The template for an event handler is shown below Figure B.5.

```
 1. import java.io.*;
 2. import java.text.*;
 3. import spades.*;

 4. public class <name> extends SpadesEvent
 6. {
 7.   public <name>(int anLp, double aTs, int anAnteLp)
 8.   {
 9.     super(anLp, aTs, anAnteLp);
10.   }
11.   public void execute(SpadesStateVar s, SpadesParam p,
12.     SpadesRandom r)
13.   {
14.     <code for event handler is implemented here>
15.   }
16. }
```

Figure B.5: SPaDES/Java – Event Handler

Let us consider the event arrival in MIN. Event arrival at the leftmost stage generates the next event arrival (lines 5-7). If a job arrives on a busy switch, the signal will join the queue; otherwise the signal will be processed by the switch and a new departure event will be generated (lines 9-14).

```
 1. public void execute(SpadesStateVar s, SpadesParam p,
 2. SpadesRandom r)
 3. {
 4.   /* Schedule next arrival. */
 5.   if ((lp%((MINParam)p).col_size) == 0)
 6.     generate (new MINArrival(lp, lp, ts +
 7.       r.expon(((MINParam)p).mean_interarrival)));
 8.   /* Check to see whether server is busy. */
 9.   if (s.server_status == BUSY) s.queue.insert(ts);
10.   else {
11.     s.server_status = BUSY;
12.     generate(new MINDeparture(lp, lp, ts +
13.       r.expon(((MINParam)p).mean_service)));
14.   }
15. }
```

Figure B.6: SPaDES/Java – Event Handler Example

B.6 Configuration File

A configuration file is needed to configure the processor mapping and to set the parameters' values. An example of a configuration file is shown in Figure B.7. It consists of two sections, i.e., general configuration and problem (application) specific configuration. The keywords in the general configuration are fixed for all applications. This includes the algorithm used (currently, 1 for sequential, 2 for CMB with demand driven optimization), the simulation duration, the problem size (number of LPs), and the file name consisting of mapping information. In the problem specific section, a user can define his/her parameters and their values. Figure B.7 shows the example of MIN configuration file.

```
#CONFIGURATION FILE FOR SIMJAVA
#GENERAL CONFIGURATION
Algorithm: 2
Duration: 10000
Problem size: 256
Mapping file: 256LP-8PP.txt
==================
#PROBLEM SPECIFIC CONFIGURATION
Row size: 16
Column size: 16
Mean interarrival: 6.25
Mean service time: 5
Transfer time: 1
==================
```

Figure B.7: SPaDES/Java – MIN Configuration File

B.7 Validation

To validate SPaDES/Java, we compare our simulation results with the results of analytical models or the results obtained by other researchers.

B.7.1 MIN

MIN is an open system, therefore, the utilization of each server (U) is modeled as $U = \max(1, \rho)$, where $\rho = \lambda/\mu$, λ is the arrival rate, and μ is the service rate [BOLC98, TRIV02]. The average queue size (L) is modeled as $L = \rho^2 / (1-\rho)$ [BOLC98, TRIV02].

Table B.1 shows that the simulation result is very close to the analytical result. The t-test produces a 90% confidence interval [-0.002, 0.004] for the difference between the utilization measured from simulation and the utilization derived from the analytical model. Similarly, the 90% confidence interval for the difference between the average queue size measured from simulation and the average queue size derived from the

analytical model is [-0.029, 0.076]. Thus, with 90% confidence, we can say that there is no significant difference between the simulation result and the analytical result.

Parameters		Service center utilization		Average queue size	
Problem size	**ρ**	**Simulation**	**Analytical**	**Simulation**	**Analytical**
8×8	0.2	0.21	0.20	0.05	0.05
	0.4	0.40	0.40	0.28	0.27
	0.6	0.60	0.60	0.89	0.90
	0.8	0.80	0.80	3.24	3.20
16×16	0.2	0.20	0.20	0.05	0.05
	0.4	0.40	0.40	0.25	0.27
	0.6	0.60	0.60	0.91	0.90
	0.8	0.79	0.80	3.18	3.20
32×32	0.2	0.20	0.20	0.05	0.05
	0.4	0.40	0.40	0.26	0.27
	0.6	0.61	0.60	0.92	0.90
	0.8	0.80	0.80	3.16	3.20

Table B.1: Validation – MIN

B.7.2 PHOLD

PHOLD is a closed system, therefore we use the convolution algorithm [BOLC98, TRIV02] to approximate ρ as shown in Equation B.1. The utilization of each server (*U*) can be derived from $U = \max(1, \rho)$ as in MIN.

$$\rho = G(K-1) / (\mu \times G(K)) \qquad \text{(B.1)}$$

where $G(K) = \sum_{\sum_{i=1}^{N} k_i = K} \prod_{i=1}^{N} F_i(k_i)$ and $F_i(k_i) = (1/\mu_i)^{ki}$ and *K* is the total number of jobs in the system and *N* is the number of service center.

To approximate the average queue size (*L*), we use the same convolution algorithm to compute Equation B.2. The terms *G* and *K* are the same as in the Equation B.1.

$$L = \left(\sum_{k=1}^{K} (1/\mu)^k \times \frac{G(K-k)}{G(K-1)} \right) - 1 \qquad \text{(B.2)}$$

The main limitation of convolution algorithm is that G(*K*) calculation for large value of *K* results in a very big number (overflow). Therefore, we can only validate for the smaller problem sizes as shown in Table B.2.

Parameters		**Service center Utilization**		**Average queue size**	
Problem size	**Message density**	**Simulation**	**Analytical**	**Simulation**	**Analytical**
4×4	1	0.478	0.516	0.416	0
	4	0.791	0.810	3.033	3
	8	0.888	0.895	6.910	7
	12	0.920	0.928	10.867	11
5×5	1	0.469	0.510	0.427	0
	4	0.786	0.806	3.039	3
	8	0.881	0.893	6.919	7
	12	0.925	0.926	10.865	11
6×6	1	0.469	0.507	0.427	0
	4	0.787	0.804	3.038	3
	8	0.881	0.892	6.919	7
	12	0.917	0.925	10.871	11

Table B.2: Validation – PHOLD

Table B.2 shows that the simulation result is very close to the analytical result except for message density of 1 (*K* = *N*). This is because the convolution algorithm will not produce an accurate result if the total number of jobs is less than or equal to the number of service centers [BOLC98, TRIV02]. If we ignore the result for message density of 1, the t-test produces a 90% confidence interval [-0.015, -0.008] for the difference between the utilization measured from simulation and the utilization derived from the analytical model. Similarly, the 90% confidence interval for the difference between the average queue size measured from simulation and the average queue size derived from the analytical model is [-0.107, -0.013]. Thus, with 90% confidence, we can say that the

simulation result is slightly less than the analytical result. However, the average difference is very small, i.e., 0.01 for the utilization, and 0.06 for the average queue size.

B.7.3 Ethernet

For the Ethernet simulation, we compare our simulation result with the analytical results published by other researchers [STAL04, CHUA01]. Stallings provides an analytical model for the maximum channel utilization [STAL04]. Therefore, our simulation result must be within this boundary as shown in Table B.3.

Parameters		Channel Utilization		
Number of Stations	**Frame Size (bytes)**	**Simulation**	**Analytical**	**Bound**
24	64	18%	17%	23%
	512	50%	62%	71%
	1024	60%	77%	83%
	1500	65%	83%	88%
48	64	16%	9%	23%
	512	45%	44%	70%
	1024	57%	61%	83%
	1500	62%	70%	88%
72	64	14%	6%	23%
	512	42%	34%	70%
	1024	54%	51%	83%
	1500	59%	60%	88%
96	64	13%	5%	23%
	512	39%	28%	70%
	1024	51%	43%	82%
	1500	56%	54%	88%

Table B.3: Channel Utilization – Ethernet

The difference between the simulation result and analytical result developed by Chuang and Zukerman [CHUA01] is mainly due to the different assumptions used by the analytical model. The analytical model assumes that each station is ready to transmit data at any time, the whole frame contains useful data (thus ignoring the header information), and there is no inter-frame gap. However, statistically, the difference is insignificant as

shown by the t-test result (Figure B.8). The *t* statistic value is not in the rejection area. Hence, within the 95% confidence interval, there is no significant difference between the simulation result and the analytical result.

t-Test: Paired Two Sample for Means

	simulation	analytical
Mean	0.44	0.44
Variance	0.03	0.06
Observations	16	16
Pearson Correlation	0.96	
Hypothesized Mean Difference	0	
df	15	
t Statistic	-0.04	
P(T<=t) two-tail	0.97	
t Critical two-tail	2.13	

Figure B.8: T-Test Result – Channel Utilization

Appendix C

Experimental Results

C.1 MIN

Physical System Layer

Problem size	Traffic intensity	Π^{prob} (events / min)	M^{prob} (units)
8×8	0.2	5	200
	0.4	10	414
	0.6	16	723
	0.8	22	1,371
16×16	0.2	21	844
	0.4	43	1,648
	0.6	64	2,862
	0.8	87	5,541
24×24	0.2	47	1,895
	0.4	95	3,675
	0.6	146	6,642
	0.8	194	12,522
32×32	0.2	83	3,323
	0.4	168	6,532
	0.6	254	11,348
	0.8	341	21,454

Simulation Model Layer

Metrics	Problem Size	Partial	CMB	TI(5)	TS	Total
Π^{ord} (events / timestep)	8×8	57	57	25	8	1
	16×16	211	211	88	26	1
	24×24	453	451	187	51	1
	32×32	768	766	317	83	1
M^{ord} (units)	8×8	2,145	2,165	302	92	88
	16×16	8,151	8,198	1,192	301	296
	24×24	16,516	16,642	2,646	656	640
	32×32	27,722	27,909	4,652	1,102	1,093
Average Memory (units)	8×8	879	888	73	68	64
	16×16	2,822	2,845	268	253	245
	24×24	5,164	5,234	589	561	376
	32×32	7,766	7,903	1,016	975	958

Simulator Layer

No of Processors	Problem size	Π^{sync} (events / s)	M^{sync} (units)
4	8×8	6,642	172
	16×16	12,521	616
	24×24	15,522	1,324
	32×32	16,073	2,264
8	8×8	9,414	232
	16×16	17,649	736
	24×24	20,755	1,498
	32×32	20,952	2,452

Normalized Event Parallelism

Problem size	Π^{prob}	Π^{ord}	Π^{sync}
8×8	8	57	5.83
16×16	26	211	5.97
24×24	51	451	6.04
32×32	83	766	6.11

Total Memory Requirement (units)

Problem size	M^{prob}	M^{ord}	M^{sync}
8×8	1,282	264	232
16×16	5,194	924	736
24×24	12,522	1,835	1,498
32×32	19,915	2,633	2,452

Strictness

		Problem Size			
Layer		**8×8**	**16×16**	**24×24**	**32×32**
Physical System		0.9701	0.9776	0.9803	0.9822
Simulation Model	Partial	0.1078	0.1838	0.2362	0.2641
	CMB	0.1175	0.2106	0.2788	0.3236
	TI(5)	0.5795	0.5681	0.5616	0.5601
	Total	1.0000	1.0000	1.0000	1.0000
Simulator (4 PPs)		0.9802	0.9853	0.9904	0.9955
Simulator (8 PPs)		0.9733	0.9813	0.9893	0.9920

C.2 PHOLD

Physical System Layer

Problem size	**Message density**	**Π^{prob} (events / min)**	**M^{prob} (units)**
8×8	1	13	472
	4	22	1,293
	8	25	2,028
	12	26	2,593
16×16	1	53	1,939
	4	88	5,233
	8	99	8,117
	12	104	10,356
24×24	1	118	4,409
	4	199	11,644
	8	224	18,302
	12	235	23,090
32×32	1	209	7,864
	4	354	20,816
	8	400	32,518
	12	417	41,463

Simulation Model Layer

Metrics	Problem Size	Partial	CMB	TI(5)	TS	Total
Π^{ord} (events / timestep)	8×8	30	22	27	9	1
	16×16	106	79	89	26	1
	24×24	236	176	191	53	1
	32×32	420	313	330	87	1
M^{ord} (units)	8×8	468	305	321	256	256
	16×16	1,878	1,299	1,237	1,024	1,024
	24×24	4,245	2,836	2,764	2,304	2,304
	32×32	7,584	4,952	4,892	4,096	4,096
Average Memory (units)	8×8	84	66	67	62	58
	16×16	338	260	265	245	229
	24×24	761	580	596	551	513
	32×32	1,354	1,060	1,026	979	912

Simulator Layer

No of Processors	Problem size	Π^{sync} (events / s)	M^{sync} (units)
4	8×8	3,700	416
	16×16	4,533	1,248
	24×24	5,289	2,688
	32×32	5,839	4,160
8	8×8	4,492	720
	16×16	8,525	1,728
	24×24	10,349	3,360
	32×32	11,274	4,992

Normalized Event Parallelism

Problem size	Π^{prob}	Π^{ord}	Π^{sync}
8×8	8	22	1.26
16×16	26	79	1.94
24×24	51	176	2.31
32×32	83	313	2.56

Total Memory Requirement (units)

Problem size	M^{prob}	M^{ord}	M^{sync}
8×8	1,293	264	720
16×16	5,233	1,036	1,728
24×24	11,644	2,317	3,360
32×32	20,816	4,106	4,992

Strictness

Layer		8×8	16×16	24×24	32×32
Physical System		0.9686	0.9762	0.9790	0.9805
Simulation Model	Partial	0.4907	0.4848	0.4835	0.4835
	CMB	0.7726	0.7916	0.8019	0.8059
	TI(5)	0.5846	0.5661	0.5577	0.5528
	Total	1.0000	1.0000	1.0000	1.0000
Simulator (4 PPs)		0.9773	0.9853	0.9914	0.9975
Simulator (8 PPs)		0.9718	0.9803	0.9883	0.9955

C.3 Ethernet

Π^{prob} (events / μs)

Number of Stations	Frame Size (bytes)			
	64	512	1024	1500
24	15	4	3	2
48	33	10	7	6
72	53	17	13	11
96	74	26	21	19

M^{prob} (unit)

Number of Stations	M^{prob}
24	192
48	384
72	576
96	768

Π^{ord} (events / timestep)

Number of Stations	Frame Size (bytes)			
	64	512	1024	1500
24	20	13	11	10
48	40	27	23	22
72	59	42	38	37
96	79	59	55	54

M^{ord} (unit)

Number of Stations	Frame Size (bytes)			
	64	**512**	**1024**	**1500**
24	1,063	1,113	1,150	1,129
48	2,856	4,534	4,556	4,561
72	5,456	10,235	10,352	10,361
96	10,429	16,516	16,516	18,417

Π^{sync} (events/s)

Number of Stations	Number of PPs			
	2	**4**	**6**	**8**
24	2,155	2,690	1,880	1,532
48	2,503	2,839	2,129	1,793
72	2,578	2,989	2,185	1,887
96	2,634	3,045	2,223	1,905

M^{sync} (unit)

Number of Stations	Number of PPs			
	2	**4**	**6**	**8**
24	96	432	984	2040
48	192	748	1920	3696
72	288	1056	2976	5328
96	384	1536	4032	7584

Appendix D

Validation of Performance Models

This appendix provides the validation of performance models given in the Equations 4.4 – 4.11.

T-Statistic Result for $\Pi^{prob} = 2 \times \rho \times \mu \times n \times n$ (Equation 4.4)

t-Test: Paired Two Sample for Means

	observed	model
Mean	927139.5	927494.7
Variance	1.15E+12	1.15E+12
Observations	12	12
Pearson Correlation	0.9999	
Hypothesized Mean Difference	0	
df	11	
t Statistic	-0.1	
P(T<=t) two-tail	1.796	
t Critical two-tail	2.201	

The t-test shows that the t-statistic result (-0.1) is not in the rejection area (-2.201 < t < 2.201). Therefore, with 95% confidence interval, there is no significant difference between the observed value and the model.

T-Statistic Result for $\Pi^{prob} = 2 \times c_1 \times \log(c_2 + m) \times \mu \times n \times n$ (Equation 4.5)

t-Test: Paired Two Sample for Means

	observed	model
Mean	1511824.5	1508497.1
Variance	2.396E+12	2.385E+12
Observations	12	12
Pearson Correlation	0.9954	
Hypothesized Mean Difference	0	
df	11	
t Statistic	0.077	
P(T<=t) two-tail	1.796	
t Critical two-tail	2.201	

The model uses c_1 = 3675.4 and c_2 = 3. Both constants (and all constants used by models in this appendix) are obtained using quasi-Newton method [MEND95]. The t-test shows that the t-statistic result (0.077) is not in the rejection area (-2.201 < t < 2.201). Therefore, with 95% confidence interval, there is no significant difference between the observed value and the model.

T-Statistic Result for $M^{prob} = c_1 \times n \times n + c_2 \times e^{\rho} + c_3 \times n \times n \times e^{\rho} + \varepsilon$ (Equation 4.6)

t-Test: Paired Two Sample for Means

	observed	model
Mean	4688.33	4827.09
Variance	38539151.88	37976516.54
Observations	12	12
Pearson Correlation	0.9927	
Hypothesized Mean Difference	0	
df	11	
t Statistic	-0.64	
P(T<=t) two-tail	1.796	
t Critical two-tail	2.201	

The model uses c_1 = -19.2, c_2 = 103.3, and c_3 = 17.5. The t-test shows that the t-statistic result (-0.64) is not in the rejection area (-2.201 < t < 2.201). Therefore, with 95%

confidence interval, there is no significant difference between the observed value and the model.

T-Statistic Result for $\Pi^{ord} = c_1 \times n \times n + c_2 \times \ln \rho + c_3 \times n \times n \times \ln \rho + \varepsilon$ (Equation 4.8)

t-Test: Paired Two Sample for Means

	observed	model
Mean	11224.33	11244.11
Variance	179720785.5	177447000.6
Observations	12	12
Pearson Correlation	0.9937	
Hypothesized Mean Difference	0	
df	11	
t Statistic	-0.045	
P(T<=t) two-tail	1.796	
t Critical two-tail	2.201	

The model uses c_1 = 6, c_2 = 8.2, and c_3 = 12.8. The t-test shows that the t-statistic result (-0.045) is not in the rejection area (-2.201 < t < 2.201). Therefore, with 95% confidence interval, there is no significant difference between the observed value and the model.

T-Statistic Result for $\Pi^{ord} = c_1 \times n \times n + c_2 \times \ln m + c_3 \times n \times n \times \ln m + \varepsilon$ (Equation 4.9)

t-Test: Paired Two Sample for Means

	observed	model
Mean	179.15	177.67
Variance	31635.16	31564.60
Observations	12	12
Pearson Correlation	0.9989	
Hypothesized Mean Difference	0	
df	11	
t Statistic	0.611	
P(T<=t) two-tail	1.796	
t Critical two-tail	2.201	

The model uses c_1 = 0.29, c_2 = -0.07, and c_3 = 0.07. The t-test shows that the t-statistic result (-0.045) is not in the rejection area (-2.201 < t < 2.201). Therefore, with 95%

confidence interval, there is no significant difference between the observed value and the model.

T-Statistic Result for $M^{ord} = c_1 \times n \times n + c_2 \times \rho + c_3 \times n \times n \times \rho + \varepsilon$ (Equation 4.10)

t-Test: Paired Two Sample for Means

	observed	model
Mean	9195.25	9174.43
Variance	79220757.66	79501320.97
Observations	12	12
Pearson Correlation	0.9991	
Hypothesized Mean Difference	0	
df	11	
t Statistic	0.195	
P(T<=t) two-tail	1.796	
t Critical two-tail	2.201	

The model uses c_1 = 7.75, c_2 = -1355, and c_3 = 22.45. The t-test shows that the t-statistic result (0.195) is not in the rejection area (-2.201 < t < 2.201). Therefore, with 95% confidence interval, there is no significant difference between the observed value and the model.

T-Statistic Result for $M^{ord} = c_1 \times n \times n + c_2 \times e^m + c_3 \times n \times n \times e^m + \varepsilon$ (Equation 4.11)

t-Test: Paired Two Sample for Means

	observed	model
Mean	3739.42	3742.24
Variance	15306747.17	14946533.36
Observations	12	12
Pearson Correlation	0.9881	
Hypothesized Mean Difference	0	
df	11	
t Statistic	-0.016	
P(T<=t) two-tail	1.796	
t Critical two-tail	2.201	

The model uses c_1 = 7.1, c_2 = 2E-5, and c_3 = 3E-5. The t-test shows that the t-statistic result (-0.016) is not in the rejection area (-2.201 < t < 2.201). Therefore, with 95% confidence interval, there is no significant difference between the observed value and the model.