A STUDY OF SIMULATION PERFORMANCE BASED ON EVENT ORDERINGS

HU YANJUN

NATIONAL UNIVERSITY OF SINGAPORE

2003

A STUDY OF SIMULATION PERFORMANCE BASED ON EVENT ORDERINGS

HU YANJUN

(B. Sci., Peking University, China)

A THESIS SUBMITTED

FOR THE DEGREE OF MASTER OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

NATIONAL UNIVERSITY OF SINGAPORE

2003

Abstract

A simulation protocol must adhere to a certain event ordering to produce correct simulation results. However, different event orderings exploit various degrees of parallelism and may require different amounts of memory. We have developed a formal methodology to predict the event parallelism and memory requirement of parallel simulation before implementation based on event orderings. This methodology was previously validated using limited queuing network benchmarks.

This thesis focuses on the study and validation of this methodology using a larger and more realistic application. We modeled and implemented an Ethernet network simulator and used it to study the effects of event orderings on simulation performance. The simulator is instrumented to obtain its event sequence and causal relationships, and various event orderings are analyzed using a time space analyzer that we have developed. The experimental results reveal that in a closed system, a weaker event ordering exploits more parallelism without increasing memory usage. We observed that in the Ethernet network simulator the upper bound on memory due to event orderings is $6n-8$, where n is the number of stations. Apart from assessing the cost of event orderings, the methodology can also analyze the performance of a simulation problem and the overhead of implementation. To study the cost of implementation, we analyzed the conservative null message simulation protocol and observed that much more memory is required to support synchronization than for maintaining event orderings.

Acknowledgement

First, I would like to express my heartfelt thanks to my supervisor, Associate Professor TEO Yong Meng, for his supervision through this project. He has conscientiously provided me with careful guidance at every stage of my research, offered various ideas whenever I ran into difficulties, and constructively corrected some of my mistakes in the course of my work. I appreciate the fact that participating in his projects has granted me many paths to develop my research and analytical abilities greatly. His support enabled me to both learn and write what is presented in this thesis. In addition, he has given me constructive suggestions on my attitude to work, which is helpful to my career development.

Another person who has made many contributions to this thesis is Bhakti Stephen Onggo, a PhD student who is currently working on time and space analysis of parallel simulation. Every time when I had some problems in my research he would kindly offer his help to me. He explained certain difficult concepts and definitions concisely to me.

I would also say thanks to Dr. Gary S. H. TAN for he inspired my research interest in parallel and distributed systems in course CS5223 – distributed system. Hand-on experience gained in the course project gave me an advantage in this study.

Others that I would like to thank include Ng Yew Kwong, Hu Yu, Gozali Johan Prawira and Dr. Li Ming, whom I enjoyed sharing discussions on parallel simulation and programming questions with.

In addition my sincere appreciation is given to my lab fellows, Ameya Virkar, Zhao Na, Zhang Gong, Liu Ming and Liu Peng for their generous help both in my research and in my life, and for the pleasant and friendly environment of the computer system lab.

Last but not least, I would like to convey my gratitude to the thesis examiners for taking time from their busy schedules to assess my research work.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Two major methods are used to understand real world problems and applications: mathematics and simulation. Mathematics is a highly abstract method. It is general but lacks the detailed information of the real world applications. On the other hand, computer simulation is more application specific and can provide more detailed information that aids in the understanding of the behavior of the real world systems. Researchers in several areas like engineering, computer science, economics, and military applications are particularly interested in using simulation to study the potential behavior of some of their complex models prior to implementation [11].

Parallel simulation emerged with the development of parallel computer systems. However, parallel simulations introduce much complexity in the management of event synchronization and additional programming effort is required to exploit parallelism efficiently. Many synchronization protocols have been proposed to speedup parallel simulations but they incorporate different degrees of complexity [19, 27]. Synchronization protocols may need additional working memory to maintain event causality during execution. Memory management in parallel simulation is also a main research interest [16, 21, 38].

This chapter is organized as follows. We first introduce parallel discrete-event simulation (PDES). Next, we survey the related works on performance analysis of parallel simulation. We finally present our performance study methodology based on event ordering.

1.1 Parallel Discrete-Event Simulation

PDES refers to the execution of a single discrete-event simulation program on a parallel computer [13]. In the past two decades, PDES has attracted a considerable amount of interest in the research community. This trend stems from the rapid development in parallel processing in the period, along with the fact that simulations involving large problem sizes and granularity often have poor performance when they are run on sequential machines. It represents a kind of problem that contains substantial amounts of parallelism but is very difficult to parallelize in practice.

The use of logical processes (LP) [22] and virtual time [16] has separated PDES from other simulation categories. Most existing PDES implementation mechanisms use a process-oriented methodology that strictly forbids processes to directly access the shared state variables. Sequencing constraints must be maintained by these strategies. The physical system is viewed as being composed of some number of physical processes that interact at various points in simulated time. Hence the simulator is organized as a set of LPs. One or more LPs can be mapped to a physical processor. All interactions between physical processes are modeled by time stamped event messages sent between the corresponding logical processes. Each logical process contains a portion of the state corresponding to the physical process it models, as well as a local clock that denotes how far the process has processed. The logical process methodology requires application programmers to partition the simulator's state variables into a set of disjoint states, and ensure that no simulator event directly accesses more than one state.

Simulation systems are divided into two categories in PDES: *synchronous* and *asynchronous*. In synchronous systems events are synchronized by a global clock. One iteratively determines which events are safe to process, and then processes them. Barrier synchronizations are used to keep iterations (or components of a single iteration) from interfering with each other. Because barrier synchronizations are necessary, these algorithms are best suited for shared memory machines in order to keep the associated overheads to a minimum [11]. However, in asynchronous systems events occur at irregular time intervals. Asynchronous LP simulation relies on the presence of events occurring at different simulated times that do not affect one another. Concurrent processing of those events thus effectively accelerates sequential simulation execution time.

PDES mechanisms generally fall into two categories of synchronization protocols: *conservative* and *optimistic*. Conservative mechanism executes only safe events. An LP blocks when no safe events can be executed. The typical conservative protocol is CMB null message protocol [5]. The obvious drawback of conservative approaches is that they cannot fully exploit the parallelism available in the simulation problem. From the programmer's point of view, the most serious drawback of existing conservative simulation protocols is that the simulation programmer must be concerned with the details of the synchronization mechanism in order to achieve good performance. On the other hand, an optimistic mechanism allows an unsafe event to be executed. An error-detection mechanism is required to determine when an error has occurred, and then it will invoke a procedure to recover. One advantage of optimistic approach is that it can exploit parallelism in situations where causality errors may occur but actually do not. The typical protocol of optimistic mechanism is Time Warp [15]. Because optimistic mechanisms need to save system states frequently, they generally consume much more memory than conservative protocols.

Although PDES remains an active area of research, it has not achieved industrial widespread use [12]. There are several reasons for this fact. Firstly, the positive results will easily find their ways to publication, so we tend to see a biased picture. Secondly, the gained speedup is always attractive, but the effort spent on programming is also quite substantial. Finally the positive results usually can only be achieved by experts in certain fields.

1.2 Related Works

Much effort has been exploited to analyze the ***parallelism*** of a simulator either before or after parallel implementation [19, 27]. Performance analysis methods generally fall into the following three categories: *analytic method*, *simulation-based method* and *critical path method*.

Analytic methods usually use stochastic process, queuing theory or operational laws. Some kinds of Markov chains underlie these analyses [25]. Felderman and Kleinrock show that the average performance difference between synchronous and asynchronous algorithm is less than $O(\log P)$ [8]. Tay et al. presents an analytical model for evaluating the performance of Time Warp simulators [35]. Wang et al. propose an analytical method to predict the parallelism of a simulation where causal relationship among events is considered [43]. In general, the analytic methods are faster than other methods, but the drawback is that it usually has unrealistic assumptions.

The second performance analysis method is based on simulation, which analyzes performance by directly simulating particular PDES protocols. Dickens and Reynolds develop a model to study the performance of a system synchronized by a windowing protocol [7]. The model extends the windowing protocol to allow computation of conditional events and predicts the probability of a causal error. Lim et al. describe three parallelism prediction tools for different synchronization protocols [19]. However, the tools can only be applied to some conservative protocols. Cavitt et al. propose a framework for identifying the factors affecting the performance of simulation [4]. The identified factors can in turn give feedbacks to simulation hardware/software configuration. Marin et al. devise a simple automated methodology to predict running time cost of discrete-event simulation [23]. However, the methodology can only be used for BSP (Bulk-Synchronous Parallel) model. Teo et al. concentrate on the performance analysis on a particular simulation library SPaDES/C++ [36]. Rawling et al. analyze an existing sequential simulation in order to predict concurrency speedup bounds for conservative parallel simulation [29]. The model is based on real commercial VLSI simulations. Noble et al. explore the performance of three synchronous discrete-event simulation algorithms: global clock algorithm, conservative look-ahead algorithms and speculative computation algorithm [26]. De Carvalho Klingelfus et al. developed an object oriented Ethernet network simulation and model system to aid in the activity of element measurements, error detection and performance analysis [6]. In summary, simulation-based method usually uses one particular protocol or one particular category of protocols to model applications. They require fewer assumptions than analytical method, but the method has only limited usage, for there are so many protocols and applications to be simulated.

Critical path analysis simulates event execution based on causal relationship and builds critical path to analyze the simulation performance. Wong et al. proposes a critical path-like analyzer to predict the memory used in a Chandy-Misra simulation [48]. The analyzer can derive the parallelism directly from a path-like analyzer. Lim et al. use a critical path analyzer to give the ideal maximum speedup for a simulation model [19]. The critical path analysis assumes each physical processor to be an independent LP and there is unlimited number of processors. Wieland et al. use a new technique to determine the critical path [46]. A metric called the earliest process time (EPT) can be implemented either as a centralized algorithm or a distributed algorithm. Critical path analysis is easy to understand but it cannot be used to compare different protocols.

Fujimoto states that the performance of conservative strategies is closely related with the degree to which processes can look ahead and predict future events [11]. For optimal protocols, state-saving overhead can seriously degrade performance. In addition, optimistic algorithms usually use more memory than conservative ones.

Parallel simulation provides the potential to speedup simulations, but additional memory is required by the parallel synchronization protocols. Specifically, for conservative protocols, the additional memory is required to hold the null messages. Optimistic protocols require additional memory to save the simulation states periodically for possible rollback. Every processor in parallel simulation has only limited space, so memory consumption is also an important issue that we should address.

There are many publications on the *space* aspect of parallel simulation [16, 21, 38]. But most publications concentrate only on the space management of some particular synchronization protocols. For conservative approaches, much effort is done to reduce the number of null messages, such as demand-driven null message algorithm presented in [1]. For optimistic approaches, the focus is on reducing optimism while limiting the usage of space. The “artificial rollback” in [21] is such an example. Many researchers examine the storage utilization of optimistic mechanisms such as Time Warp. To support rollback, it is necessary to save the old states of a logical process but there is no need to save the “ancient history” [13]. Hence these memories can be reutilized to save new state vectors. Several approaches have been proposed to limit the amount of memory that is required to perform the simulation in Time Warp.

The first one is fossil collection and global virtual time (GVT) [50]. The smallest timestamp among all unprocessed event messages is called GVT. No event with timestamp smaller than GVT will ever be rolled back, so storage used by such events can be discarded. In addition, irrevocable operations (I/O for example) cannot be committed until GVT passes the simulated time at which the operation occurs.

The second approach is incremental and infrequent state savings. In conjunction with fossil collection, there are many other mechanisms to save more memory. When the state vector is large and only a part of it is modified by each event, incremental state saving may be useful. Only changes to the state are recorded to reduce both memory utilization and copying time. A drawback of this mechanism is that the rollbacks become more expensive. An alternative approach is to save entire state vectors, but reduce the frequency of state saving [20]. It decreases the time required to perform state saving, but increases rollback overhead. This tradeoff suggests that there may be an optimal state saving frequency that balances state saving overhead and re-computation costs [28].

The next method is rollback-based recovery mechanisms. With the aforementioned mechanisms, when the system does run out of memory, there is no recourse but to terminate the simulation. It is problematic because the "fault" may lay with the Time Warp mechanism itself rather than the application program. Several approaches have been developed to address this concern. Such mechanisms include cancel-back [16] and artificial rollback [21] algorithm.

The last method is to limit memory by using the protocols with limited optimism. If the simulation mechanism is too optimistic in executing the program, then the program, as a result, will run out of memory. There are emerging approaches that use limiting optimistic protocols [49].

1.3 Event Ordering Based Approach

Simulation protocols maintain a certain event ordering to produce correct simulation results. Ordering of concurrent events in discrete-event simulation is an important issue as it has an impact on modeling expressiveness, model correctness and causal dependencies [32]. In sequential simulation, only one event ordering is maintained by global FEL. In parallel simulation, every LP maintains its own FEL and many events can be executed simultaneously. Synchronization protocols order the events in an appropriate manner to guarantee that no causality errors occur. Different event orderings are allowed to generate correct simulation results, but they give different degrees of parallelisms. In addition, each LP needs extra memory to keep track of pending events in its future event list (FEL) to follow a certain event ordering. Therefore different event orderings may require different amounts of memory.

Teo et al. have developed a formal methodology to study how event ordering influences the performance of parallel simulation [40]. The methodology can predict the performance of parallel simulation before it is actually implemented. It executes events based on causal relationship and event ordering to analyze event parallelism and memory requirement of a simulator. It can compare the performance between different event orderings, which is more general than simulation-based methods. Because event orderings, not synchronization protocols, are taken into account, the methodology requires less implementation than simulation-based performance analysis methods.

Four simulation event ordering rules are formally defined with partial order set theory: *total event ordering*, *timestamp event ordering*, *time interval event ordering* and *partial event ordering* (Axiom 1 to Axiom 4).

AXIOM 1: *Let $\langle E, <_{par}\rangle$ be a poset, where E is a set of events. Under partial event ordering, e_1 happens before e_2 (denoted by $e_1 <_{par} e_2$), if:*

- *$\neg(e <_{par} e)$, for any event $e \in E$;*
- *e_1 and e_2 are events in the same process, and e_1 comes before e_2;*

- *e_1 is the sending event in process P_1, and e_2 is the corresponding receiving event in process P_1;*
- *if $e_1 <_{par} e_2$ and $e_2 <_{par} e_3$, then $e_2 <_{par} e_3$.*

***AXIOM 2**: Let $\langle E, <_{par}\rangle$ be a poset, where E is a set of events. Assume that each $e \in E$ can be stamped with a simulation time (denoted by ts(e)). Under total event ordering, e_1 happens before e_2 (denoted by $e_1 <_{tot} e_2$), if:*

- *$ts(e_1) < ts(e_2)$, or*
- *$ts(e_1) = ts(e_2) \wedge e_1$ has higher priority than e_2.*

***AXIOM 3**: Let $\langle E, <_{par}\rangle$ be a poset, where E is a set of events. Assume that each $e \in E$ can be stamped with a simulation time (denoted by ts(e)). Under timestamp event ordering, e_1 happens before e_2 (denoted by $e_1 <_{ts} e_2$), iff $ts(e_1) < ts(e_2)$.*

***AXIOM 4**: Let $\langle E, <_{par}\rangle$ be a poset, where E is a set of events. Suppose that the simulation duration can be divided into mutually exclusive time windows, $\{W_1, W_2, \ldots, W_n\}$, where $W_i = W_j$ iff $i=j$. Assume that each $e \in E$ can be placed in a W_i with base time denoted by tw(e). Under time interval event ordering, e_1 happens before e_2 (denoted by $e_1 <_{ti} e_2$), iff $tw(e_1) < tw(e_2)$.*

The definitions in Axiom 1 and Axiom 4 are consistent with those by Lamport in [17] where "happened before" relation is the same as partial event ordering. For Axiom 1, e1 happens before e2 because the sending event will causally affect e2. Partial event ordering is anti-symmetric, so if that e1 is a receiving event and e2 is the corresponding sending event, e2 will happen before e1 [40].

The event orderings in decreasing order of *strictness* are *total event order*, *timestamp event order*, *time-interval event order* and *partial event order*. The detailed definition of event orderings and proof of their strictness are illustrated in [40]. The main difference among these four event orderings lies in the definition of concurrent event. The methodology can be applied to all event orderings as long as they are well defined.

The methodology is based on the typical steps of a simulation. A computer simulation is a program that emulates the behavior of another system. A typical modeling and simulation process contains three steps: *physical system*, *simulation model* and *implementation model* as shown in Figure 1.1. Physical system represents the real-world problem that one simulates. A simulation model is a logical model of a physical system that defines the input parameters, output results, and other physical system components to be simulated. There are three world views in simulation model: *event oriented*, *process oriented* and *activity scanning* [13]. The physical system and simulation model is independent of the implementation. Either sequential or parallel implementation needs to be built on the simulation model.

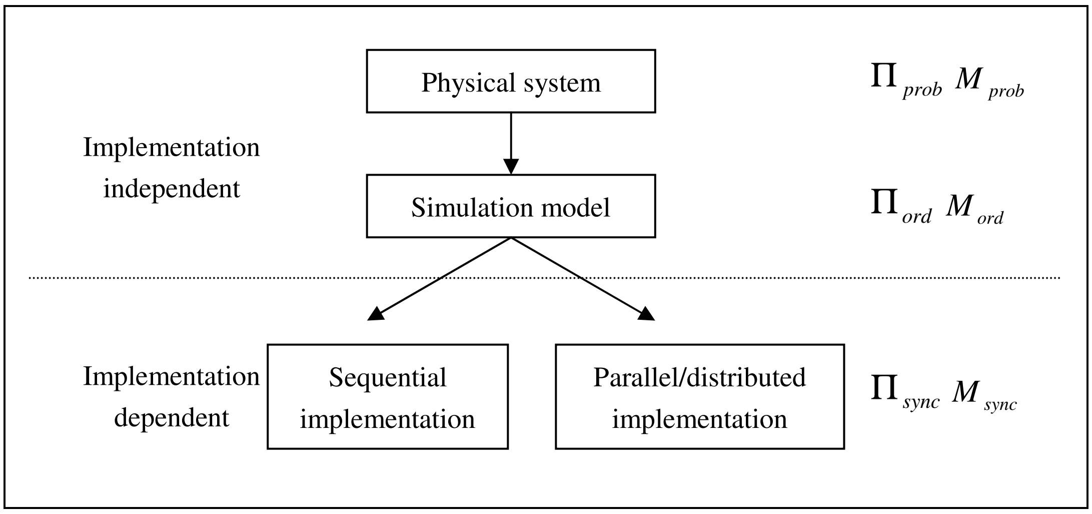


Figure 1.1: A typical simulation process

We divide the memory required by a simulator into three main parts: M_{prob}, M_{ord} and M_{sync}. M_{prob} denotes the memory to model the states of the physical system, M_{ord} denotes the memory required by future event list (FEL) to schedule event execution based on the selected event ordering, and M_{sync} denotes the additional amounts of memory to implement a synchronization protocol on a specific execution platform. Therefore, the total memory requirement of implementing a simulation model on real machines with a particular implementation is $M_{prob}+M_{ord}+M_{sync}$ [40].

We measure M_{prob} by observing the queue size, and its upper bound is defined as the total maximum queue length, i.e., $M_{prob} \le \sum_{i=1}^{n} Q_i$, where Q_i is the maximum queue size at service center i, and n is the number of service centers. For simplicity, we only count the entry number for the queue. The actual M_{prob} is dependent on the data structure of queue implementation.

M_{ord} depends on the characteristics of system under study, i.e., event arrival and service rates, and the event ordering adopted. The upper bound of M_{ord} is defined as the sum of all FEL lengths, i.e., $M_{ord} \le \sum_{i=1}^{n} FEL_i$, where FEL_i is the maximum FEL size at service center i, and n is the number of service centers. The actual value of M_{ord} is dependent on the implementation of FEL.

M_{sync} accounts for the additional memory used for synchronization. For sequential implementation, $M_{sync} = 0$. In optimistic protocol, memory is required for state saving in anticipation of rollbacks. In the case of the null message protocol, it can be defined as the total of the maximum buffer sizes required for maintaining null messages. Therefore, for the conservative null message parallel simulation used in SPaDES/Java [41], we can define $M_{sync} \le \sum_{i=1}^{n} NMB_i$, where NMB_i is the maximum null message buffer size at LP_i, and *n* is the total number of LPs involved in the simulation.

Event parallelism is defined as the average number of events executed per unit time. Average event parallelism (Π) is different from speedup here. The range of Π is $[1, \infty]$. All events are assumed to take the same execution time and we need to specify what one unit time is.

For sequential simulation the average event parallelism is one. However, different types of events may take different execution time. When a sequential simulation is mapped to the parallel environment, events can be executed simultaneously at different processors. The number of events per unit time will increase, thus the parallelism will be larger than one for parallel simulation. However, parallel simulation needs additional overhead for synchronization, such as null message, which will decrease the parallelism.

Similar to the memory classification, the average event parallelism of a simulator is also studied at three steps, namely: physical system, event ordering, and implementation [27]. In the physical system level, events may happen concurrently. Hence, physical system has parallelism which is called the *inherent event parallelism* (Π_{prob}). Discrete-event simulation compresses simulation time by applying a certain event ordering. Different event orderings exploit different degrees of event parallelism which is called *event ordering parallelism* (Π_{ord}). The communication overhead and other implementation overhead are neglected, so event ordering parallelism is optimal. At the implementation level, maintaining a certain event ordering on a specific platform requires addition overhead of synchronization. We refer this parallelism as the effective event parallelism (Π_{sync}).

Inherent event parallelism (Π_{prob}) refers to the parallelism that exists in the physical system. It is mainly determined by physical system factors, the traffic intensity for example. In a physical system some service centers can execute events concurrently. The dependency between events influences the inherent event parallelism. Less dependency between events gives higher parallelism. The topology between service centers can influence the inherent event parallelism because it will influence the dependency between events [27]. Π_{prob} is measured from an analytical method and it is defined as the sum of all LPs' utilization, i.e., $\Pi_{prob} = \sum_{i=1}^{n} U_i$, where U_i denotes the utilization of LP_i. Teo et al. has proved the measurement from a common measure of program parallelism [37]

Different event ordering exploits different degrees of event parallelism. This parallelism is referred to as Π_{ord}. As mentioned before, four simulation event orderings are defined in the methodology representing four different degrees of parallelism, i.e. *total event ordering*, *timestamp event ordering*, *time-interval event ordering* and *partial event orderings*. This work can be extended to include other event orderings. Both causal restriction and event ordering rules are considered in the measurement of Π_{ord}. The detailed measurement of Π_{ord} is presented in Chapter 2 when we present the implementation of the methodology.

At the implementation level, maintaining a certain event ordering on a specific execution platform requires synchronization overhead, hence the implementation may reduce Π_{ord}. We call this parallelism the effective event parallelism Π_{sync}. Both of the implementation algorithm and execution platform (processor, network, operating system, etc) may affect Π_{sync}. Π_{sync} is measured from the actual simulation and the detailed measurement is presented in Chapter 2.

It is known that the total communication time or cost is dependent on the interconnection topology of processors (LPs) used in parallel simulation. The effects of interconnection topology of a physical system on exploitable event ordering parallelism are studied at [27]. Four synthetic benchmarks representing basic queuing network topologies are implemented and studied: Linear Pipeline, Pipeline with Feedback, Circular Pipeline and PHOLD. It is found that feedback channel reduce Π_{ord} that can be exploited by relaxing the event ordering, i.e. the physical system limits the amount of Π_{ord} exploitable by parallel simulation.

The degree of event parallelism is related to the granularity that the number and size of events or tasks into which a problem is decomposed. The formal methodology studies the performance (event parallelism and memory requirement) from three levels. At the event ordering level, we study the performance of parallel simulation with different event orderings. Each event is assumed to take one unit time to execute. Π_{ord} is independent of the implementation. At the implementation level, the granularity is considered and we normalize the event execution time to the average execution time as presented in section 2.1.

1.4 Research Contribution

It is essential to understand the degree of event parallelism *before* substantial programming effort is invested to develop a simulator [36]. If there is low degree of parallelism in the system, the performance benefits of exploiting parallelism will be low. In addition, every processor in a parallel system has only limited space capacity. Therefore, it is also important to predict the memory consumption of a parallel simulation before implementation. Teo. et al develop a performance analysis framework – time space analyzer (TSA) tool which implements performance analysis based on event orderings [40, 42]. The methodology has previously been validated with several limiting queuing network benchmarks such as LPIPE and PHOLD.

In this thesis, we use a realistic application, Ethernet network, to further study and validate the methodology. Our performance results are consistent with the existing results [27, 40], i.e., a weak event ordering gives higher parallelism without increasing memory usage in a closed system. Apart from assessing the cost of event orderings, the methodology can also analyze the simulation performance of a simulation problem and the overhead of implementation. To study the cost of implementation, we analyzed the conservative null message simulation protocol and observed that much more memory is required in implementation than for maintaining event orderings. The relationship among performance results of these levels is also discussed in this thesis.

1.5 Thesis Overview

The rest of this thesis is organized as follows:

Chapter 2 introduces our overall research methodology. We introduce the implementation and validation tools used in our research, including CSIM, SPaDES/Java, and TSA. We also validate TSA in detail with a simple Pipeline example.

Chapter 3 introduces Ethernet network modeling and its implementation. Ethernet network is introduced through three steps: physical system, conceptual model and implementation. We specify the processes and resources in Ethernet network simulator at the conceptual model. At implementation level, both the sequential and parallel simulator are implemented and validated. Lastly, we instrumented Ethernet network simulator to obtain event sequence, which will be analyzed by TSA.

Chapter 4 illustrates the experimental results and analysis. Both time (event parallelism) and space (memory requirement) are characterized at three levels: physical system, event ordering and implementation. We also compare and discuss the relationship among three levels. Next, the performance tradeoff is analyzed.

Chapter 5 concludes the thesis and discusses future work.

Chapter 2

Methodology

We discuss our research methodology in this chapter. An analytical method is used to analyze the inherent event parallelism. TSA is used to analyze event parallelism and memory requirement for different event orderings. We modeled and implemented the Ethernet network simulator using the SPaDES/Java simulation library and studied its performance. The implementation and validation tools used include CSIM, SPaDES/Java, and TSA. TSA is validated in detail using a simple Pipeline example.

2.1 Research Methodology

Figure 2.1 illustrates our overall research approach. The performance results contain three steps. In step 1, we use an analytical model to obtain the inherent event parallelism of a problem. In step 2, we use TSA to derive event parallelism and memory requirement for different event orderings. Step 3 measures effective event parallelism and memory for synchronization from the actual simulation.

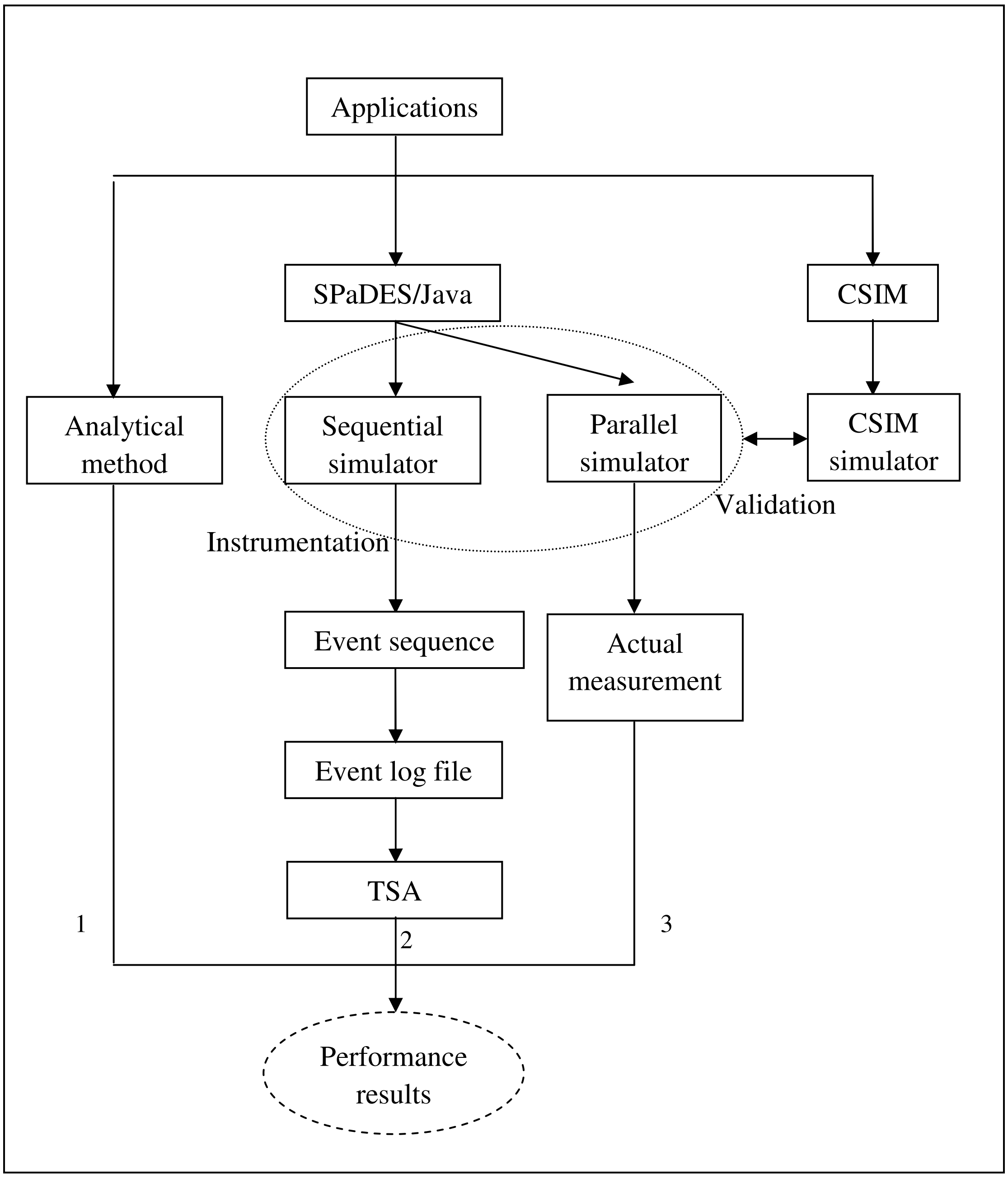


Figure 2.1: Research approach

As presented in Chapter 1, the inherent event parallelism (Π_{prob}) is measured in terms of the sum of processors' utilization. For an open system, the utilization of a service center is defined as λ/μ, where λ is its arrival rate and μ is its service rate. The utilization of an LP can also be calculated by the ratio of its mean *service* *time* to mean *inter-arrival time*.

For a closed system, such as Ethernet network, we can apply mean value analysis (MVA) [14] to analyze the queuing characteristics of the problem. MVA uses a number of fundamental queuing relationships to determine the mean values of throughput, delay and queue size for closed queuing networks. Unlike the service centers with finite service units, Ethernet network nodes are delay servers (centers) where

- Infinite servers/dedicated servers queues on a service center;
- There is no waiting time but only service time for a service center.

Hence the mean response time is equal to mean service time for a delay center.

$$Q_i = X_i R_i = X_i S_i = U_i$$

where Q_i is the average queue size, X_i is the throughput, R_i is the mean response time, S_i is the mean service time and U_i is the utilization of an LP. We can observe that the utilization of a delay center is the mean number of jobs receiving service. Therefore, the utilization of a delay center is equal to its average service rate.

In step 2, a sequential Ethernet network simulator is developed using the SPaDES/Java simulation library. We obtain the event sequence and causal relationships from instrumentation of sequential Ethernet network simulator. All events are then recorded in an event log file. Every event is recorded with its detailed information, such as event type, timestamp, the location, etc. The event sequence are analyzed by TSA to derive event ordering parallelisms (Π_{ord}) and memory requirement (M_{ord}).

The CSIM simulation library is used to validate our model and implementation. If Ethernet network simulation is developed correctly using SPaDES/Java, it will produce the same simulation results as the one developed by CSIM [45].

In step 3, we implement the parallel Ethernet network simulator using the SPaDES/Java simulation library. A conservative null message simulation protocol is used to synchronize the parallel execution on different LPs. We measure the actual execution of the parallel simulator to obtain the memory requirement for synchronization (M_{sync}), which is measured as the sum of maximum null message buffer sizes in all LPs.

The effective event parallelism (Π_{sync}) is also measured from the actual simulation. When event ordering parallelism is measured, we assume an event is executed in one unit time and thus all events take the same execution time. However, in actual simulation different types of events may have various execution times. With reference to this, we measure the unit time (T_{unit}) as the average event execution time.

The execution time of an LP in a parallel SPaDES/Java simulation includes the following several parts:

- Time used to execute event messages;
- Time used to execute null messages;
- Communication delay, time used to wait for messages from other LPs;
- Other delays.

The null messages and other execution delays are incurred due to the additional implementation overhead. Quite a lot of factors may affect the event parallelism at implementation level. To simplify the measurement, we consider only the execution of events and null messages when measuring the execution time of an LP. From the definition of event parallelism, average number of executed events per unit time, we get the measurement of the effective event parallelism as follows:

$$\Pi_{sync} = \frac{\#Events}{\left(T \big/ T_{unit} \right)} \qquad \text{(Eq. 2.1)}$$

where $\#Events$ is the number of all events in the problem and T is the execution time (events and additional null messages) of the LP which has the longest execution time compared to the others.

2.2 Tools

The following tools are used in this study. The Ethernet network simulation is written in SPaDES/Java and it is validated by CSIM. TSA analyzes the simulation performance for different event orderings.

2.2.1 CSIM

CSIM is a simulation library in C language by Watkins [45] and it supports three simulation worldviews, namely, *event scheduling*, *three phase event scheduling* and *process interaction*. Three phase approach is different from event scheduling approach by specifying the conditional events and scanning them in a new phase. C has two major advantages over many other languages: *portability* and *availability*.

Objects in the real system are modeled in terms of entities and resources in CSIM simulation library. *Entities* present active objects in the system such as customers or processors. Entities have a close affiliation with events because they are active. Entities are usually involved in several activities. A *resource* in the real-world systems is usually some form of reusable asset such as the amount of free storage in a computer system or a checkout in a supermarket. The principal characteristic of a resource is that it has only limited capacity.

A simulator developed by CSIM usually has a better performance in comparison with one by other simulation libraries. This is due to the higher efficiency of the C language. However, CSIM does not support parallel simulation, so we cannot measure effective event parallelism (Π_{sync}) and memory for synchronization (M_{sync}).

2.2.2 SPaDES/Java

SPaDES/Java (Structured Parallel Discrete-Event Simulation in Java) is an object-oriented modeling toolkit for general-purpose simulations [41]. The synchronization processes and mechanism are hidden from the simulationists. It supports both sequential and parallel simulation. Parallel event synchronization is facilitated through a hybrid carrier-null, demand-driven flushing conservative null message mechanism.

The SPaDES system adopts the approach of augmenting a general-purpose language with essential constructs to support simulation modeling based on the process-oriented modeling technology. The simulation programmer can concentrate on modeling and be lifted from the burden of programming the complicated event synchronization protocol and message passing mechanism.

SPaDES adopts a modified process-interaction modeling view called *process-oriented* modeling view. In this view, entities in the real world are viewed as a set of processes each encapsulating its own state and behaviors, and processes interact with one another through message passing. Furthermore, it is necessary for a process-oriented model to be mapped to an operational model that is suitable for parallelization. The operational model of SPaDES is based on the virtual time paradigm [16].

In the process-oriented view, real-world entities are categorized into *permanent* and *temporary* entities. A permanent entity, modeled as a *resource*, exists throughout the simulation duration. A temporary entity, modeled as a *process*, is a process that can be created dynamically at any point during the simulation and thus does not exist throughout the simulation duration. A *process* can be in some states during its entire simulation lifetime. In the operational model, resources are modeled as LPs and processes are modeled as time-stamped event messages passed between LPs. SPaDES/Java adopts RMI library to facilitate the message passing between processors.

Resources are the permanent simulation entities present to provide services to the active processes upon request. Each resource comprises of a default FIFO queue, created when the resource is constructed, and whose function is to maintain the arrival of processes to the resource according to their timestamp values, followed by event priority. Each resource is really a collection set of *service units*, which is the basic functional unit of a resource. When an active process requests for service at any particular resource, the total number of service units required must be explicitly mentioned. SPaDES/Java implements all the event lists using *binary min-heaps*. The time complexity for inserting and removing a message is *O(log n).*

Using Java as the base language, SPaDES/Java is portable across all platforms. It can support parallel simulation, so we can measure Π_{sync} and M_{sync}. SPaDES/Java is object-oriented, which facilitates the program development and maintenance. However, one drawback of SPaDES/Java is that its performance is worse than CSIM. Java's platform independence requires a Java virtual machine to run on the local machine, thus sacrificing some degree of its performance. However, in comparison with CSIM, SPaDES/Java is a better choice for our implementation.

2.2.3 TSA

Teo et al. originally implement TSA in C [40]. It can be instrumented to a simulator by CSIM to derive the performance results. TSA measures M_{prob}, M_{ord}, and Π_{ord} by analyzing the event sequence provided by a simulator. Java version TSA is the translation of original C version TSA [42]. Java version TSA can be instrumented to a simulator in SPaDES/Java.

TSA is designed to measure the performance results for different event orderings. The input of TSA is an event sequence with its causal relationships from simulator. Every event is recorded with its event type, its location and timestamp. Event sequence is stored in a doubly-linked list. Because event sequence is obtained from a sequential simulator, the events are automatically sorted by their timestamp.

Events are fetched into TSA in the order they are executed in a sequential simulator. TSA executes these events in parallel by following a particular event ordering. Each event is assumed to execute in one unit time. Figure 2.2 shows the main loop in a sequential simulator with TSA instrumentation. The simulator invokes TSA for each event that is removed from the future event list. Typically a simulator advances its virtual time to the event's timestamp. We record the event information (line 5) and then write the event to a log file or schedule it to the TSA routine (line 6).

```
1. While <<simulation is running>>{
2.     <<remove top event e from future event list>>;
3.     execute(e);
4.     simulation_clock=timestamp of e;
5.     TSA_record(e);
6.     TSA_schedule(e);/event_log(e);
7. }
```

Figure 2.2: Simulation executive main loop with TSA instrumentation

TSA has two options to analyze event sequence: (a) It executes in parallel with the simulation with one event scheduled to TSA immediately when the simulation executes it; (b) TSA executes after the simulation by fetching events from a log file and works as a post-execution instrumentation analyzer. We adopted the latter option. For a large simulation that has a large execution time, we run the simulation once. The event log file is used by TSA many times without rerunning the simulation. Another benefit is that the event log file can be used to validate the instrumentation.

When TSA is initialized, it sets up four instrumentation classes representing four simulation event orderings. Each class maintains two arrays: *maxFEL* and *maxCEL*, with *n* slots, where *n* is the problem size. These two arrays keep track of the maximum lengths of the FEL and CEL of each LP throughout the simulation. Each class also records the critical path length. When the TSA-instrumented simulator is running and a new customer arrives in the system, or an event has been scheduled in a particular LP, a new *event* is created to record this change in state of the simulator. When all LPs have at least one event in its event list, TSA advances its time by one time unit, i.e., increasing the critical path by one, and execute the top events according to the defined event ordering rule.

After all events are analyzed, TSA computes M_{prob}, M_{ord} and Π_{ord} as defined below:

$$M_{prob} = \sum_{i=1}^{n} \max CEL_i$$

$$M_{ord} = \sum_{i=1}^{n} \max FEL_i$$

$$\Pi_{ord} = \frac{\#events}{critical\_path\_length_{ord}}$$

where $\#events$ is the number of all events and $critical\_path\_length_{ord}$ is the critical path length of a particular event ordering.

2.3 TSA Validation

We validated TSA before it is used to analyze performance results. A simple linear Pipeline (PL) example is used to validate TSA. PL is manually analyzed and the corresponding event sequence is stored into an event log file. The causal relationships between events and other event information are recorded in the file. The file is small enough for us to analyze event sequence and manually derive the performance results. Then the results are compared with the results generated by TSA. Figure 2.3 illustrates our validation methodology.

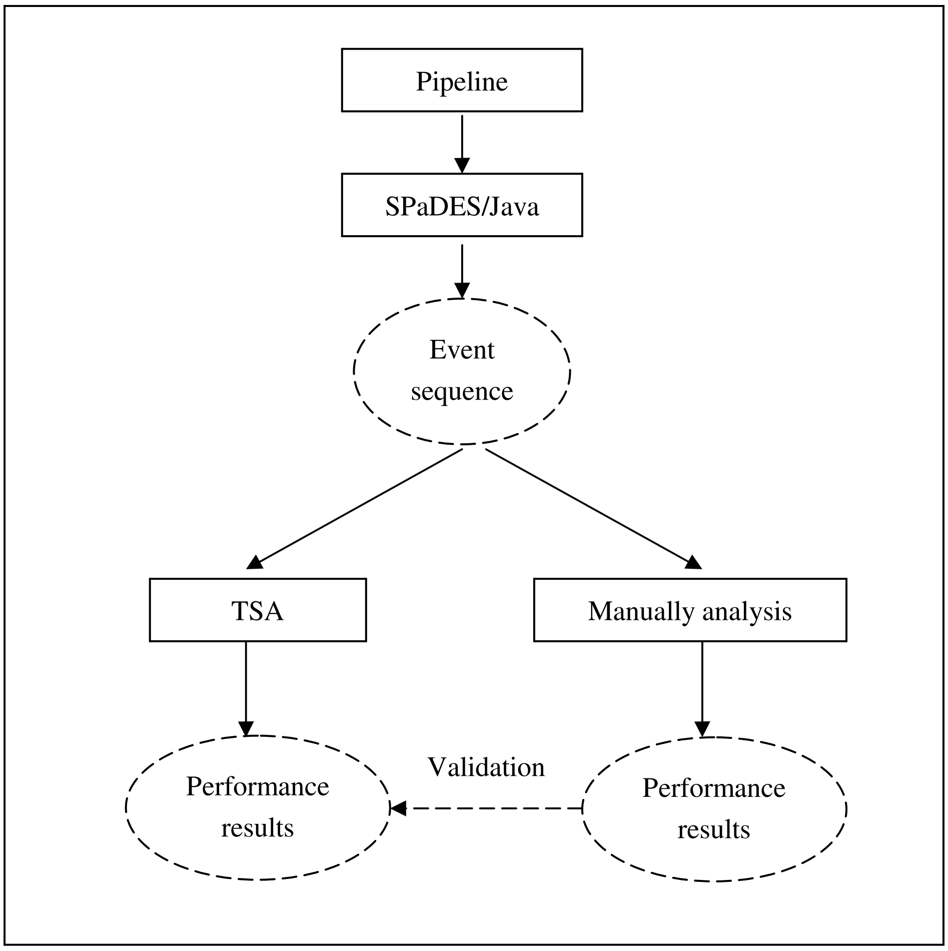


Figure 2.3: TSA validation methodology

A 2-LP Pipeline example is used as our validation program. A laundry with wash point (LP_0) and dry point (LP_1) is such an example. There are four event types in linear Pipeline: external arrival, internal arrival, external departure and internal departure. Four messages are modeled to flow through the pipeline. The inter-arrival-time is fixed to 7 time units and the service time in one LP is set to 8 time units. $Message_0$ is scheduled to enter LP_0 at simulation time 1. The runtime information is listed in Appendix A, which is a “trace” of a simulation.

Message ID	External Arrival at LP_0	Internal Departure from LP_0	Internal Arrival at LP_1	External Departure from LP_1
$message_0$	$e_1 : 1$	$e_3 : 9$	$e_4 : 9$	$e_6 : 17$
$message_1$	$e_2 : 8$	$e_7 : 17$	$e_8 : 17$	$e_{10} : 25$
$message_2$	$e_5 : 15$	$e_{11} : 25$	e_{12} 25	$e_{15} : 33$
$message_3$	$e_9 : 22$	$e_{13} : 33$	$e_{14} : 33$	$e_{16} : 41$

Table 2.1: Event sequence with timestamps in Pipeline simulation

Table 2.1 lists all the sixteen events with time stamps generated from PL simulation. Every event is recorded with its message ID, timestamp, location, Event type, next location and the antecedent event information as required by TSA. Appendix B lists the detailed event information for all sixteen events in the log file.

According to aforementioned measurement, M_{prob} is the sum of maximum CEL lengths (*maxCEL*) of all LPs, M_{ord} is the sum of maximum FEL lengths (*maxFEL*) of all LPs and Π_{ord} is the ration of the number of events to the critical path length of a particular event ordering. The results of M_{prob}, M_{ord} and Π_{ord} for PL are listed at Table 2.2.

Event ordering	M_{prob}	M_{ord}	Critical path length	Π_{ord}
Partial		3	10	1.60
Time interval*	2	3	11	1.45
Time stamp		3	12	1.33
Total		3	16	1.00

**window size of time interval event ordering is 2 time units*

Table 2.2: Performance results of Pipeline simulation

Because $message_1$ should wait in CEL [0] for the $message_0$ to depart from

LP_0, the *maxCEL[0]* is equal to 1. $Message_2$ should wait in CEL[1] for $message_1$ to depart from LP_1, so *maxCEL[1]* is equal to 1. Thus M_{prob} is the sum of *maxCEL[0]* and *maxCEL[1],* which is 2. When a new message arrives at LP_0, it will schedule its internal departure and next message's external arrival, so *maxFEL [0]* is equal to 2. Arrival at LP_1 can only schedule one departure event, so *maxFEL [1]* is equal to 1. Therefore, M_{ord} is the sum of *maxFEL[0]* and *maxFEL[1]*, which is 3.

Let us analyze Π_{ord} now. Table 2.3 shows the event dependency information of the 16 events, where "$e_1 \to e_2$" means that e_1 is the antecedent event of e_2.

Simulated time	LP_0	LP_1
1	$e_1 \to e_2$	
8	$e_2 \to e_5$	
9	$e_3 \to e_4$	$e_4 \to e_6$
15	$e_5 \to e_9$	
17	$e_7 \to e_8$	$e_8 \to e_{10}$
	$e_7 \to e_{11}$	
25	$e_{11} \to e_{12}$	$e_{12} \to e_{15}$
	$e_{11} \to e_{13}$	
33	$e_{13} \to e_{14}$	$e_{15} \to e_{16}$

Table 2.3: Causal relationships between events in Pipeline

Table 2.4 lists the event execution sequence of the four simulation event orderings. Whether an event can be executed in one step is determined by the causal restriction and event ordering rules. First, the critical path length of total event ordering is 16 since there are 16 events in the system. For total event ordering, only one event can be executed at one step and there is no event parallelism, Π_{ord} is 1.

Events can be executed under partial event ordering as long as they do not violate causal restriction. In timestamp event ordering, if two events can be concurrently executed at one step, they must reside in one time window. Therefore e_9 and e_{10} cannot be executed concurrently at step 6 under time interval event ordering (window size is 2 time units), even though they can be concurrently executed under partial event ordering. For timestamp event ordering, only two events with the same timestamp can be executed concurrently in one step. e_5 and e_6 cannot be executed concurrently at step 4 under timestamp event ordering because they have different timestamps, even though they can be concurrently under time interval event ordering.

Step	Partial event ordering	Time interval event ordering	Time stamp event ordering	Total event ordering
1	e_1	e_1	e_1	e_1
2	e_2	e_2	e_2	e_2
3	e_3 & e_4	e_3 & e_4	e_3 & e_4	e_3
4	e_5 & e_6	e_5 & e_6	e_5	e_4
5	e_7 & e_8	e_7 & e_8	e_7 & e_6	e_5
6	e_9 & e_{10}	e_9	e_8	e_6
7	e_{11} & e_{12}	e_{11} & e_{10}	e_9	e_7
8	e_{13} & e_{14}	e_{12}	e_{11} & e_{10}	e_8
9	e_{15}	e_{13} & e_{14}	e_{12}	e_9
10	e_{16}	e_{15}	e_{13} & e_{14}	e_{10}
11		e_{16}	e_{15}	e_{11}
12			e_{16}	e_{12}
13				e_{13}
14				e_{14}
15				e_{15}
16				e_{16}

Table 2.4: Event execution sequences for all event orderings in Pipeline

We execute the Pipeline simulation and obtain the same event sequence. TSA executes the event sequence and produces the same performance results as our manual analysis method. Therefore, TSA implementation is validated.

2.4 Summary

We illustrated our overall research methodology in this chapter. An analytical method was used to analyze the inherent event parallelism of a problem. TSA was used to analyze the performance of parallel simulation based on event orderings. The Ethernet network simulator was implemented using SPaDES/Java simulation library. CSIM simulation library was used for validation. Lastly, a simple Pipeline example was used to validate our TSA implementation in detail.

Chapter 3

Ethernet Modeling and Implementation

We discuss the modeling and implementation of Ethernet network in this chapter. Ethernet is currently the most-used LAN technology. There is no central control in Ethernet network and all stations transmit data independently, so we expect a high degree of event parallelism. Simulation plays a vital role in attempting to characterize the behavior of Network applications [9].

3.1 Problem

Ethernet is a branching broadcast communication system for carrying digital data packets among locally distributed computing stations [24]. A station can be attached anywhere to a passive coaxial cable, sometimes known as Ether, via a device called a transceiver. Ethernet network is reliable because of its distributed control - a single point failure can cause only partial interruption.

Ethernet uses CSMA/CD (Carrier Sense Multiple Access with Collision Detect) protocol. Any Ethernet station having frames to send will attempt to do so after it finds an empty cable. If two or more stations transmit frames simultaneously, there will be a collision. Each station detects the collision, abort its transmission, wait a random period of time, and then try again until its packet is successfully transmitted or the transmission is considered to fail because the maximum number of retransmission, 16, is reached. Binary exponential back-off algorithm is used to determine the time (back-off) before a retransmission. The back-off in terms of the number of slots (512 bit times) is defined by a uniformly distributed number between $[0, N]$, and

$$\begin{aligned} N &= 2^i - 1 \qquad (1 <= i <= 10) \\ &= 1023 \qquad (10 < i <= 16) \end{aligned}$$

where i is the number of transmissions made so far.

The Ethernet network exists to move frames carrying application data between computers. Hence the structure of a frame is central to the operation of the system. Preceded by a start bit, a frame starts with the destination and source address, which are both 6 bytes long. Following the address are 46 to 1,500 bytes of data. The last 4 bytes of a frame are the Frame Check Sequence or CRC. Only the address and data part are accessible to software. The minimum frame size is 64 bytes and the maximum frame size is 1518 bytes.

Simulation methods have been used to analyze the performance of Ethernet. Watkins uses a simulation method to study Ethernet protocol efficiency in the steady state [45]. He observes that an Ethernet network with larger frames has higher protocol efficiency than one with smaller frames.

3.2 Simulation Model

The input parameters, output, system states and components (processes and resources) of Ethernet network are specified in our simulation model. The input parameters are:

- *Number of stations*. This is the number of stations on the LAN.
- *Frame size.* This is the size of a frame in bytes with values from 64 bytes to 1024 bytes.
- *Time between transmissions*. This is the mean idle time after a frame has been either transmitted or discarded (because 16 attempts have been made) until the next transmission attempt.

The simulation outputs are:

- *Transmit delay*. This is the average time delay per successful frames.
- *Protocol efficiency*. This is the ratio of the number of successful frames to the maximum number possible when the LAN is being operated at its maximum rate.

There are two important objects in the Ethernet protocol – “Stations” which communicate with each other on the channel and “Frames” with which they communicate. There are five states for a station: “Idle”, “Wait”, “Listen”, “Contention” and “Finish”. The state transit diagram of a station is shown in Figure 3.1. When the simulation is initialized, every station is in “Idle” state. A station goes to “Wait” state if it wants to send a frame through the channel. If the channel is clear, it transmits the frame to the cable immediately. Otherwise, it blocks until a clear channel is found. After transmitting a frame, a station waits for 2-slot time. If no collision occurs in this period of time, the package was transmitted successfully; otherwise, the station has collided. If collision occurs, the station waits for a random period of time, go back to “Wait” state and retransmit the frame. However, if the number of retransmissions is larger than 16, the transmission fails and the station goes back to “Idle” state and begin its next activation.

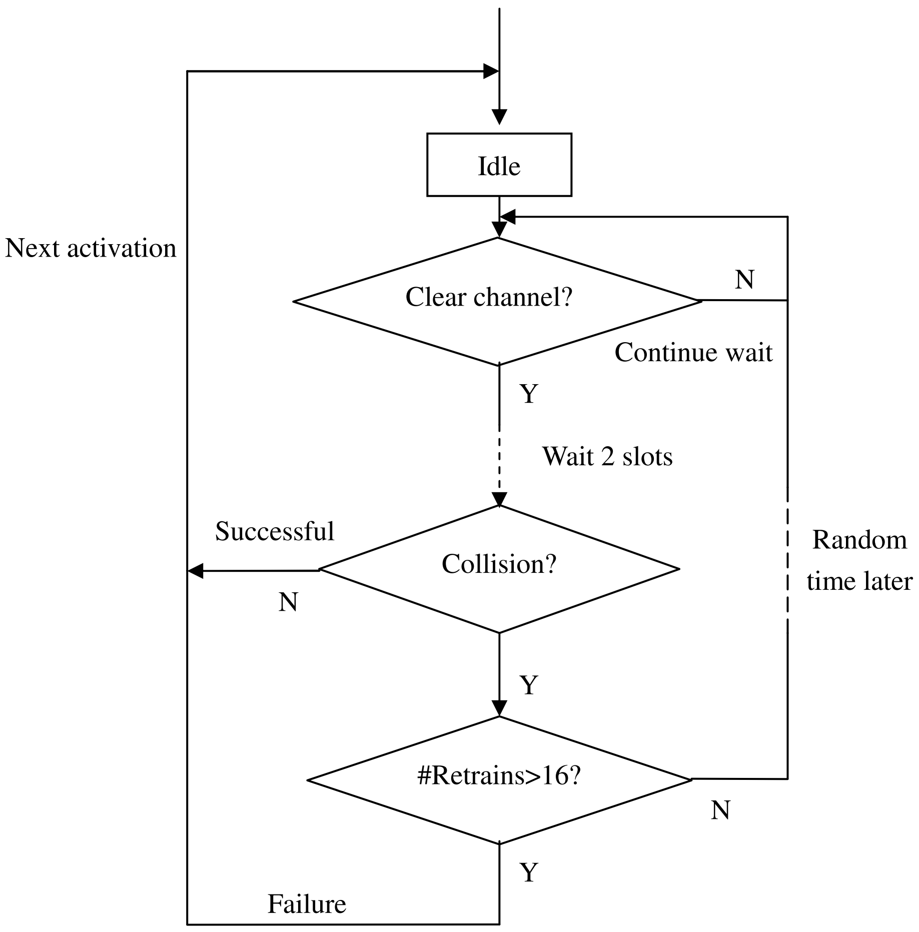


Figure 3.1: State transit diagram of an Ethernet station

There are three states for a frame: “Arrival”, “Transit” and “Departure”. The “Transit” state can be modeled implicitly after the “Arrival” state and before the “Departure” state. When a frame arrives at a station, it passes through it without delay. All frames contend for the clear channel in Ethernet network. A station must wait for an empty channel to transmit frames. The *frame transmission time* (frame-size divided by LAN-speed) cannot be neglected, so a frame cannot be modeled as one process. We model two processes for a frame: *frame head* (the first bit of a frame) and *frame tail* (the last bit of a frame). Every time a station wants to transmit frames, it schedules the frame head activation immediately, but schedules the frame tail activation *frame transmission time* later. When a frame arrives at a station which is in the “Listen” state (within 2-slot time after transmission), the transmission is flagged as failed. The station then goes to “Contention” state and corrupts the frame(s) it sent.

A station is also modeled as a resource in parallel simulation. Frame head or tail stays at a station for a period of time equal to the *neighbour delay time* (distance between two neighbouring stations divided by signal transmission speed $2*10^8$ m/s). There is no need to model conditional event list (CEL) for a station resource because it is a delay center where infinite service units are available. The station process always stays at a station resource and does not move, while a frame moves between stations.

In summary, there are 4 objects in our conceptual model of Ethernet network as shown in Figure 3.2: station process, station resource, frame head process and frame tail process. The frame head and tail are modeled separately. Frame head arrival at a station means the arrival of the frame. Frame tail departure from a station refers to the departure of the frame. If a frame is corrupted within 2 slot time, the frame's tail will be rescheduled to transmit immediately. Otherwise, the transmission is flagged as successful.

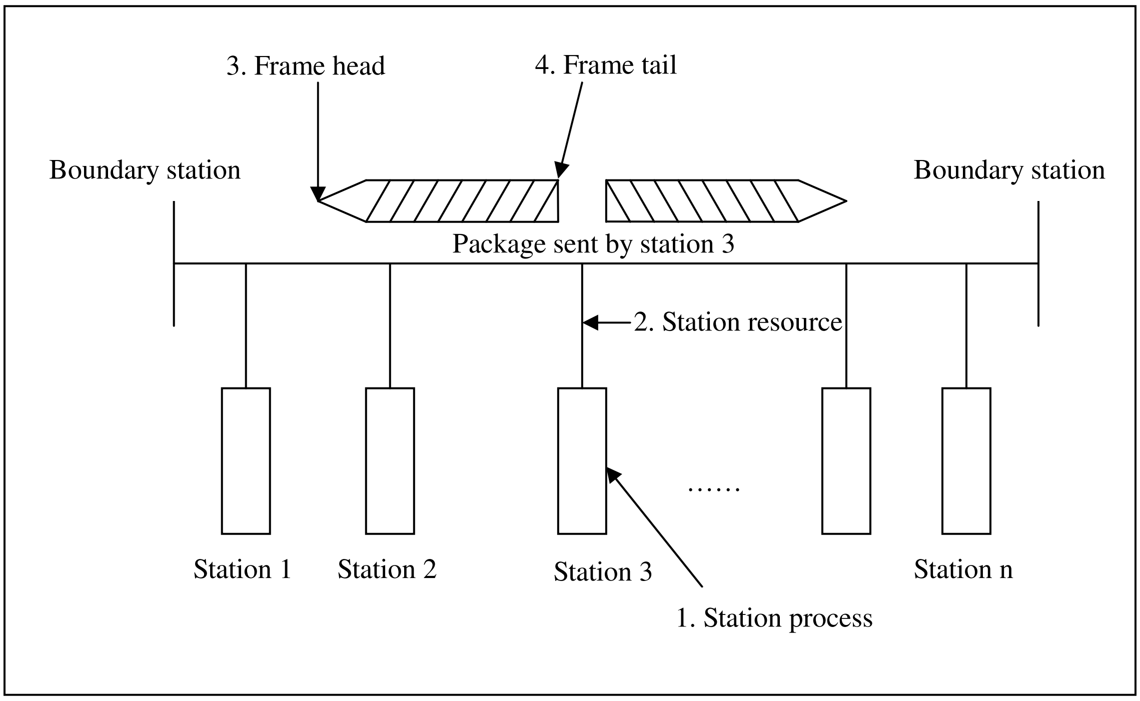


Figure 3.2: Processes and resources in simulation model of Ethernet.

3.3 Simulation and Implementation

At the implementation level, some assumptions are made about our simulated Ethernet network. Firstly, we assume there is a straight cable without any branches. All stations are equally distributed along the cable. The channel is noiseless and the station is reliable. Frames travel along the cable from the source station in both directions and are absorbed at the cable ends. The time between transmissions is exponentially distributed. Protocol efficiency is required with all stations working continuously, i.e. no idle periods. Ethernet standards specify a maximum delay of 51.2 μs (slot size) but this is supposed to include the transmit delay through repeaters. The length of cable is assumed to have the maximum delay equally distributed throughout its entire length (approximately 10 km), which is the product of the slot size (51.2 μs) and the signal transmission speed ($2*10^8$ m/s).

3.3.1 Sequential

We implemented the simulator with SPaDES/Java and validated it with the C simulator developed by Watkins [45]. There are two classes of active objects in Ethernet network: station and frame. In addition, one kernel class is used to initialize the simulation parameters, start the simulation and print the simulation results. The simulation terminates when the simulated time exceeds a specified duration. The kernel class is also responsible for initially activating the station classes. The kernel class is listed in Figure 3.3.

```
import spades_Java.*;

//EthernetKernel.java
// Executive instance
public class EthernetKernel extends Executive{
        :
        Resource Service[];
        //station process
        Station station[];
        :
        //define parameters for Ethernet
        double LAN_Speed;    /* LAN speed in bps */
        double cable_length; /* Cable length in meter */
        int num_stations;    /* Number of users in the LAN */
        int frame_size;      /* Frame size in bytes */
```

```
     double duration;     /* Simulation duration in ms */
     :
     public void init(){
         :
         <<Initialize Ethernet parameters>>
         :
         <<Initialize resource>>
         :
         //Active station process
         for (int i=0, i<num_stations, i++){
              :
              station[j]=new Station("Station "+j,this);
              <<Initialize station[i]>>
              mapProcess(station[j],Service[j]);
              activate(station[j], 0);
              :
         }
      }
      public static void main(String[] args){
            EthernetKernel ek = new EthernetKernel();
            ek.initialize(args.length, args);
            ek.startSimulation(duration);
            <<Print simulation results>>
      }
}
```

Figure 3.3: Kernel class of Ethernet network simulation

Once a station entity has been created it exists permanently for the life of the simulation and is involved in a succession of transmission attempts. The following pseudo code describes the behavior of a station class.

```
import spades_Java.*;
Class Station extends SProcess{
     :
     int successful;      /* Number of frames sent */
     int number_retrans;  /* Number of retransmission so far */
     Aframe[] end;        /* Frames transmitted out */
     <<Initialize other Station parameters>>
     :
     public void execute(){
       switch(phase){
           case Idle:{
                <<Set phase to Wait>>
                wait(0.0);
                break;
           }
           case Wait:{
                if <<Channel is clear>> {
                     <<Create and transmit frames>>
                     <<Set phase to Listen>>
20:                         wait(2*slot_time);
21:                   }else{
22:                         //continue waiting
23:                         susPend();
24:                   }
25:                   break;
26:             }
27:             case Listen:{
28:                   if <<Frame is or will be successfully
29:                          transmitted>>{
30:                         if <<Both frames transmitted>>{
31:                               <<Set phase to Finish>>
32:                               wait(0.0);
33:                         }else{
34:                               <<Wait until 2 frames left>>
35:                               <<Set phase to Finish>>
36:                               susPend();
37:                         }
38:                   }else{
39:                         <<Retransmit this frame>>
40:                   }
41:                   break;
42:             }
43:             case Contention:{
44:                   <<Set phase to Wait>>
45:                   wait(0.0);
46:                   break;
47:             }
48:             case Finish:{
49:                   <<Reset this station>>
50:                   <<Set phase to Idle>>
51:                   wait(0.0);
52:                   break;
53:             }
54:             }
55:       }
56: }
```

Figure 3.4: A station class

The station class follows straightly from the state transit diagram of a station in Figure 3.1. A station needs to record the number of frames it sent out (Line 4) and number of retransmissions so far in this station (Line 5). If a clear channel is found, the station transmits two (one for boundary station) frames in both directions. The station schedules the frame head's arrival at its neighbouring station *neighbour delay time* later. In the mean time, the station schedules the frame tail's departure from itself *frame transmission time* later. The station also needs to record frames that it sent (Line 6) because it is necessary for a station to identify the frames it sent when a collision occurs. After transmitting two frames, the station will listen for 2 slot size time. If no other frames arrive at this station in this period of time, the transmission is successful. If both frames have been transmitted, the station will go to "Finish" state and prepare for another transmission some idle time (exponentially distributed) later. Otherwise the station will suspend itself and wait for another frame's departure. However, if some frame(s) arrive at this station in 2 slot size time, the transmission is considered to fail.

Once a collision has occurred a time delay must be calculated after which a retransmission can be attempted. However, if the maximum number of attempts has already been made, the transmission request must be rejected. The function *retrans_time()* in Figure 3.5 calculates the retransmission time according to binary exponential back-off algorithm. The actual retransmission is performed with *retransmit()*, which is called in phase "Listen" after the two-slot delay.

```
1: class Station extends SProcess {
2:         :
3:      void retransmit(Station this_station){
4:              /*Have maximum number of attempts been made?*/
5:              if (<<Number of retransmission = 16>>){
6:                    <<Set phase to Finish>>
7:                    wait(noise_burst*1.005);
8:              }else{
9:                    <<Set phase to Contention>>
10:                    /*Wait for a random and restransmit*/
11:                    wait(retrans_time(this_station));
12:              }
13:       }
14:       :
15:       float retrans_time(int this_station){
16:                  int   t;
```

```
                int   num;
                float  maximum;
                number_retrans++;
                num = number_trans;
                if (num>10){
                      num=10;
                }
                maximum=2^num;
                t=(int)uniform(0,maximum);
                return slot_size*t;
      }
      :

      public void execute(){
            switch(phase){
            :
            case Listen:{
                  if <<Frame is or will be successfully
                         transmitted>>{
                        :
                  }else{
                        <<Retransmit this frame>>
                  }
                  break;
            }
            :
       }
}
```

Figure 3.5: Retransmission mechanism

It is necessary for a station class to determine when it interacts with a frame class. If a station cannot find a clear channel in “Wait” state, it will be passivated until a departure frame clears the channel. When the last frame has left the station and is propagating down the cable the station must be reactivated so that the next transmission can be made. Note that in the case of very short frames of less than 128 bytes, this would not be performed because by the time the station had been activated after the initial two-slot delay both frames would already have been transmitted.

Now let us look at the frame class. The following pseudo code outlines the frame class.

```
import spades_Java.*;
Class Aframe extends SProcess{
      :
      int source;           /* The sender station of the frame */
      int direction;        /* The direction of propagation */
      int arr_sta;          /* The station arriving at */
      int dep_sta;          /* The station leaving from */
      <<initialize other parameters of a frame>>
      :
      public void execute(){
            switch(phase){
            case Arrival:{
                 int this_stn=this.arr_stn;
                 if (<<The arrival station is in Listen state>>
                     and <<The station is not corrupted>>){
                       invalidate_frame(this_stn);
                 }

                 if (<<Frame going to left>>){
                       <<Flag a frame passing by from left>>
                       if (<<Frame not at end of cable.>>){
                             wait(neighbour_delay);
                       }else{
                             terminate();
                       }
                 }else{//Frame going to right
                       <<Flag a frame passing by from right>>
                       if (<<Frame not at end of cable.>>){
                             wait(neighbour_delay);
                       }else{
                             terminate();
                 }

                 break;
            }
            case Departure:{
                 if (<<This is the frame’s source station>>){
                      <<Flag a frame has been transmitted>>
                      if (<<Finished both frames>>){
                            if (<<Frame not corrupted>> and
                         <<Station is not in Listen state>>){
                               reactivate(<<This station>>);
                            }
                      }
                 }

                 if (<<Frame is going to left>>){
                      <<Frame passing by decrease by 1>>
                      if (<<Frame can be sent to left>>){
                            wait(neighbour_delay);
                      }else{
                            terminate();
                      }
                 }else{//Frame is going to right
55:                     <<Frame passing by decrease by 1>>
56:                     if (<<Frame can be sent to right>>){
57:                          wait(neighbour_delay);
58:                     }else{
59:                          terminate();
60:                     }
61:                }
62:
63:                if (<<Channel is clear after this departure>>
64:                  and  <<This station suspends>>){
65:                       reactivate(<<this station>>);
66:                }
67:
68:                break;
69:           }
70:      }
71: }
```

Figure 3.6: A frame class

An arriving frame schedules its possible arrival at its neighbouring stations and increases the number of frames passing by the station. If the station happens to be a boundary station, the frame will be terminated. When a frame departs from a station, it will first test whether the station is the frame’s source station. If it is, the frame will be flagged to be transmitted. If the frame is not corrupted, we will activate the station to transmit another frame. The departing frame will schedule its departure from neighbouring station and decrease the number of frames passing by the station. If the station is a boundary station, the frame will be terminated.

We have seen before that a station needs to suspend to wait for a frame’s activation. The activation of the waiting station is suspended if the channel is busy. Only a departure event can make the channel clear, so the channel will be checked after every departure event. The waiting station will be reactivated if a departure event makes the channel clear (Line 65). Another interaction occurs when the last frame is sent out and the station is reactivated to make the next transmission (Line 42).

When a frame arrives at a station that is in “Listen” state, it will collide with the station. The corrupted frames will be passivated (extracted from FEL) and be rescheduled to occur some noise time (noise_burst) later. The pseudo code to handle collisions is listed in Figure 3.7.

```
Class Aframe extends SProcess{
      :
      public void execute(){
            switch(phase){
            case Arrival:{
                 int this_stn=this.arr_stn;
                 if (<<The arrival station is in Listen state>>
                     and <<The station is not corrupted>>){
                       invalidate_frame(this_stn);
                     }
          :
          void invalidate_frame(int station){
                 <<Mark this station’s transmission to be fail>>
                 if <<Station transmit frame left>>{
                       //remove from FEL
                       passivate(<<Left frame>>);
                       <<Left frame>>.wait(noise_burst);
                 }
                 if <<Station transmit frame right>>{
                       passivate(<<Right frame>>);
                       <<Right frame>>.wait(noise_burst);
                 }
          }
          :
    }
```

Figure 3.7: Collision handler

If all stations are activated to occur at simulation time 0 at the initial stage, each station will try to transmit two (one for boundary station) frames in both directions. Then the simulation will run according to the aforementioned mechanism. The whole pseudo source code of Ethernet network simulation is presented in Appendix C.

As CSMA/CD protocol is the essential part of Ethernet, it is implemented in our simulator. But some other researchers choose to eliminate the implementation of CSMA/CD protocol to reduce the implementation complexity [44].

3.3.2 Parallel

In SPaDES/Java, every LP maps to a resource. A resource maintains its own future event list and executes events from its own event list. Null message synchronization protocol is used to ensure event causality. A station is mapped to a resource in Ethernet network simulation. Null message protocol used in SPaDES/Java requires that users statically specify the links that indicate which LP may communicate with which other LPs. Obviously there are links between two neighbouring stations. Additional links exist from one LP to itself because the frame head's arrival is scheduled by itself. Therefore the station in Ethernet network is self-transitive. Both null messages and event messages transmit through links. The links for a 3-station Ethernet network are illustrated in Figure 3.8.

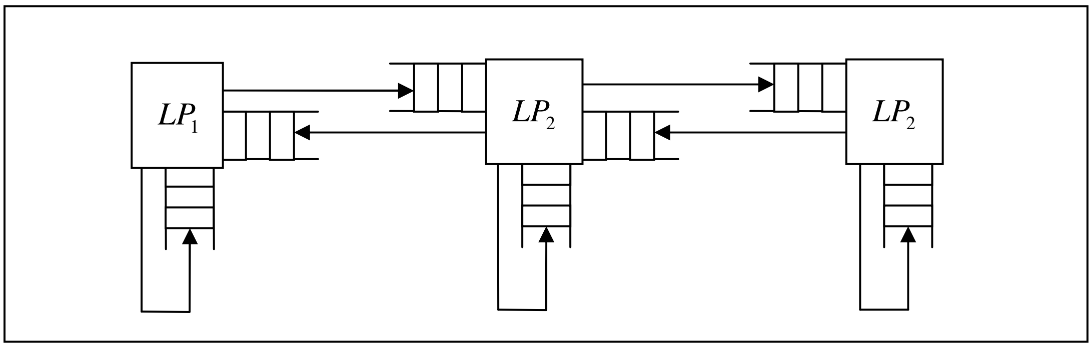


Figure 3.8: Links in a 3-station Ethernet network

An event message enters an LP's output link only when the LP executes a process' arrival event. SPaDES/Java incurs two activities for an arrival event. One is to schedule the process' departure event in local FEL. Another is to call the process' execution and send the process to its output channel. The scheduled departure event in FEL is in fact a dummy event with the departure timestamp and the process name. When the dummy event is executed, it will remove the actual departure event in the output link and send the event to its neighbour LP. The neighbouring LP then puts the process' arrival event to its FEL. This is the mechanism SPaDES/Java transmits an event between two linked LPs.

SPaDES/Java provides the same primitives for sequential and parallel simulation. Hence it is easy to use the sequential simulation codes for parallel execution. One difference that exists between sequential and parallel implementation is frame passing through a station. Frame transmits from one station to its neighbour after *neighbour delay time*. This delay is modeled by calling *wait()* primitive in sequential implementation. Global FEL then sorts all processes. The sequential implementation of frame passing is shown in Figure 3.9a.

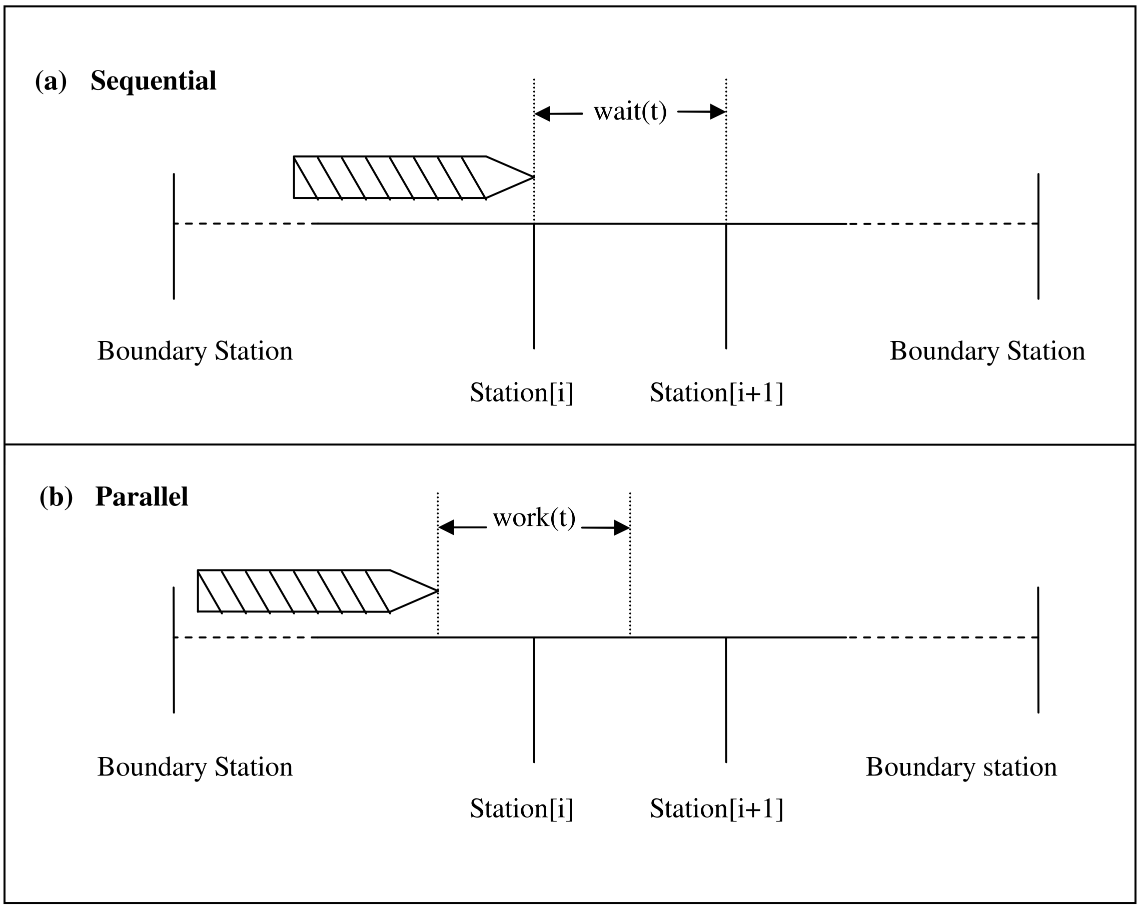


Figure 3.9: Sequential and parallel implementation of frame passing

However, this approach cannot be applied to parallel simulation because arrival events at two neighbouring stations are located at different LPs. The transmission delay between two stations is now modeled as the service time delay in resource. Hence a station resource is modeled as a delay center where a frame can get service immediately when it arrives at a station. The response time of all delay centers is zero and the transmission delay from one station to its neighbour is also zero. The *work()* primitive is called to schedule its arrival at neighbour delay when a message arrives at a station. This mechanism is a little more complicated than the mechanism used in sequential implementation. However, the two approaches are equivalent. Figure 3.9b illustrates the parallel implementation of message transmitting between stations.

3.4 Verification

Simulation verification ensures that the simulator program implements conceptual model correctly [30]. In this study, the Ethernet simulator is developed with a special-purpose simulation language SPaDES/Java, not one general purpose higher order language such as PASCAL or FORTRAN. A simulation language/library will usually provide the sub-model for each simulation function (e.g., time-flow mechanism, process and resource manager, random number and random variants generators, and integration routines). Therefore, using a special purpose simulation language will not only reduce the programming time, but also increase the probability of having a correct program.

The simulation library, SPaDES/Java, has preciously been verified by other queuing network applications [40, 41, 42]. Here we only illustrate the verification of the Ethernet simulator.

Firstly, the development is to start with a simplified version of the model and then to refine it in a number of steps. In the first step, the modeling of the complicated frame collisions and the resulting retransmission strategy is completed ignored. Then the retransmission strategy can be incorporated in a second step and the necessary data collection and analysis functionality in a third step. The parallel Ethernet simulator was developed in the last step. This multistage development and verification limit the errors in a short development period and avoids the major revision to the simulator.

Secondly, the code was reviewed by people other than the author to check the model logic.

Thirdly, the simulator produced a trace file which consists of detailed event execution representing the step-by-step progress of the simulator over the simulation time. The trace file recorded the detail information for every event generated from the simulator, including the customer ID, event's timestamp and the LP. The trace file is similar to the one listed in Appendix A. This method allows detection of subtle errors. The trace file displayed that some events may occur but not be scheduled to TSA in the later instrumentation stage. It turned out the causal relationship between events are not correctly maintained by TSA.

Fourthly, one test program is written to check the execution of Ethernet simulator. The purpose of using test program is to guarantee that the following conditions are correctly held:

1. The timestamp of every message into one LP is always larger than its timestamp out of the LP;

2. The arrival time of every message to the receiver LP is always larger than its departure time from the sender LP;
3. The number of messages in FEL is always larger than or equal to zero;
4. Finally, the number of messages into an internal LP is equal to the number of messages away from it;

Some extreme-conditions are also tested in every stage of the Ethernet simulator development. For example, we test the case where only two stations in the Ethernet network contend the channel. Such an extreme condition will allow us to check the correctness of the collision and retransmission modeling easily.

Finally, the state of the simulated system, i.e., the contents of the event list, state variables, statistical counters were printed and checked with the model logic.

3.5 Validation

The aim of validation in a simulation study is to ensure confidence in the study's results. A model is sound and dependable if it accomplishes what is expected [10]. Developing a simulation model is an iterative process with successive refinements at each stage. Hence validation occurs at several stages in a simulation project. Our experiments are validated at two stages. One existing Ethernet model by CSIM [45] is used to validate our implementation using SPaDES/Java. We assume that the CSIM model is itself validated [31]. In order to validate the sequential implementation, we compare our simulation results with those by the CSIM model.

Next, we validate the parallel implementation.

3.5.1 Sequential Simulation

Binary exponential back-off algorithm uses uniformly distributed number generation to calculate retransmission delay. We also assume an exponentially distributed mean idle time between transmissions. CSIM and SPaDES/Java use different pseudo random number seed value for generating random variants. In the first step the *fix value* [30] validation is used and the random mechanism is excluded from the simulator.

Let us look at the validation process in detail now. We generate simulation results from both the CSIM version and the sequential SPaDES/Java version. Comparison between the two simulation results is used to validate our implementation. In fix value validation, all model input and internal variables are fixed and we can check the model results against hand calculated values. Firstly, the random variant is replaced in the binary exponential back-off algorithm by changing the back-off in terms of the number of slots to $(n+1)/2$. The delay time between frame corruption and retransmission also increases with number of retransmissions. Next, the mean idle time between transmissions is fixed at 0.1 ms. All stations are initialized to activate at different times, i.e., increment of 0.1 ms, 0 ms for station 0, 0.1 ms for station 1, 0.2 ms for station 2, and so on. The simulation duration is *100* seconds. Because of the fine simulation resolution, this generates about 10^8 events during this period of simulated time.

Table 3.1 shows the comparison of simulation results between CSIM model and sequential SPaDES/Java model. The output of transmit and efficiency is already defined in the simulation model (section 3.2). The two programs produce the same simulation results. Therefore the SPaDES/Java model is validated by the CSIM one.

Parameters		CSIM		SPaDES/Java	
#Stations	Frame size (bytes)	Transmit delay (ms)	Efficiency (%)	Transmit delay (ms)	Efficiency (%)
10	128	0.15	83.0	0.15	83.0
	256	0.12	96.0	0.12	96.0
20	128	0.24	83.1	0.24	83.1
	256	0.26	90.9	0.26	90.9
30	128	0.84	76.6	0.84	76.6
	256	0.42	89.5	0.42	89.5

Table 3.1: Validation of SPaDES/Java Ethernet simulator (fix value)

The simulator output, transmit delay and efficiency, in Table 3.1 is only used for validation because we use fix value method to validate the Ethernet simulator.

Although we exclude the effect of random variants in the two simulation libraries, two implementations may schedule simultaneous events to occur at different orders. To save future events, CSIM uses a *tertiary tree* while SPaDES/Java uses a *binary minheap*. Heap-based sort is *unstable* because two simultaneous events may not keep their original order after sorting. This prevents problems to be completely validated.

In the second step, we validated sequential SPaDES/Java Ethernet simulator by comparing our simulation outputs with those by Watkins' model. Because simulation modeling normally requires repeatability, random number generator can always produce the same sequence of random numbers starting with the same initial condition (seed). Therefore we use the same number stream for both the CSIM simulator and SPaDES/Java simulator to compare the simulation outputs. The two simulation outputs are listed in Table 3.2. The mean idle time is 0 and the simulation time is 1000 ms.

Parameters		CSIM		SPaDES/Java	
#Stations	Frame size (bytes)	Transmit delay (ms)	Efficiency (%)	Transmit delay (ms)	Efficiency (%)
10	1024	1.13	92.8	1.13	92.8
	256	0.52	91.0	0.52	91.0
20	1024	4.42	88.9	4.42	88.9
	256	1.92	79.4	1.92	79.4
60	1024	15.4	85.4	15.4	85.4
	256	12.6	62.3	12.6	62.3
200	1024	35.3	80.3	35.3	80.3
	256	28.7	60.0	28.7	60.0

Table 3.2: Validation of SPaDES/Java Ethernet simulator

Our simulation results are the same as those generated by Watkins' and higher than those from other studies [18, 34]. Different model assumptions are used in [18, 34] and the performance results are obtained from theoretical analysis. For example, they suppose "that n stations are contending for the channel and suppose that each station transmits during a contention mini-slot with probability *p*." Their results are derived directly from the paper of Metcalfe and Boggs [24]. Many simulation and theoretical studies of Ethernet assume a simple distribution for the arrival of packets. Poisson distribution is usually used. However, real network traffic often consists of heavy load that are divided by long period light traffic [2].

As in Watkins' model, our Ethernet network simulator assumes there is no idle period, i.e. network load is continuous and heavy. Therefore the network throughput and efficiency are higher.

3.5.2 Parallel Simulation

A similar method is exploited to validate the parallel implementation. Ethernet simulation results of parallel SPaDES/Java version are compared with CSIM version. Finally we produced the same simulation results between CSIM version and parallel SPaDES/Java version as in Table 3.2 and thus validate the parallel implementation.

3.6 Events Instrumentation

We instrumented the Ethernet network simulator to obtain event sequence and causal relationships. There are five types of events in Ethernet network simulation, shown in Figure 3.10: (1) *External Arrival*: A frame arrives from external environment to the Ethernet network. An external arrival event is scheduled only if a station finds a clear channel to transmit frames; (2) *Internal Arrival*: A frame arrives at an intermediate station from the neighbouring station, scheduled by its neighbouring station; (3) *Boundary Arrival*: A frame arrives at a boundary station from its neighbor, scheduled by its neighbor's arrival event; (4) *Internal Departure*: A frame departs from one intermediate station, scheduled either by this frame's external arrival event or its internal departure from the neighbouring station; (5) *Boundary Departure*: A frame departs from one boundary station, scheduled by neighbor’s internal departure.

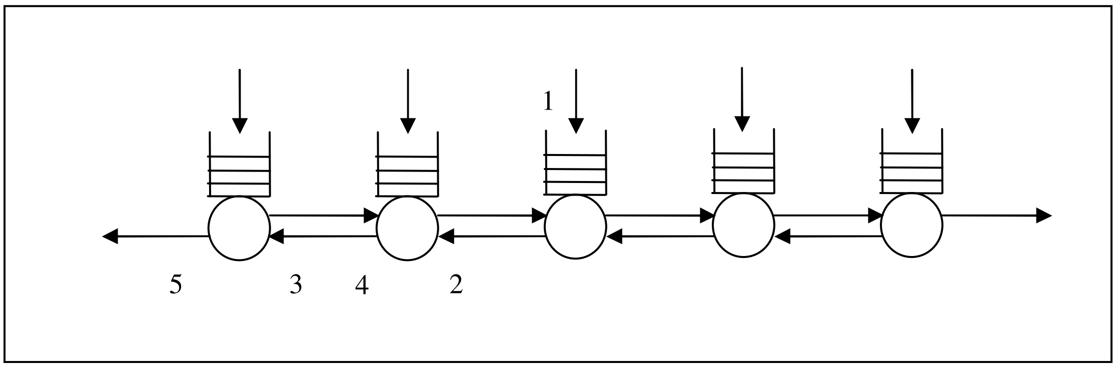


Figure 3.10: Five types of events in Ethernet network simulator

A station loops over some states, but can stay only at one state at any particular point of the simulation time. Hence every station process has only one event in the LP’s FEL and it does not change with event orderings. Therefore we do not consider events for station activation. We concentrate on the activation of frames and the interaction between stations and frames in our experiments.

TSA requires that one knows all the causal relationships between events so that one can analyze the performance based on causal restriction. We record every event together with its antecedent event. An event can be executed only after its antecedent event has been executed according to causal restriction.

It is complicated to find causal antecedent events for external arrival events. A station will schedule an external arrival event when it is in “Wait” state and a clear channel is available. We check back from this point to find which other events may be the antecedent event of an external event in Ethernet’s control flow graph (CFG), shown in Figure 3.11. The following six paths give the possible antecedent events of an external arrival event.

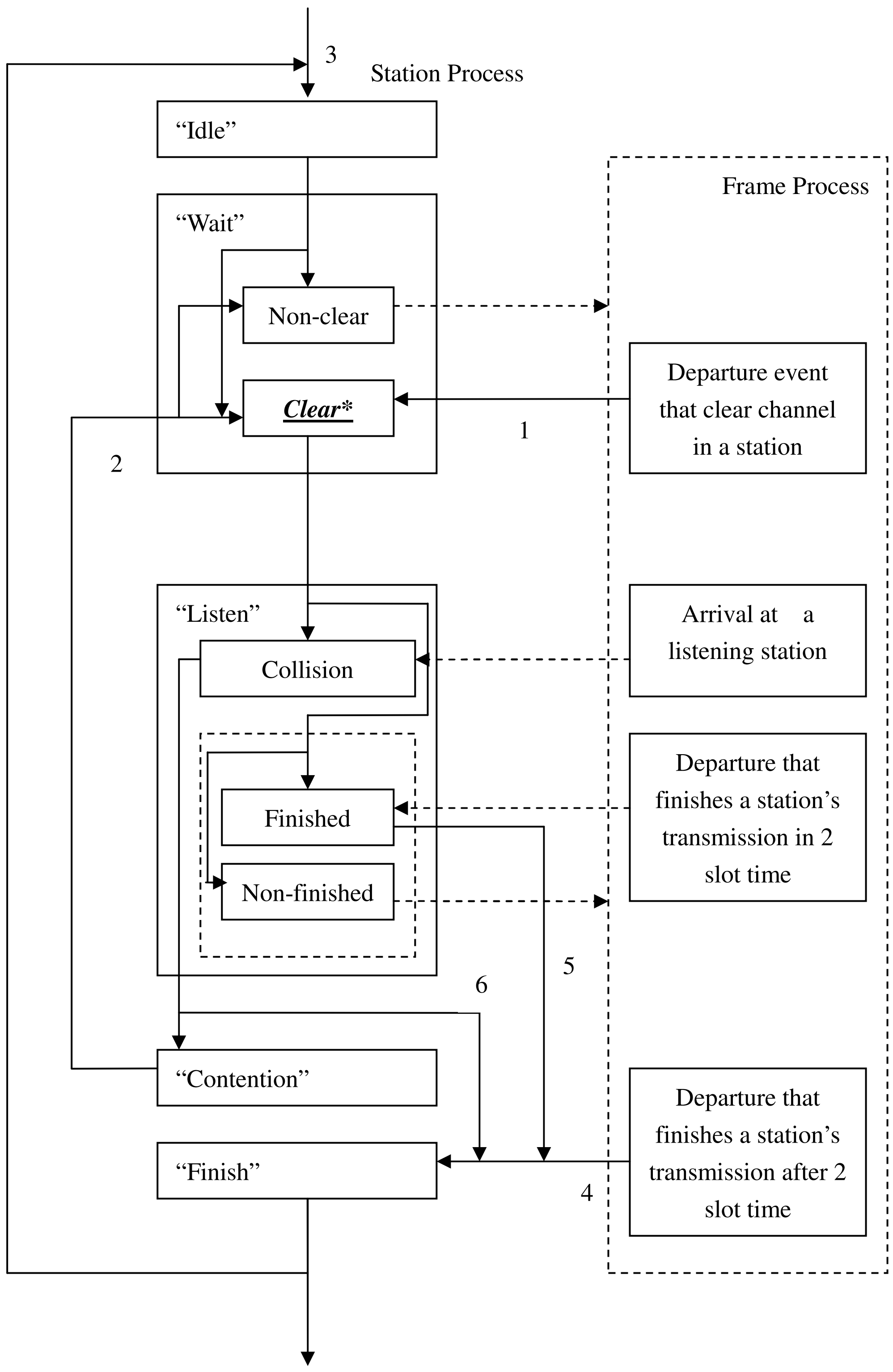

3
Station Process
"Idle"
Frame Process
"Wait"
Non-clear
Departure event that clear channel in a station
Clear*
1
2
"Listen"
Arrival at a listening station
Collision
Departure that finishes a station's transmission in 2 slot time
Finished
Non-finished
5
6
"Contention"
Departure that finishes a station's transmission after 2 slot time
"Finish"
4

Figure 3.11: Control flow graph (CFG) of Ethernet simulator

- Path 1: The channel becomes clear after a frame has departed. The departure event will then reactivate the station that is suspended on the “Wait” state. The departure event thus becomes the antecedent event of such an external arrival event.
- Path 2: A frame arrives at a listening station and collides with the station. The station will try to retransmit its frame(s) if the number of retransmissions is less than 16. If the channel happens to be empty when the station returns to “Wait” state, an external arrival event will be scheduled. Hence the arrival event that makes the station collided becomes the antecedent event of an external arrival event in this circumstance.
- Path 3: In the initial configuration all stations will find clear channel and transmit frames immediately. There is no antecedent event for such an external arrival event.
- Path 4: A successfully-transmitted frame will depart from its source station after 2 slot size time if the frame size is not less than 128 bytes. After the last frame is departed, the station will successfully finish this transmission and go to “Finish” state. If the station finds a clear channel immediately in the next loop of transmission, it will schedule one external arrival event. Hence the departure event will be the antecedent event of the next loop’s external arrival event.
- Path 5: The situation is similar to path 4 except that the departed frame size is less than 128 bytes. When the frame departs from its source station, the station is still within 2-slot time and at “Listen” state. The station will go to “Finish” state

when 2-slot time passed from its initial transmission. If the station finds a clear channel immediately in the next loop of transmission, it will schedule one external arrival event. This departure event thus becomes the antecedent event of next loop's external arrival event.

- Path 6: If a frame arrives at a listening station, it will collide with the station. If the number of retransmissions so far is already 16, the transmission of the station is considered to fail and the station will go to "Finish" state. Similar to path 4 and path 5, if the station finds a clear channel immediately in the next loop of transmission, it will schedule one external arrival event. Therefore the corrupted arrival event becomes the antecedent event of next loop's external arrival event.

Collision occurs both in path 2 and in path 6. The difference between the two situations relies on the number of retransmissions. The number of retransmissions is less than 16 in path 2, so the station goes back to "Wait" state to retransmit. But in path 6, 16 retransmissions flag a failed transmission and the station goes to "Finish" state. Frame transmission is successful both in path 4 and in path 5. But frame size is less than 128 bytes in path 4 and larger than or equal to 128 bytes in path 5.

The five basic event types are classified in detail to 11 event types according to our analysis. Table 3.3 lists all of them and their scheduling information.

Event Type	Scheduling	Comment
External Arrival	$A_k^{p1} \begin{cases} \xrightarrow{t1} A_k^{p2} \\ \xrightarrow{t2*} D_k^{p1} \end{cases}$	*Collision may reschedule the departure less than* t_2
Internal Arrival 1	$A_k^{p2} \xrightarrow{t1} A_k^{p3}$	*Schedule arrival at its neighbour*
Internal Arrival 2	$A_k^{p2} \begin{cases} \xrightarrow{t1} A_k^{p3} \\ \xrightarrow{0} collision \to A_m^{p2} \\ \xrightarrow{0} collision \to A_n^{p2} \end{cases}$	*Collision and retransmission (path2) or next loop transmission (path6) is successful*
Boundary Arrival 1	$A_k^{pn} \to null$	*Frame will stop transmit in boundary station*
Boundary Arrival 2	$A_k^{pn} \xrightarrow{0} collision \to A_m^{pn}$	
Internal Departure 1	$D_k^{p2} \xrightarrow{t1} D_k^{p3}$	*Schedule departure from its neighbour*
Internal Departure 2	$D_k^{p2} \begin{cases} \xrightarrow{t1} D_k^{p3} \\ \xrightarrow{0} A_m^{p2} \\ \xrightarrow{0} A_n^{p2} \end{cases}$	*Path1*
Internal Departure 3	$D_k^{p2} \begin{cases} \xrightarrow{t1} D_k^{p3} \\ \xrightarrow{idletime} A_m^{p2} \\ \xrightarrow{idletime} A_n^{p2} \end{cases}$	*Path4 or path5 for an internal station*
Boundary Departure 1	$D_k^{pn} \to null$	
Boundary Departure 2	$D_k^{pn} \xrightarrow{0} A_m^{pn}$	*Path 1*
Boundary Departure 3	$D_k^{p1} \begin{cases} \xrightarrow{t1} D_k^{p2} \\ \xrightarrow{idletime} A_m^{p1} \end{cases}$	*Path4 and path5 for a boundary station*

* A_k^{p1} *refers to frame* k *arrival on service center* $p1$, *and similarly* D *refers to departure event.* t_1 *is neighbour delay time and* t_2 *is frame transmission time*

Table 3.3: Event types and their scheduling information in Ethernet simulator

Let us explain the scheduling relationships of the 11 types of events. External arrival A_k^{p1} schedules both the frame's arrival event to neighbouring station A_k^{p2} and the frame's departure event from source station D_k^{p1}. A_k^{p2} will be executed *neighbor delay time* (t_1) later because the head of the frame will take that period of time to arrive at its neighbouring station. D_k^{p1} will be executed frame-trans-time (t_2) later because it takes a frame that period of time to pass through a station.

There are two other kinds of arrival events for a frame: *internal arrival* and *boundary arrival*. Internal arrival means the arrival of a frame at an intermediate station and boundary arrival means the arrival at a boundary station. Internal arrival event schedules its arrival at neighbouring station t_1 later (internal arrival 1). If this arrival event incurs a collision and the collided station finds a clear channel in the next transmission, other two external arrival events will be scheduled (internal arrival 2). Similarly, there are two event types (boundary arrival 1 and boundary arrival 2) for boundary arrival events.

A frame's internal departure event schedules its departure from neighbouring station $t1$ time later (internal departure 1). If a frame departs from a boundary station, no new events will be scheduled (boundary departure 1). If the departure event clears the channel where a station is suspended, the station will be reactivated and schedule two new frames immediately (internal departure 2). When the last uncorrupted frame departs from its source station (internal station), the transmission is flagged to be successful and the station will go to "Finish" state. If the station can find a clear channel in the first transmission of next activation, it will schedule two new frames (internal departure 3).

Similarly, there are three types of events for boundary departure event. A frame departing from a boundary station won't schedule new events (boundary departure 1). If a departure event clears the channel in a suspended boundary station, the station can schedule one new frame immediately (boundary departure 2). If a successful frame departs from its source station, which is a boundary station, it will schedule its departure from its neighbouring station *t*1 later and also flag a successful transmission. Then the boundary station returns to "Finish" state and attempts a new transmission. If it finds a clear channel immediately, a new frame will be scheduled (boundary departure 3).

There are three special departure events corresponding to path 1, 4 and 5 in Ethernet's CFG. However, one departure event belong s to one path for one frame can depart from only one station in one particular state:

- Path 1: This station is in "Wait" state;
- Path 4: This station is in "Finish" state;
- Path 5: This station is in "Listen" state.

Path 3 in Figure 3.10 corresponds to the initial configuration in Ethernet network simulation where every station's FEL length is set to 2 (1 for boundary station). We instrumented a sequential Ethernet network simulator to obtain list of events. Sometimes we are not certain about which type an event is going to be when it is just obtained from FEL. We need to wait until a station's next transmission to confirm its type.

3.7 Summary

We introduced in this chapter, our Ethernet model and its implementation. The Ethernet network was presented from three levels: problem, conceptual model and implementation. In the conceptual model, we discussed the activation of stations and frames. The simulator is implemented using SPaDES/Java and validated using a CSIM model. Lastly we presented how to instrument and obtain event sequence from Ethernet network simulation.

Chapter 4

Experimental Results and Analysis

The experimental results and analysis are introduced in this chapter. Average event parallelism and profile of memory requirement are based on that required by the physical system, as a result of different event orderings and the overhead of simulation synchronization. Both event parallelism and memory requirement are illustrated from three steps. Lastly, we present the performance tradeoff.

4.1 Event Parallelism

As presented in Chapter 1, the event parallelism is defined as the average number of events executed per unit time. Event parallelism is measured from three steps in our experiments: physical problem, event orderings and implementation.

4.1.1 Problem

Event parallelism existing in the problem is referred to as the inherent event parallelism (Π_{prob}). As illustrated in Chapter 1, Π_{prob} is measured as the sum of all service centers' utilization. The utilization of a service center is defined as λ / μ, where λ is the arrival rate and μ is the service rate of the service center. We can directly measure these values in our simulator, calculate the utilization for all stations, and then obtain Π_{prob}.

Before presenting the Π_{prob} results, we first determine the parameters of the simulated Ethernet network. Workload characterization consists of a description of the workload by means of quantitative parameters and functions. The objective of workload characterization is to derive a model that is able to show, capture, and reproduce the behavior of the workload and its most important features [3]. The number of stations is varied from *10* to *40*. Most applications use frames with size of 2 to the power of an integer value. Hence we vary the frame size from *128* to *1024* bytes.

Many simulation and theoretical studies of Ethernet assume a simple distribution for the arrival of packets. Poisson distribution is usually used. However, the work load in real networks is rarely Poisson distributed. Usually there are some bursts of heavy load that are divided by long time light traffic [2]. Hence we do not assume a particular distribution for the arrival of frames and set the *mean idle time* to be *0*, which represents a heavy and constant load state where every station always has data to transmit.

The detailed Π_{prob} results for Ethernet network are shown in Table 4.1.

Frame size (bytes)	Π_{prob}			
	10 stn	20 stn	30 stn	40 stn
128	9.2	15.9	22.3	28.1
256	9.3	17.1	22.9	30.6
512	9.4	17.8	25.6	33.7
1,024	9.5	18.4	27.0	35.3

Table 4.1: Π_{prob} for Ethernet

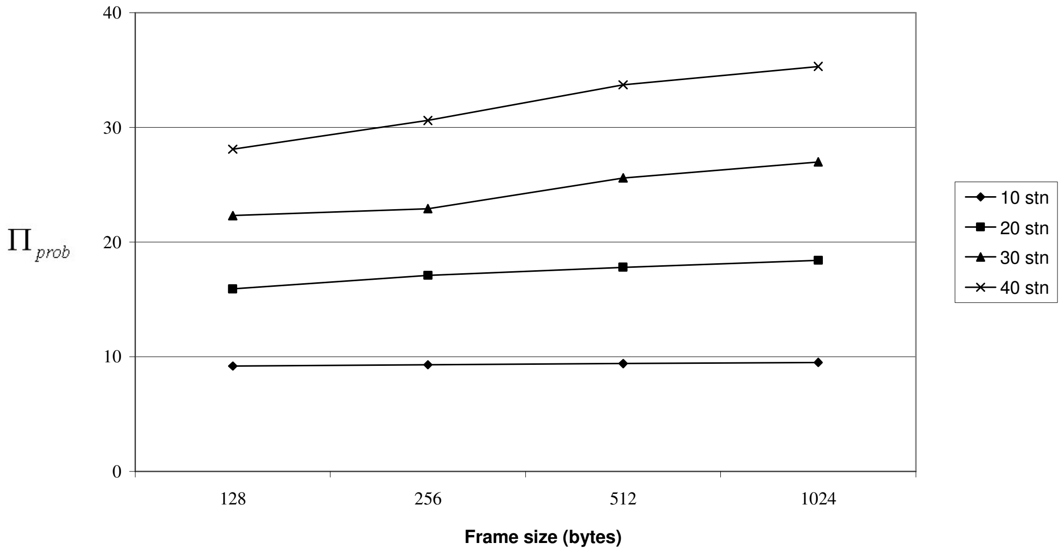


Figure 4.1: Π_{prob} for Ethernet

Figure 4.1 shows how Π_{prob} changes with number of stations and frame size. Π_{prob} increases when more stations exist in the system, for more stations create the potential for more concurrent stations in Ethernet network simulator. Π_{prob} also increases slightly with frame size. A frame with larger frame size spends more time passing through a station and thus reduce the service rate (μ) of stations. The service rate then influences the utilization of stations and increases the Π_{prob}.

4.1.2 Event Orderings

Events occurring at different physical times are executed chronologically in sequential simulation. Even two concurrent events in the physical system are processed sequentially in the simulation before parallel computers were introduced. Parallel simulation can relax this restriction and use other event orderings to generate correct simulation results, as long as it does not violate the event causality. We have already introduced how TSA measures Π_{ord} in Chapter 2. Now let us look at Π_{ord} results for Ethernet network simulation.

We use the same input parameters that we used while presenting inherent event parallelism in section 4.1.1. The number of stations is varied from 10 to 40 and frame size is varied from 128 bytes to 1024 bytes. The mean idle time between transmissions is set to be 0. The window size of *time interval event ordering* (TI) is set to be *0.002* ms. We estimate the window size to be the transmission delay between two neighbouring stations. If there are 25 stations and the cable length is 10 km, the transmission delay will be $\left(10km/25-1\right)/\left(2*10^8 m/s\right)$, approximately 0.002 ms. The simulation terminates when simulation time exceeds *100,000 ms*. The results of our experiment are based on the average of five replications of Ethernet network simulation. Figure 4.2 shows the detailed Π_{ord} results.

Parameter		Π_{ord}			
#Station	Frame size (bytes)	Partial event order (PAR)	Time interval event order (TI)	Time stamp event order (TS)	Total event order (TOT)
10	128	6.2	3.5	2.2	1.0
	256	6.4	3.6	2.3	1.0
	512	6.5	3.7	2.3	1.0
	1,024	6.8	3.9	2.4	1.0
20	128	12.5	7.3	4.6	1.0
	256	12.9	7.4	4.7	1.0
	512	13.4	7.7	4.9	1.0
	1,024	14.1	8.2	5.1	1.0
30	128	20.5	11.9	7.5	1.0
	256	20.7	11.9	7.5	1.0
	512	21.1	12.1	7.7	1.0
	1,024	22.1	12.8	8.1	1.0
40	128	27.9	16.2	10.4	1.0
	256	29.0	16.6	10.5	1.0
	512	29.5	16.9	10.7	1.0
	1,024	30.7	17.8	11.3	1.0

Table 4.2: Π_{ord} for Ethernet

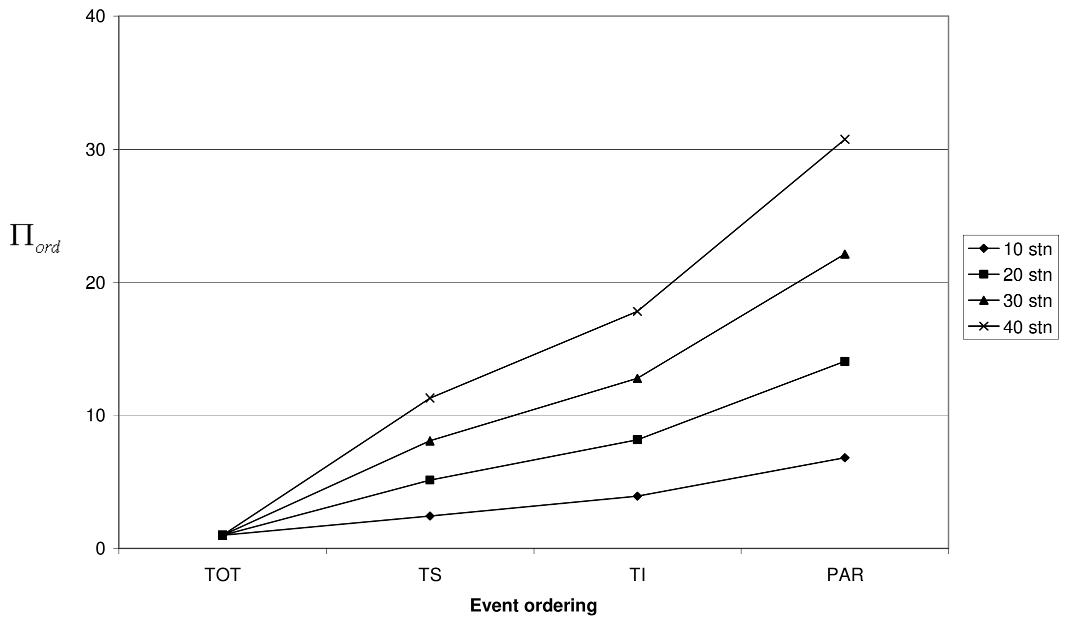


Figure 4.2: Π_{ord} changes with event orderings (frame size 1024 bytes)

Figure 4.2 demonstrates that the event orderings significantly influence Π_{ord}. Partial event ordering (PAR) achieves high parallelism while time stamp (TS) event ordering can exploit only little parallelism. Partial event ordering considers only the causal restriction between events while time interval event ordering and time stamp event ordering have additional ordering rules to restrict the number of concurrent events, thus limiting the event parallelism. A weaker event ordering exploits high parallelism in a closed system, which is consistent with the results by Teo et al [40].

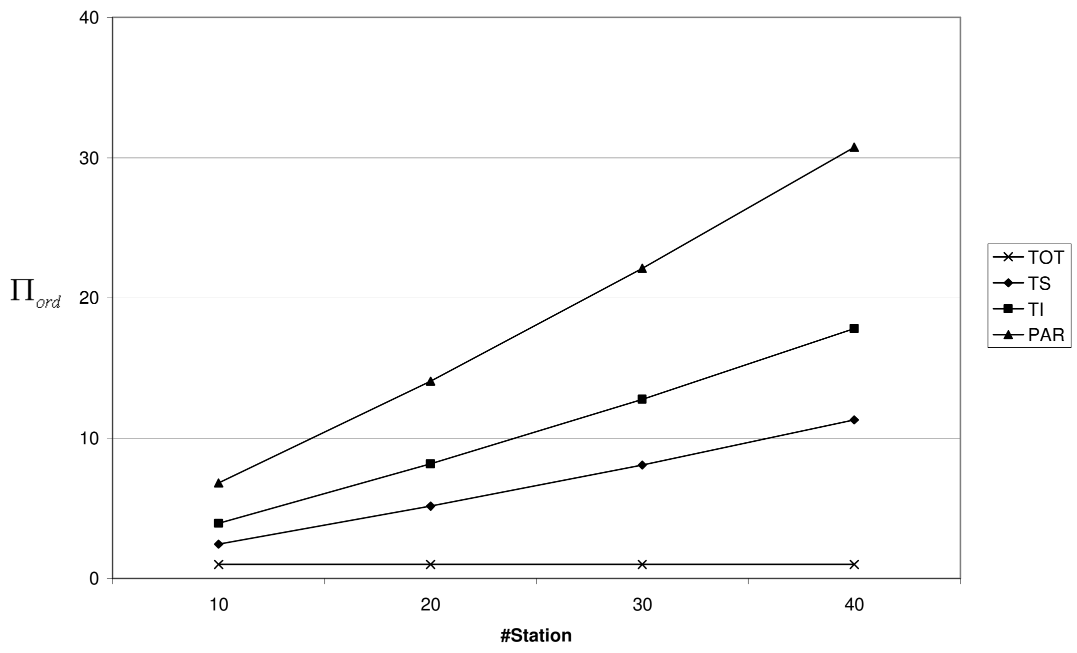


Figure 4.3: Π_{ord} changes with problem size (frame size 1024 bytes)

Π_{ord} increases almost linearly with number of stations as shown in Figure 4.3. More stations potentially allow more concurrent events and thereby give higher parallelism.

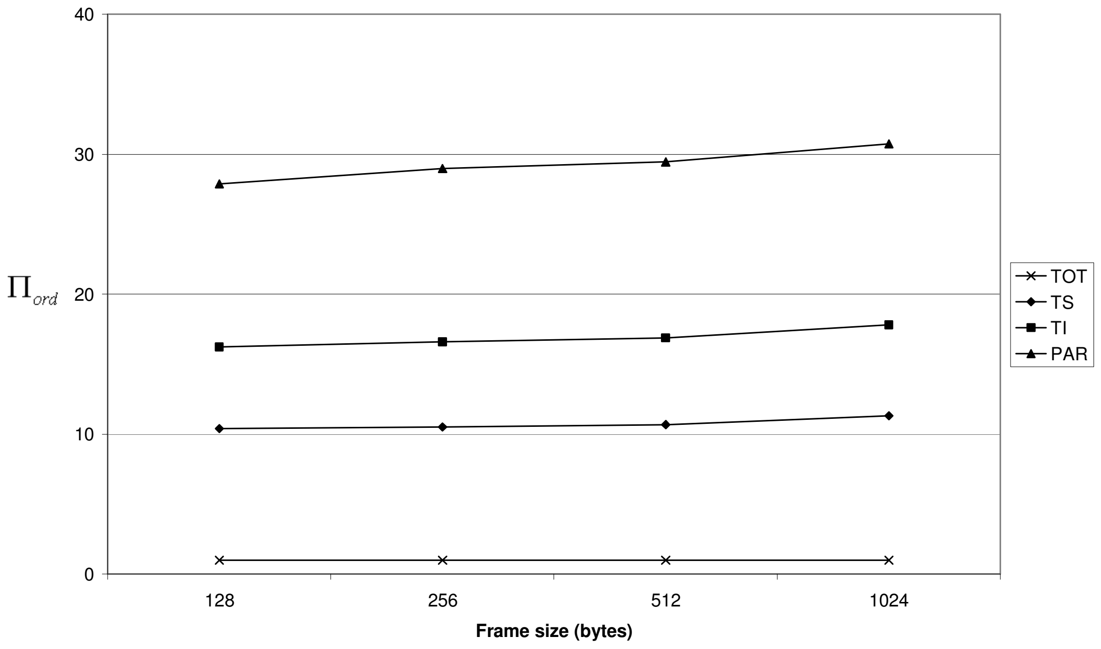


Figure 4.4: Π_{ord} changes with frame size (n=40)

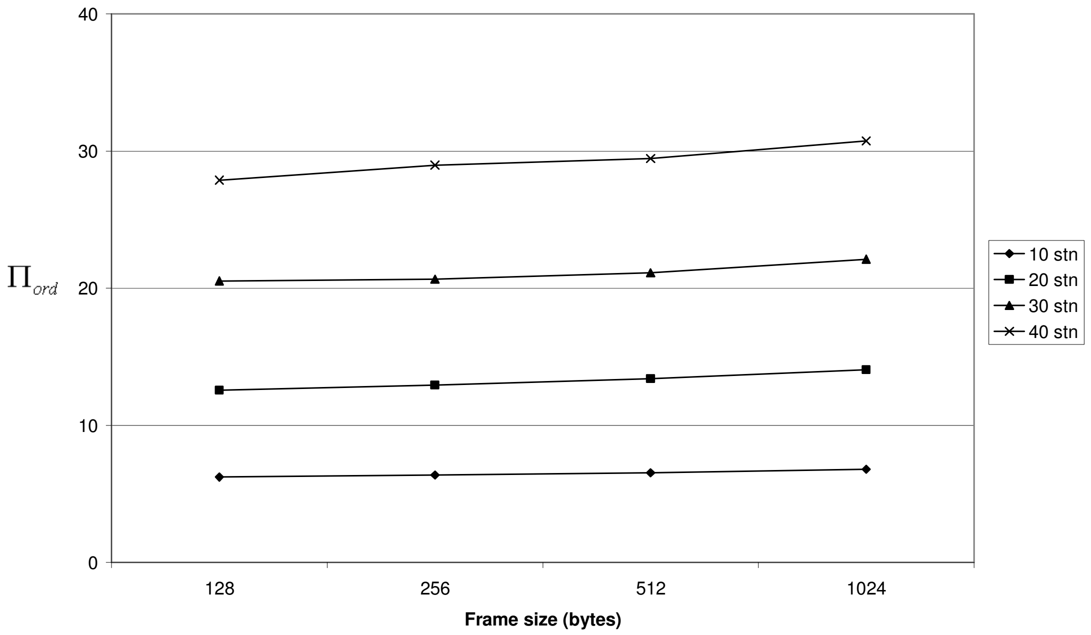


Figure 4.5: Π_{ord} changes with frame size (partial event order)

Figure 4.4 and 4.5 show that Π_{ord} increases slightly with the frame size. It can be interpreted by analyzing the event dependency in Ethernet network simulator. There are three dependences between events in Ethernet network simulator:

1) The dependency between events representing transmission (arrival or departure) between neighbouring stations. Since the cable length and the signal transmission speed are constant in our simulator, the dependency is determined by the number of stations.
2) The dependency between events representing a frame's external arrival and its departure from the source station. The time delay is the *frame transmission time*, which is determined by frame size when the LAN speed is fixed to 10Mbps.
3) The dependency between external arrival events and their possible antecedent events. Since the antecedent event of an external arrival event usually occurs at its previous transmission, the time delay for the dependency is mainly determined by the mean idle time.

Frame size can influence the time delay between a frame's external arrival and its departure from the source station (Category 2). The time delay is long for a large frame, therefore it is less likely for the departure event to block and wait for its antecedent event (the external arrival event). Therefore, a larger frame size gives slightly higher Π_{ord}.

4.1.3 Implementation

As illustrated in Chapter 2, effective event parallelism (Π_{sync}) is measured from actual simulation. It is measured as $\#Events \big/ \left(T/T_{unit}\right)$, where $\#Events$ is the number of all events in the simulated problem, T is the execution time (only events and null messages) of the LP which has the longest execution time compared to the others, and T_{unit} is the average event execution time (Eq. 2.1).

The Tembusu cluster (64 Intel PIII 1.4 GHz dual processors, each with 1 GB RAM, connected via a 1G Bps Myrinet switch) is used in our experiments. We map one LP per node in the cluster and terminate the simulation when the simulation time exceeds *100,000* ms. The results are listed at Table 4.3.

Parameters		#Events ($\times 10^6$)	#Null messages ($\times 10^6$)	T (s)	T_{unit} (ms)	Π_{sync}
#Station	Frame size (bytes)					
5	128	18.3	221.5	290.6	0.039	2.4
	256	9.9	164.8	208.3	0.041	2.0
	512	5.3	135.1	114.0	0.033	1.5
	1,024	3.3	123.3	95.4	0.037	1.3
10	128	29.7	936.4	419.4	0.041	2.9
	256	17.7	765.9	335.6	0.043	2.3
	512	11.0	660.2	216.2	0.043	2.2
	1,024	7.4	605.3	179.0	0.042	1.7
15	128	45.9	2215.8	545.5	0.044	3.7
	256	29.0	1997.0	397.2	0.043	3.1
	512	18.2	1582.2	304.8	0.043	2.6
	1,024	11.9	1452.6	271.2	0.042	1.9
20	128	88.6	4362.0	871.0	0.047	4.7
	256	43.9	3293.7	554.5	0.047	3.7
	512	27.6	2899.4	484.5	0.048	2.8
	1,024	18.1	2674.6	386.2	0.044	2.1

Table 4.3: Π_{sync} for Ethernet

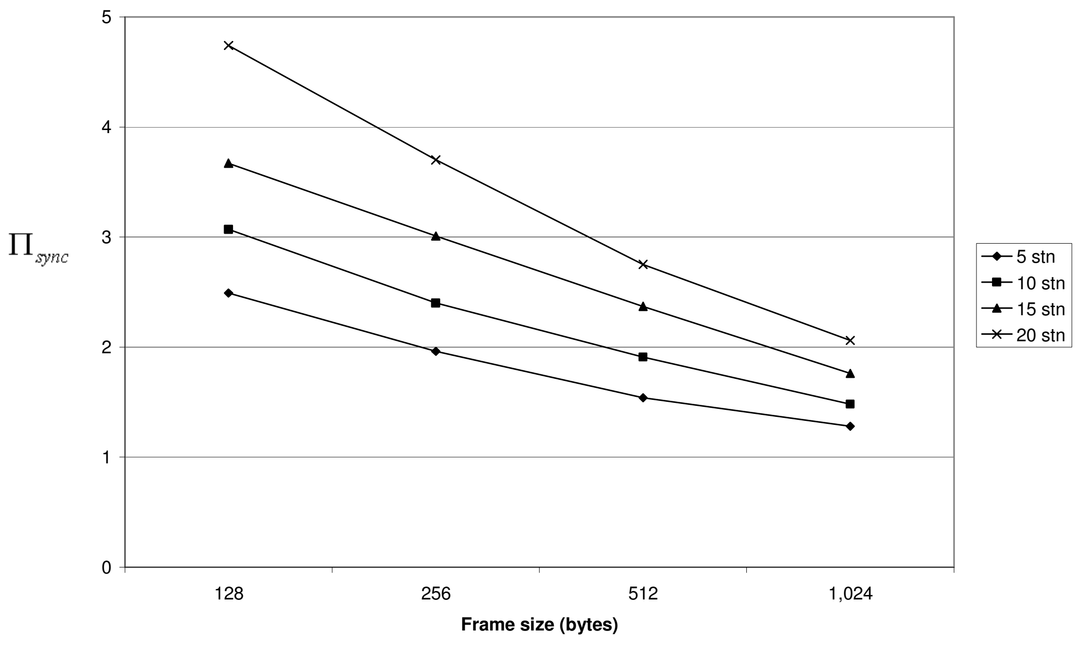


Figure 4.6: Π_{sync} for Ethernet

Figure 4.6 shows how Π_{sync} changes with frame size and the number of stations. Just like the inherent event parallelism and event ordering parallelism, more stations give higher effective event parallelism. There are potentially more concurrent events with larger number of LPs in the system. We also observe that larger frame size exploits less parallelism. Effective event parallelism is influenced by the ratio of events execution time to null messages execution time. The number of executed events will decrease when frame size is increased because the time a frame takes to transmit on the channel will increase. However, the lookahead of null messages (neighbour delay) will stay constant because the number of stations does not change. A large frame size increase the average time delay between two events, and hence increase the number of null messages. Therefore, the ratio of event execution time to null message execution time will decrease and thus effective event parallelism will decrease. Figure 4.7 compares the event execution time (T-event) and null message execution time (T-nullmsg) change with frame size (bytes). The number of stations is 10.

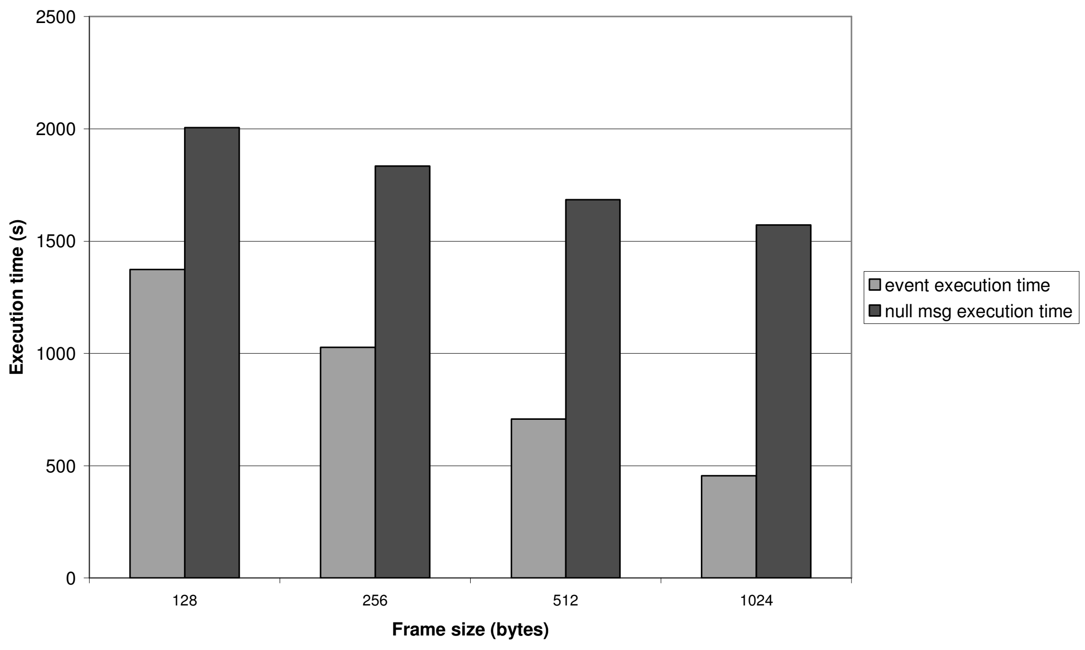


Figure 4.7: Event and null message execution time changes with frame size

4.1.4 Relationship between Different Parallelisms

We have measured the event parallelism from three steps: problem, event orderings and implementation. In this section, we establish the relationships between the parallelisms of the three steps.

#Station	Frame size (bytes)	Π_{prob}	Π_{ord}	Π_{sync}
10	128	9.2	6.2	2.9
	256	9.3	6.4	2.3
	512	9.4	6.5	2.2
	1,024	9.5	6.8	1.7
20	128	15.9	12.6	4.7
	256	17.1	12.9	3.7
	512	17.8	13.4	2.8
	1,024	18.4	14.1	2.1

Table 4.4: Comparison of three parallelisms in Ethernet simulation

Table 4.4 illustrates that all of the three parallelisms will increase with the number of stations.

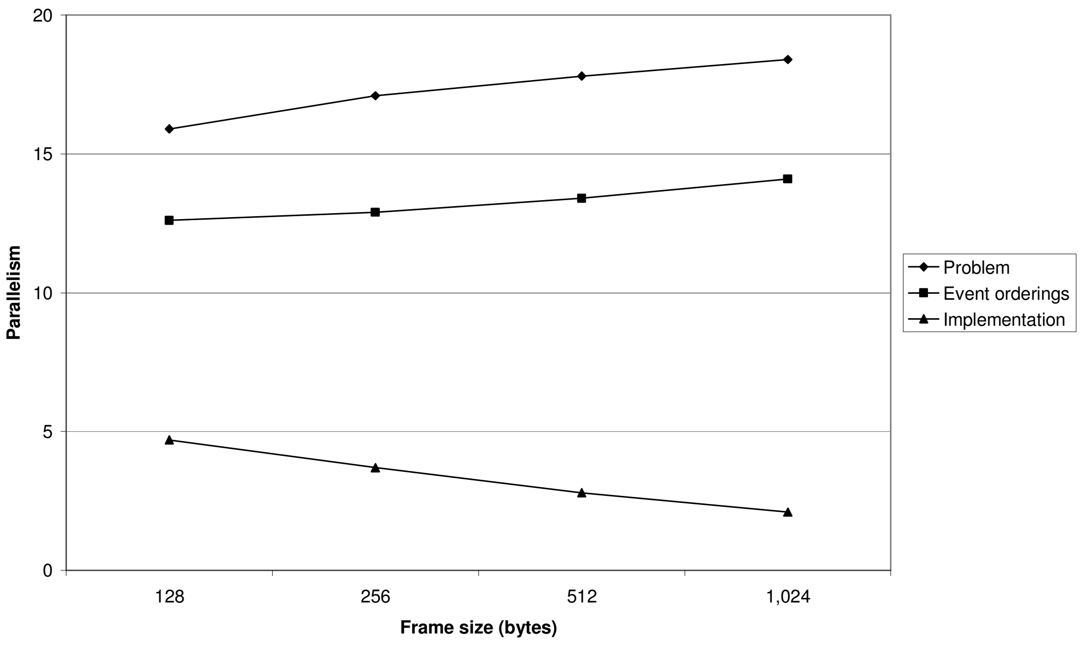


Figure 4.8: Relationships of different parallelisms (#station is 20)

Figure 4.8 illustrates the relationship of three parallelisms and the detailed results are presented in Table 4.4. We assume that the event ordering used in the null message synchronization protocol in SPaDES/Java is partial event ordering. We observe that Π_{prob} is higher than Π_{ord} and Π_{ord} is higher than Π_{sync}. Clearly, maintaining a certain event ordering decreases the event parallelism existing in the problem. In addition, when one particular synchronization protocol is used in a specific platform, it is implemented to maintain event causality according to certain event ordering rules. Hence the event parallelism will decrease due to the additional implementation overhead. The event parallelism after implementation (Π_{sync}) is much less than that to maintain event orderings (Π_{ord}). A large number of null messages deteriorate the performance of parallel Ethernet network simulator greatly.

We also observe that Π_{prob} and Π_{ord} increase with frame size but Π_{sync} decreases with frame size. The reasons for this result have been presented when the results were interpreted separately in previous sections.

4.2 Memory Requirement

We divided the memory required to support a simulator into three main components, namely, memory to model the states of the physical system (M_{prob}), memory required by the event list to schedule event execution based on the selected event ordering (M_{ord}), and additional memory to implement the synchronization protocol (M_{sync}).

TSA can measure the M_{prob} and M_{ord} by following particular event ordering rules. M_{sync} is implementation dependent, so we measure it from the actual simulation.

4.2.1 Problem

The station in Ethernet is a delay center where no events need to wait [14]. Two or more events can pass through the station simultaneously. Therefore, M_{prob} is zero for Ethernet simulation.

4.2.2 Event Orderings

M_{ord} results for Ethernet simulation are presented in Table 4.5.

Parameter		M_{ord}			
#Station	Frame size (bytes)	Partial event order (PAR)	Time interval event order (TI)	Time stamp event order (TS)	Total event order (TOT)
10	128	52	52	52	52
	256	52	52	52	52
	512	52	52	52	52
	1,024	52	52	52	52
20	128	112	112	112	112
	256	112	112	112	112
	512	112	112	112	112
	1,024	112	112	112	112
30	128	172	172	172	172
	256	172	172	172	172
	512	172	172	172	172
	1,024	172	172	172	172
40	128	232	232	232	232
	256	232	232	232	232
	512	232	232	232	232
	1,024	232	232	232	232

Table 4.5: M_{ord} for Ethernet

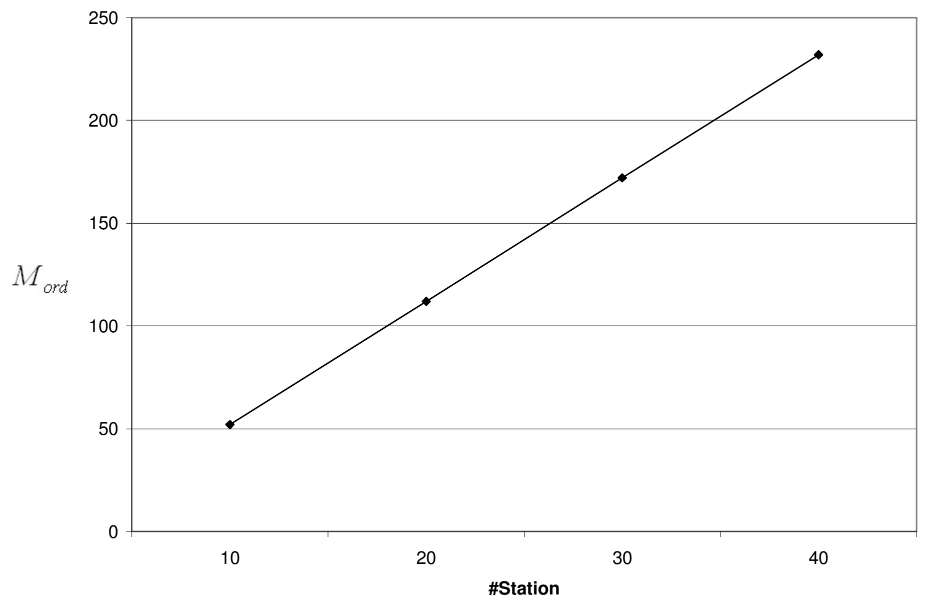


Figure 4.9: M_{ord} increases linearly with problem size

Figure 4.9 shows that the M_{ord} increases linearly $(6n-8)$ with problem size n. Surprisingly, M_{ord} is constant for different event orderings and frame sizes. A strong event ordering exploits more event ordering rules than necessary to follow the causality restriction in a closed system where the maximum number of jobs is known before implementation. The worst case scenario where M_{ord} achieves its maximum value occurs at the very beginning of Ethernet simulation, which will be explained in detail in the following paragraphs.

At the initial stage of Ethernet simulation, every frame is transmitted in two directions except the frame sent by a boundary station. Hence there are two frames waiting for transmission, i.e., FEL has two entries. Boundary stations have only one entry because they can only send frames in one direction. Therefore, M_{ord} is

$2n-2$ in the initial configuration as shown in Figure 4.10.

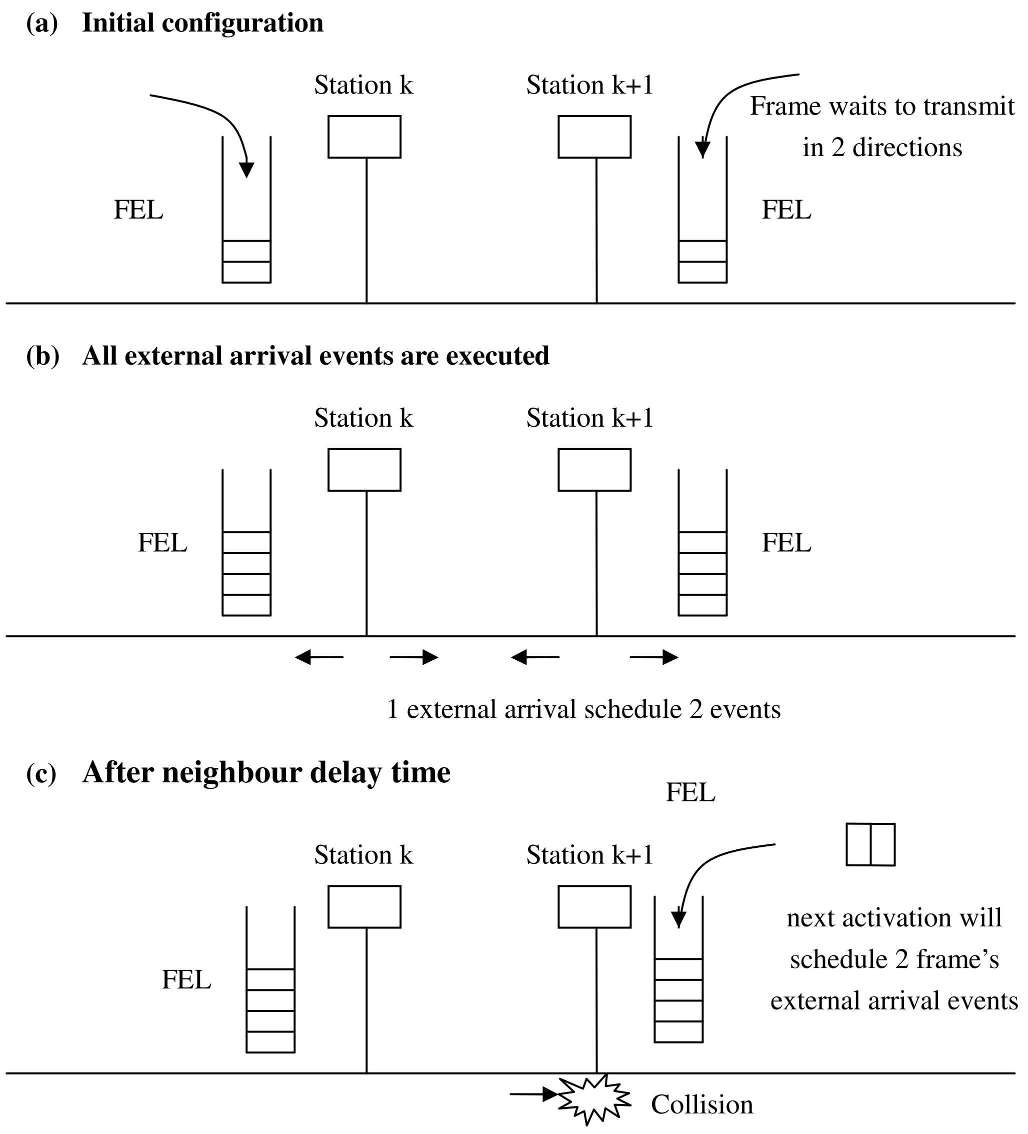

(a) Initial configuration
Station k
Station k+1
Frame waits to transmit in 2 directions
FEL
FEL
(b) All external arrival events are executed
Station k
Station k+1
FEL
FEL
1 external arrival schedule 2 events
(c) After neighbour delay time
FEL
Station k
Station k+1
next activation will schedule 2 frame's external arrival events
FEL
Collision

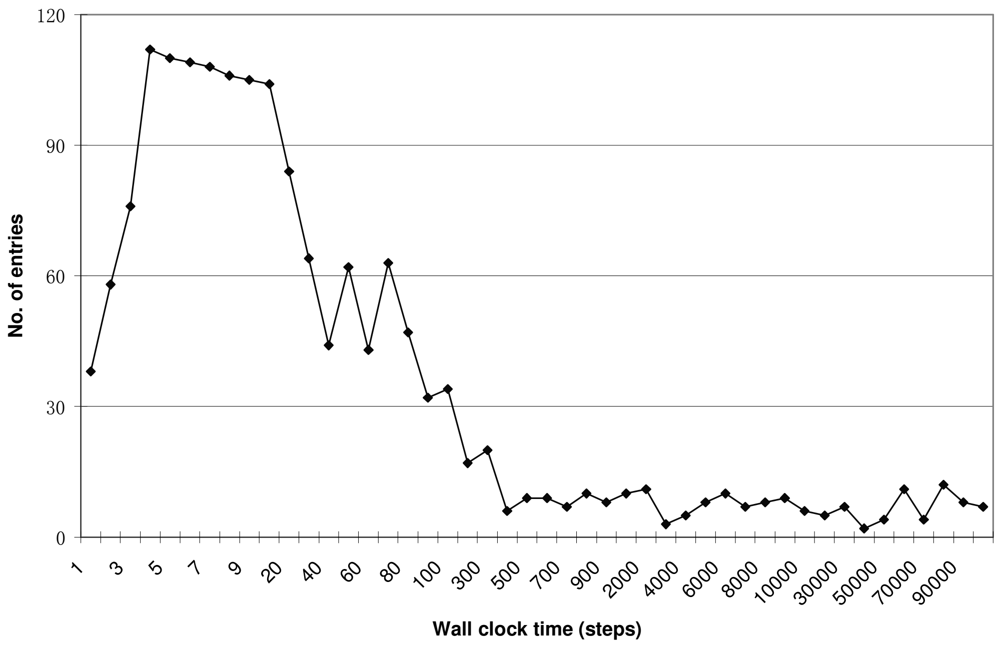
Figure 4.10: Worst case scenario for total event ordering

Worst case scenario occurs when every station wants to transmit frames at simulation time 0. The external arrival of a frame will schedule two events in FEL (refer to Table 3.2): the frame head's arrival at neighbouring station and the frame tail's departure from its source station. After every station finishes executing its external arrival event, there will be 4 entries in its FEL except boundary stations, which have two entries in their FELs. Hence M_{ord} is $4n-4$ after simulation time 0 as shown in Figure 4.10b.

After the initial execution of all external arrival events, there will be 2n-2 frames transmitted in the channel. No doubt all frames will collide with each other. In addition, all stations are equally distributed on the channel in our simulated Ethernet, so the delays between neighbouring stations are the same. After the delay time has passed, every frame will arrive at their neighbouring stations, which are all in the "Listen" state then. The collision arrival will schedule this station's next activity loop by scheduling two external arrival events in its FEL. It will also schedule its next arrival at the neighbor's neighbour, so the entries in this station's FEL will increase by 2. Hence there will be 6 entries in all internal stations after all the arrivals are executed at the same time. For a boundary station, there are only 2 entries in FEL, because it will absorb the arrival frames and only schedule one external arrival for next activity loop. Now M_{ord} is $6n-8$ as shown in Figure 4.10c.

After the worst scenario, M_{ord} begins to decrease because more and more frames arriving at a boundary station will be absorbed. The sum of entries in all FELs will decrease. In the next transmission, some stations will compete for the channel again with CSMA/CD protocol resolving collision and guaranteeing fairness.

CSMA/CD protocol gives priority to a station which has consecutive frames to send. Figure 4.11 shows the memory profile for partial event ordering over time. It shows that M_{ord} decreases soon after the aforementioned worst case. It remains at a small value around 10 after the system goes to a steady state. In the steady state one station usually dominates the channel and transmits a large amount of frames in a period of time. Other stations have to wait during that time interval. Then in other time intervals, there will be another station dominating the channel and transmitting large number of frames. This is the reason why M_{ord} is always small in the steady state.



**#station is 20, frame size is 1024 bytes*

Figure 4.11: Ethernet memory profile for partial event ordering

4.2.3 Implementation

M_{sync} is measured as the sum of maximum buffer size for null messages in all LPs. This is the additional memory used for implementation. We monitor the actual parallel simulator execution and use the simulation-based method to measure M_{sync}. The input configuration is the same that we measure Π_{sync}.

Frame size (bytes)	M_{sync}			
	5 stn	10 stn	15 stn	20 stn
128	579	2,433	5,092	10,036
256	674	2,834	7,371	12,532
512	1,127	4,284	8,549	15,389
1,024	1,310	6,029	14,205	28,127

Table 4.6: M_{sync} for Ethernet

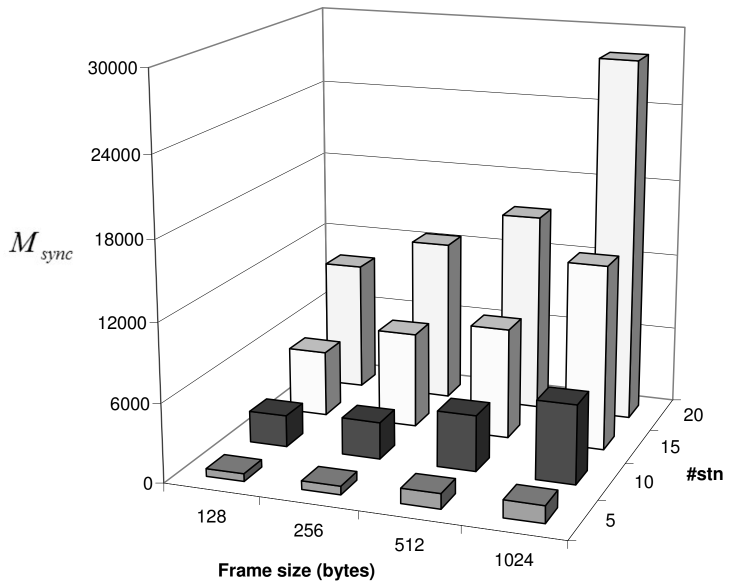


Figure 4.12: M_{sync} for Ethernet

Figure 4.12 illustrates how M_{sync} changes with number of stations and frame size. As we stated before, M_{sync} accounts for the memory to hold incoming null messages in an LP. Hence the maximum number of null messages between two executed events determines M_{sync}. Lookahead greatly influence the number of null messages in a parallel SPaDES/Java simulator. A large lookahead can decrease the number of null messages greatly. A small lookahead, on the other hand, can greatly increase the number of null messages. In Ethernet simulation, the lookahead is the time of transmission delay between two neighbouring stations. The cable length is constant, so more stations in Ethernet will decrease the transmission delay between two neighbouring stations. Therefore, a large number of stations mean a small lookahead. A small lookahead will increase the number of null messages. Hence M_{sync}, the memory used to hold null messages, will increase with number of stations.

Similarly, a large-size frame has a long time interval between its source departure event and its external arrival event. The number of null messages will increase when a long time interval exists between two events. Therefore, a large frame requires more M_{sync} than a small frame. Null messages deteriorate the performance seriously, which is shown in Table 4.7. The NMR (null message ratio) is defined as the ratio of the number of null messages to the total number of event messages and null messages.

#Station	Null message ratio (%)
5	97.7
10	98.3
15	98.8
20	99.0

**Frame size: 256 bytes, simulation time: 100,000 ms*

Table 4.7: Null message ratio changes with problem size

We can see that the number of null messages is already very high even when the problem size is small. A four-station Ethernet simulating for one second (simulation time) requires about 1,000,000 additional null messages to synchronize the parallel simulation. In addition, the NMR is also high. There are average 50 null messages for every event message.

Parameter		Memory usage		
#Station	Frame size (byte)	M_{prob}	M_{ord}	M_{sync}
10	128	0	52	2,433
	256	0	52	2,834
	512	0	52	4,284
	1,024	0	52	6,029
20	128	0	112	10,036
	256	0	112	12,532
	512	0	112	15,389
	1,024	0	112	28,127

Table 4.8: Memory requirement of Ethernet simulation

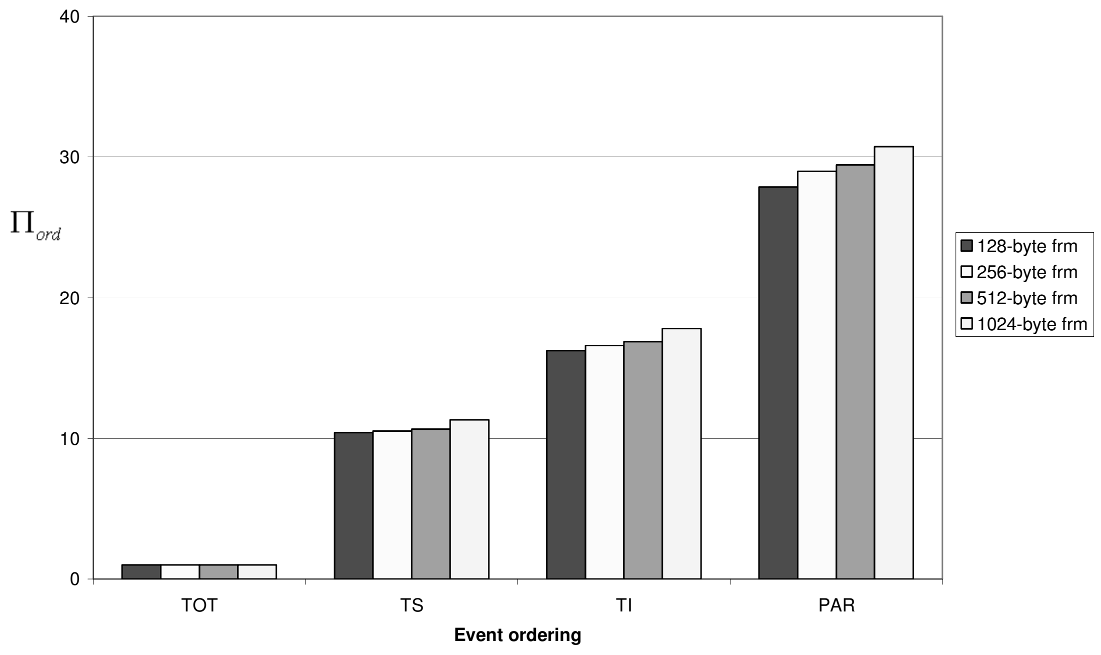
Detailed memory consumption of Ethernet is illustrated in Table 4.8. We count M_{ord} for conservative protocol from actual monitoring. M_{ord} is the same as other event orderings, i.e., 6n-8. We observe that much more memory is required for implementation than for maintaining event orderings.

4.3 Performance Tradeoff

This study successfully validated the methodology with the Ethernet application. Our experimental results are consistent with the existing results of queuing network benchmarks [27, 40]. This time space analysis is found to be applicable to both larger and realistic applications.

High speedup may require more memory usage in parallel simulation. Efficient implementation of a synchronization protocol exploits a higher parallelism while keeping the required memory under a certain level. Therefore, it is important to analyze the time-space tradeoff for performance tuning. In the Ethernet network simulator the event parallelism increases with the number of stations either in event ordering level (Π_{ord}) or in implementation level (Π_{sync}). However, by increasing the number of stations in the system, more memory is expended. Thus, there is a tradeoff between event parallelism and memory requirements with number of stations either in event ordering level or in implementation level.

If only event ordering is considered, Figure 4.13 shows that a weaker event ordering exploits higher event parallelism (Π_{ord}) without increasing the amount of memory requirement (M_{ord}) in Ethernet network simulator. The number of nodes is 40. A stricter event ordering is believed to impose more event ordering rules than necessary to follow causality restriction. Therefore, the parallelism gain can be achieved only by relaxing the event ordering rules.



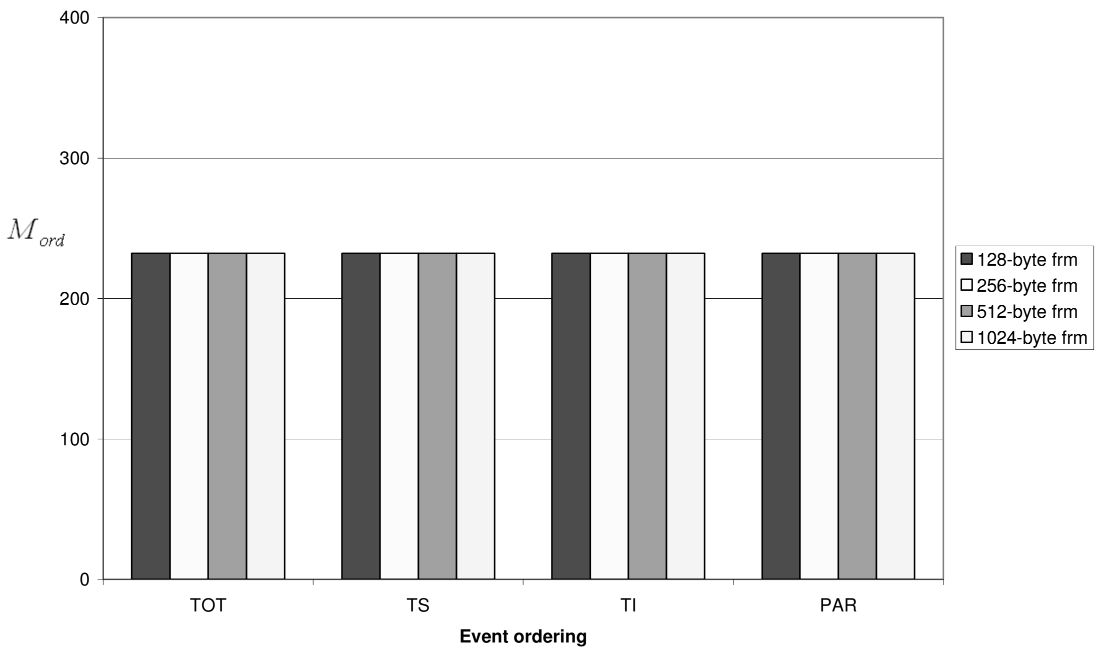


Figure 4.13: Time (Π_{ord}) and space (M_{ord}) tradeoff

In implementation level, Figure 4.6 and Figure 4.12 show that small frame size can provide a good balance between effective event parallelism (Π_{sync}) and memory used for synchronization (M_{sync}).

4.4 Summary

This chapter presented and analyzed the performance results of Ethernet network simulator in detail. The event parallelism and memory usage were analyzed from three levels: problem, event orderings and implementation. The inherent event parallelism was analyzed with an analytical method. TSA was used to measure event parallelism and memory requirement due to event orderings. We then parallelized the Ethernet simulation using SPaDES/Java to measure the performance in implementation level. We also discussed the relationships between performances of these three levels. Lastly, performance tradeoff, which can be used for performance tuning, is analyzed.

Chapter 5

Conclusion

Teo et al. have developed a formal methodology to evaluate event parallelism and memory requirement in parallel simulation before implementation. The advantage of this methodology is that one can predict parallelism and memory consumption before much effort is expended to parallelize a simulator. If we find that a simulator has little parallelism, no efforts need to be wasted to implement it. In addition every computer has limited capacity, so the memory consumption is also an important issue we should address in parallel simulation. Processors with at least the upper bound of the memory can guarantee that the parallel program can be executed.

Simulation protocol adheres to a certain event ordering to produce the simulation results correctly. Different event orderings produce different degrees of parallelism. In addition, to maintain a particular event ordering, one needs to save some pending events in its event list. Various amounts of memory may be required for different event orderings. Four simulation event orderings were formally defined with partial order set theory in the methodology. They are *partial event ordering*, *time interval event ordering*, *time stamp event ordering* and *total event ordering*.

This methodology has previously been validated using limited queuing network benchmarks.

In this thesis, we used Ethernet as an application to further validate and assess the application of this methodology. Ethernet is a large and complicated system. Usually there are hundreds of computers attached to the channel. We therefore expect much parallelism in the system. It is valuable to evaluate such a system before it is actually implemented.

The conceptual model of our Ethernet simulator is the same as one existing Ethernet model by CSIM, which can not only simplify the developing process but also reduce the validation effort. SPaDES/Java was used to develop both the sequential and parallel simulator.

After developing and validating the Ethernet simulator, we instrumented the simulator to get the detail information of event generated from the simulator. A time space analyzer tool was used to analyze these information to get the event parallelism and memory consumption results.

Our experimental results reveal that a weaker event ordering exploits more event parallelism without increasing memory usage, which is consistent with the previous results of benchmark studies. We observed that in the Ethernet network simulator the upper bound on memory to maintain event orderings is $6n-8$, where n is the number of stations. Therefore, this study successfully validates the time space analysis methodology. We studied in detail the relationship between event orderings and the performance of parallel simulations. Performance tradeoff analysis can be used for tuning the performance of parallel simulations.

Apart from assessing the cost of event orderings, we also used this methodology to analyze the performance of a simulation problem and the overhead of implementation. An analytical method is used to study the event parallelism inherent in the problem. To study the cost of implementation, we analyzed the conservative null message simulation protocol in SPaDES/Java and observed that much more memory is required to support synchronization than for maintaining event orderings. The relationship among performance results of these levels is also discussed in this thesis.

However, there are also some deficiencies in this research. One disadvantage in using Ethernet is its unique implementation. It was modeled as a closed system and had no queue [45]. This has two implications. First, we could not evaluate M_{prob}. At the design stage of this study, we chose Ethernet because it is one complex real-world application. In addition, Watkins' existing model can simplify the development and validation effort. Ethernet simulator failed to validate Mprob because Ethernet was modeled as a closed system and had no queue [45]. The performance (event parallelism and memory requirement) at the problem level has been well studied in previous studies [40, 42]. Second, the dependencies between events become more complicated if there is no queue to hold the incoming events.

Therefore, we needed to expend much effort to maintain these dependencies. The arrival and departure events of a frame in Ethernet also have no direct dependencies. They are both pre-determined when the frame enters into the Ethernet system. If the frame is successfully transmitted, the time interval between the two events is determined by frame size and LAN speed if the frame is not corrupted. Otherwise, we cannot determine the time interval.

Another deficiency is that the protocol used in parallel SPaDES/Java is inefficient due to high overhead of null messages. Optimization of the conservative protocol in SPaDES/Java is required.

Ethernet simulator is used in this study to validate the methodology and parameters of the TSA. It failed to validate Mprob because there is no internal queue presented in Ethernet network. Mprob has been well studied in previous studies [40, 42]. It is found that Mprob is dependent on the characteristic of problem, such as number of service centers and traffic intensity.

References

[1] Bain W.L. and Scott D.S., "An Algorithm for Time Synchronization in Distributed Discrete Event Simulation", Proc. Of the SCS Multi-conference on Distributed Simulation, 19, 3, pp. 30-33, February 1988.

[2] Boggs D.R., Mogul J.C. and Kent C.A., "Measured Capacity of an Ethernet: Myths and Reality" Tech. Rep. 88/4, Digital Western Research Laboratory, April 1988.

[3] Calzarossa M., Massari L. and Tessera D., "Workload Characterization Issues and Methodologies", Performance Evaluation, pp. 459-484, 2000.

[4] Cavitt D.B., Overstreet C.M. and Maly K.J., "A Performance Analysis Model for Distributed Simulations", Winter Simulation Conference, pp. 629-636, 1996.

[5] Chandy K.M. and Misra J., "Distributed Simulations: A Case Study in Design and Verification of Distributed Programs", IEEE Trans. on Software and Engineering. SE-5, 5, pp. 440-452, Sep. 1979.

[6] de Carvalho Klingelfus A.L. and Godoy W. Jr., "Mathematical modeling, performance analysis and simulation of current Ethernet computer networks", 5th IEEE International Conference on High Speed Networks and Multimedia Communications, pp. 380 -382, 2002.

[7] Dickens P.M. and Reynolds P.F. Jr., "A Performance Model for Parallel Simulation", Proceedings of Winter Simulation Conference, pp. 618-626, Dec. 1991.

[8] Felderman R. and Kleinrock L., "An Upper Bound on the Improvement of Asynchronous Versus Synchronous Distributed Processing", Distributed Simulation 1990. Society for Computer Simulation, pp. 131-136, January 1990.

[9] Floyd S. and Paxson V., "Difficulties in Simulating the Internet", IEEE/ACM Transactions on Networking, Vol.9, No.4, pp. 392-403, August, 2001.

[10] Forrester J.W., “Industrial Dynamics”, MIT Press, Cambridge, Massachusetts, 1961.

[11] Fujimoto R.M., “Parallel Discrete Event Simulation”, Communication of the ACM, 33, 10, pp. 31-53, 1990.

[12] Fujimoto R.M. "Parallel Discrete Event Simulation: Will the Field Survive?", ORSA Journal on Computing (feature article), Vol. 5, No. 3, pp. 213-230, summer 1993.

[13] Fujimoto R.M., “Parallel and Distributed Simulation Systems”, John Wiley & Sons, Inc., 2000.

[14] Jain R., “The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling”, New York, Wiley, 1991

[15] Jefferson D.R., “Virtual Time”, ACM Trans. Prog. Lang. and Syst. 7, 3, pp. 404-425, July 1985.

[16] Jefferson D.R., “Virtual Time II: Storage Management in Distributed Simulation”, Proceedings 9th annual ACM symposium on Principles of Distributed Computation, pp. 75-90, 1990.

[17] Lamport L., “Time, Clocks, and the Ordering of Events in a Distributed System”, Communication ACM, 21, 7 (July), pp. 558-565, 1978.

[18] Leon-Garcia A. and Widjaja I., “Communication Networks, Fundamental Concepts and Key Structures”, McGraw Hill, 2000.

[19] Lim C.C., Low Y.H., Gan B.P., Jain S., Cai W., Hsu W.J. and Huang S.Y., “Performance Prediction Tools for Parallel Discrete-Event Simulation”. Workshop on Parallel and Distributed Simulation, pp. 148-155, 1999.

[20] Lin Y-B. and Lazowska E.D., “Reducing the State Overhead for Time Warp Parallel Simulation”, Technical Report 90-02-03, Dept. of Computer Science and Engineering, University of Washington, Seattle, Washington, Feb. 1990.

[21] Lin Y-B. and Preiss B., “Optimal Memory Management for Time Warp Parallel Simulation”, ACM Trans. on Modeling and Computer Simulation, 1, 4, pp. 283-307, October 1991.

[22] Lin Y-B. and Fishwick P.A., “Asynchronous parallel discrete event simulation”, IEEE Transactions on Systems, Man and Cybernetics, 26(4):397-412, 1996.

[23] Marin M., "Towards Automated Performance Prediction in Bulk-Synchronous Parallel Discrete-Event Simulation", Proceedings of the XIX International Conference of the Chilean Computer Science Society, pp. 112-118, 1999.

[24] Metcalfe R.M. and Boggs D.R., "Ethernet: Distributed Packet Switching for Local Computer Networks", Comm, ACM, pp. 395-404, 1976.

[25] Nicol D. and Fujimoto R.M., “Parallel Simulation Today”. Annals Oper. Res. 53, pp. 249-286, 1994.

[26] Noble B.L., Peterson G.D. and Chamberlain R.D., “Performance of Synchronous Parallel Discrete-Event Simulation”, Proc. of 28th Hawaii Int'l Conf. on System Sciences, Vol. II, pp. 185-186, January 1995.

[27] 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 (DS-RT 2002), pp. 77-84, IEEE Computer Society Press, Fort Worth, Texas, USA, October 2002.

[28] Preiss B.R., MacIntye I.D. and Loucks W.M., “On the Trade-off between Time and Space in Optimistic Parallel Discrete-Event Simulation”, Proc. 6th Workshop on Parallel and Distributed Simulation, pp. 33-42, Jan. 1992.

[29] Rawling M., Francis R. and Abramson D., “Potential Performance of Parallel Conservative Simulation of VLSI Circuits and Systems”, Proceedings of 25th Annual Simulation Symposium, pp. 71-81, Apr. 1992.

[30] Robert G. Sargent, “Simulation Model Verification and Validation”, Proceeding of the 1991 Winter Simulation Conference, 1991.

[31] Robinson S., “Simulation Model Verification and Validation: Increasing The Users' Confidence”, Proceedings of the Winter Simulation Conference, pp. 53-59, Dec. 1997.

[32] Ronngren R. and Liljenstam M., “On Event Ordering in Parallel Discrete Event Simulation”, 13th Workshop on Parallel and Distributed Simulation May 01 - 04, pp. 38-45, Atlanta, Georgia, 1999.

[33] Saunders S., “Data Communications Gigabit Ethernet Handbook”, McGraw-Hill, 1998

[34] Tanenbaum A.S., “Computer Networks”, Third Edition, Prentice Hall, 1996.

[35] Tay S.C., Teo Y.M. and Ayani R., "Performance Analysis of Time Warp Simulation with Cascading Rollbacks", Proc. of 12th Workshop on Parallel and Distributed Simulation (PADS'98), pp.30-37, IEEE Computer Society Press, Canada, May 1998.

[36] Teo Y.M. and Tay S.C., "Performance Evaluation of a Parallel Simulation Environment", Proceedings of the 32nd Annual Simulation Symposium, pp. 86-93, IEEE Computer Society Press, San Diego, USA, April 1999.

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

[38] Teo Y.M. and Tay S.C., "Performance and Granularity Control in the SPaDES Parallel Simulation System", Proceedings of the 4th International Symposium on Parallel Architectures, Algorithms, and Networks (I-SPAN), pp. 94-99, IEEE Computer Society Press, Perth, Australia, June 1999.

[39] Teo Y.M. and Onggo B.S.S., “A Methodology for Space Analysis of Discrete-event Simulation”, Technical Report, Department of Computer Science, National University of Singapore, June 2001.

[40] Teo Y.M., Onggo B.S.S. and Tay S.C., “Effect of Event Orderings on Memory Requirement in Parallel Simulation”, Proc. of the 9th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems, pp. 41-48, 2001.

[41] 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

[42] 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., IEEE Computer Society Press, Washington, USA, May 2002.

[43] Wang H., Teo Y.M. and Tay S.C., "An Analytic Method for Predicting Simulation Parallelism", Proceedings of the 33rd Annual Simulation Symposium, pp. 211-218, IEEE Computer Society Press, Washington D.C., USA, April 2000.

[44] Wang J. and Keshav S., "Efficient and Accurate Ethernet Simulation", Proceedings of the 24th Conference on Local Computer Networks (LCN `99), Oct. 1999.

[45] Watkins K., "Discrete Event Simulation in C", McGraw-Hill, 1992.

[46] Wieland F., Som T., Reiher P., Wedel J. and Jefferson D., "A Critical Path Tool for Parallel Simulation Performance Optimization", Proceedings of the 25th Hawaii International Conference on System Sciences, pp. 196-206, Jan. 1992.

[47] Willinger W., Taqqu M.S., Sherman R. and Wilson D.V., "Self-Similarity Through High-Variability: Statistical Analysis of Ethernet LAN Traffic at the Source Level", ACM SIGCOMM'95. Computer Communication Review, 25:100-113, 1995.

[48] 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.

[49] Yamaguchi Y., Osawa N. and Yuba T., "3D Animation Based on Dynamic System Modeling for Parallel Discrete Event Simulation Systems", 9th European Simulation Symposium (ESS'97), pp. 259-263, Oct. 1997.

[50] Young C.H. and Wilsey P.A., "Optimistic Fossil Collection for Time Warp Simulation", Proceedings of the 29th Hawaii International Conference on System Sciences, Vol. 1, pp. 364-372, Jan. 1996.

Appendix A: Run-time Report of Pipeline Simulation

```
Schedule event: 0 with timestamp: 1.0

Starting SPaDES/Java sequential simulation ....

Message 0 arrive the 0 LP. At timestamp: 1.0
Schedule event: 1 with timestamp: 8.0
Service time: 8.0
Message 0 leaves the 0 LP. At timestamp: 9.0

Message 1 arrive the 0 LP. At timestamp: 8.0
Schedule event: 2 with timestamp: 15.0
!!!!!! delayed
Service time: 8.0
Begin service at time: 9.0
Message 1 leaves the 0 LP. At timestamp: 17.0

Message 0 arrive the 1 LP. At timestamp: 9.0
Service time: 8.0
Message 0 leaves the 1 LP. At timestamp: 17.0

Message 2 arrive the 0 LP. At timestamp: 15.0
!!!!!! delayed
Service time: 8.0
Begin service at time: 17.0
Message 2 leaves the 0 LP. At timestamp: 25.0

Message 1 arrive the 1 LP. At timestamp: 17.0
Service time: 8.0
Message 1 leaves the 1 LP. At timestamp: 25.0

Message 2 arrive the 1 LP. At timestamp: 25.0
Service time: 8.0
Message 2 leaves the 1 LP. At timestamp: 33.0

Elapsed time: 151.0 milliseconds
Current simulation ended
```

Appendix B: Events in Pipeline Simulation

**EXTARR=External Arrival, INTARR=Internal Arrival, EXTDEP=External Departure, INTDEP=Internal Departure.*

e_1:

```
****** Message 0 arrives LP 0 at timestamp: 1.0 ******
The Event Type: EXTARR
The Event ID: 0
The Event lp: 0
The ante Event Type: EXTARR
The ante Event ID: -1
The ante Event lp: -1
The Event startTime: 1.0
The Event anteTime: 0.0
The Event nextLp: 0
************************************************
```

e_2:

```
****** Message 1 arrives LP 0 at timestamp: 8.0 ******
The Event Type: EXTARR
The Event ID: 1
The Event lp: 0
The ante Event Type: EXTARR
The ante Event ID: 0
The ante Event lp: 0
The Event startTime: 8.0
The Event anteTime: 1.0
The Event nextLp: 0
************************************************
```

e_3:

```
****** Message 0 leaves LP 0 at timestamp: 9.0 ******
The Event Type: INTDEP
The Event ID: 0
The Event lp: 0
The ante Event Type: EXTARR
The ante Event ID: 0
The ante Event lp: 0
The Event startTime: 9.0
The Event anteTime: 1.0
The Event nextLp: 1
************************************************
```

e_4 :

```
****** Message 0 arrives LP 1 at timestamp: 9.0 ******
The Event Type: INTARR
The Event ID: 0
The Event lp: 1
The ante Event Type: INTDEP
The ante Event ID: 0
The ante Event lp: 0
The Event startTime: 9.0
The Event anteTime: 9.0
The Event nextLp: 1
**************************************************
```

e_5 :

```
****** Message 2 arrives LP 0 at timestamp: 15.0 ******
The Event Type: EXTARR
The Event ID: 2
The Event lp: 0
The ante Event Type: EXTARR
The ante Event ID: 1
The ante Event lp: 0
The Event startTime: 15.0
The Event anteTime: 8.0
The Event nextLp: 0
**************************************************
```

e_6 :

```
****** Message 0 leaves LP 1 at timestamp: 17.0 ******
The Event Type: EXTDEP
The Event ID: 0
The Event lp: 1
The ante Event Type: INTARR
The ante Event ID: 0
The ante Event lp: 1
The Event startTime: 17.0
The Event anteTime: 9.0
The Event nextLp: 1
**************************************************
```

e_7 :

```
****** Message 1 leaves LP 0 at timestamp: 17.0 ******
The Event Type: INTDEP
The Event ID: 1
The Event lp: 0
The ante Event Type: INTDEP
The ante Event ID: 0
The ante Event lp: 0
The Event startTime: 17.0
The Event anteTime: 9.0
The Event nextLp: 1
************************************************
```

e_8:

```
****** Message 1 arrives LP 1 at timestamp: 17.0 ******
The Event Type: INTARR
The Event ID: 1
The Event lp: 1
The ante Event Type: INTDEP
The ante Event ID: 1
The ante Event lp: 0
The Event startTime: 17.0
The Event anteTime: 17.0
The Event nextLp: 1
************************************************
```

e_9:

```
****** Message 3 arrives LP 3 at timestamp: 22.0 ******
The Event Type: EXTARR
The Event ID: 3
The Event lp: 0
The ante Event Type: EXTARR
The ante Event ID: 2
The ante Event lp: 0
The Event startTime: 22.0
The Event anteTime: 15.0
The Event nextLp: 0
************************************************
```

e_{10}:

```
****** Message 1 leaves LP 1 at timestamp: 25.0 ******
The Event Type: EXTDEP
The Event ID: 1
The Event lp: 1
The ante Event Type: INTARR
The ante Event ID: 1
The ante Event lp: 1
The Event startTime: 25.0
The Event anteTime: 17.0
The Event nextLp: 1
************************************************
```

e_{11}:

```
****** Message 2 leaves LP 0 at timestamp: 25.0 ******
The Event Type: INTDEP
The Event ID: 2
The Event lp: 0
The ante Event Type: INTDEP
The ante Event ID: 1
The ante Event lp: 0
The Event startTime: 25.0
The Event anteTime: 17.0
The Event nextLp: 1
************************************************
```

e_{12}:

```
****** Message 2 arrives LP 1 at timestamp: 25.0 ******
The Event Type: INTARR
The Event ID: 2
The Event lp: 1
The ante Event Type: INTDEP
The ante Event ID: 2
The ante Event lp: 0
The Event startTime: 25.0
The Event anteTime: 25.0
The Event nextLp: 1
************************************************
```

e_{13}:

```
****** Message 3 leaves LP 0 at timestamp: 33.0 ******
The Event Type: INTDEP
The Event ID: 3
The Event lp: 0
The ante Event Type: INTDEP
The ante Event ID: 2
The ante Event lp: 0
The Event startTime: 33.0
The Event anteTime: 25.0
The Event nextLp: 1
************************************************
```

e_{14}:

```
****** Message 3 arrives LP 1 at timestamp: 33.0 ******
The Event Type: INTARR
The Event ID: 3
The Event lp: 1
The ante Event Type: INTDEP
The ante Event ID: 3
The ante Event lp: 0
The Event startTime: 33.0
The Event anteTime: 33.0
The Event nextLp: 1
************************************************
```

e_{15}:

```
****** Message 2 leaves LP 1 at timestamp: 33.0 ******
The Event Type: EXTDEP
The Event ID: 2
The Event lp: 1
The ante Event Type: INTARR
The ante Event ID: 2
The ante Event lp: 1
The Event startTime: 33.0
The Event anteTime: 25.0
The Event nextLp: 1
************************************************
```

e_{16}:

```
****** Message 3 leaves LP 1 at timestamp: 41.0 ******
The Event Type: EXTDEP
The Event ID: 3
The Event lp: 1
The ante Event Type: EXTDEP
The ante Event ID: 2
The ante Event lp: 1
The Event startTime: 41.0
The Event anteTime: 33.0
The Event nextLp: 1
************************************************
```

Appendix C: Pseudo Code of Ethernet Simulation in SPaDES/Java

```
package Ethernet;
//Import SPaDES/Java library and other packages
import spades_Java.*;

//EthernetKernel.java
// Executive instance
public class EthernetKernel extends Executive{
      :
      Resource Service[];
      //station process
      Station station[];
      :
      //define parameters for Ethernet
      double LAN_Speed;             /* LAN speed in bps */
      double cable_length;          /* Cable length in meter */
      int num_stations;             /* Number of users in the LAN */
      int frame_size;               /* Frame size in bytes */
      double duration;              /* Simulation duration in ms */
      :
      public void init(){
           :
           //Initialize Ethernet parameters
           LAN_Speed = 10000000;
           cable_length = 10000;
           num_stations = 20;
           frame_size = 1024;
           idle_time = 0;
           duration = 100000;
           :
           <<Initialize resource>>
           :
           for (int i=0, i<num_stations, i++){
                 :
                 station[j]=new Station("Station "+j,this);
                 <<Initialize station[i]>>
                mapProcess(station[j],Service[j]);
                 activate(station[j], 0);
                 :
           }
      }
       public static void main(String[] args){
             EthernetKernel ek = new EthernetKernel();
             ek.initialize(args.length, args);
             ek.startSimulation(duration);
             <<Print simulation results>>
       }
}

// Station.java
// Models a station that transmits frames to channel in Ethernet
class Station extends SProcess {
       EthernetKernel ek;
       int successful;          /* Number of frames sent */
       int number_retrans;      /* Number of retransmission so far */
       Frame[] end;             /* Frames transmitted out */
       :
       <<Initialize other Station parameters>>
       :
       public Station(String n, EthernetKernet h){
             super();
             name = toString().toCharArray();
```

```
        ek = h;
        :
 }
 :
 public Boolean channel_clear(){
       if (<<No frames passing left>> and <<no frame passing righ>>)
             return true;
       else
             return false;
 }
:
 void retransmit(Station this_station){
       /*Have maximum number of attempts been made?*/
            if (<<Number of retransmission is larger or equal to 16>>){
                  <<Set phase to Finish>>
                  wait(noise_burst*1.005);
            }else{
                  <<Set phase to Contention>>
                  /*Wait for a random and restransmit*/
                  wait(retrans_time(this_station));
            }
      }
      :
      /*Calculate time delay after a collision - binary exponential*/
      float retrans_time(int this_station){
            int   t;
            int    num;
            float  maximum;
            number_retrans++;
            num = number_trans;
            if (num>10){
                  num=10;
            }
            maximum=2^num;
            t=(int)uniform(0,maximum);
            return slot_size*t;
      }
      :
      public void execute(){
            switch(phase){
            case Idle:{
                  <<Set phase to Wait>>
                  wait(0.0);
                  break;
            }
            case Wait:{
                  if (channel_clear()){
                        <<Create & transmit frames to two directions>>
                        <<Set phase to Listen>>
                        wait(2*slot_time);
                  }else{
                        susPend();//continue waiting
                  }
                  break;
            }
            case Listen:{
                  if <<Frame is or will be successfully transmitted>>{
                        if <<Both frames have been transmitted>>{
                              <<Set phase to Finish>>
                              wait(0.0);
                        }else{
                              <<Wait until 2 frames left this station>>
                              <<Set phase to Finish>>
                              susPend();
                        }
                  }else{
                        <<Retransmit this frame>>
                  }
                  break;
            }
            case Contention:{
                  <<Set phase to Wait>>
                  wait(0.0);
                  break;
            }
            case Finish:{
                  <<Reset this station>>
                  <<Set phase to Idle>>
                  wait(0.0);
                  break;
            }
            }
       }
}

// Aframe.java
// Models frames transmitted between stations in Ethernet
class Aframe extends SProcess {
      EthernetKernel ek;
      int source;          /* The sender station of the frame */
      int direction;       /* The direction of propagation */
      int arr_sta;         /* The station arriving at */
      int dep_sta;         /* The station leaving from */
      <<initialize other parameters of a frame>>
      :

      public Aframe(String n, EthernetKernel h)
      {
            super();
            name = toString().toCharArray();
            ek = h;
      }
      :
      void invalidate_frame(int station){
            <<Mark this station’s transmission to be fail>>
            if <<Station transmit frame left>>{
                  passivate(<<Left frame>>);//remove from FEL
                  <<Left frame>>.wait(noise_burst);
            }
            if <<Station transmit frame right>>{
                  passivate(<<Right frame>>);
                  <<Right frame>>.wait(noise_burst);
            }
      }
      :
      public void execute(){
            switch(phase){
            case Arrival:{
                 this_stn=this.arr_stn;
                 if (<<The arrival station is in Listen state>> and
                     <<The station is not corrupted>>){
                       invalidate_frame(this_stn);
                 }

                 if (<<Frame going to left>>){
                       <<Flag that a frame passing by from left>>
                       if (<<Frame not at end of cable.>>){
                             wait(neighbour_delay);
                       }else{
                             terminate();
                       }
                 }else{//Frame going to right
                       <<Flag that a frame passing by from right>>
                       if (<<Frame not at end of cable.>>){
                             wait(neighbour_delay);
                       }else{
```

```
                terminate();
            }

            break;
        }
        case Departure:{
            if (<<This station is the frame’s source station>>){
                <<Flag that a frame has been transmitted>>
                if (<<This station finished both frames>>){
                    if (<<Frame not corrupted>> and
                        <<Station is not in Listen state>>){
                         reactivate(<<This station>>);
                    }
                }
            }

            if (<<Frame is going to left>>){
                <<Frame passing by decrease by 1, reset flag>>
                if (<<Frame can be sent to left>>){
                    wait(neighbour_delay);
                }else{
                    terminate();
                }
            }else{//Frame is going to right
                <<Frame passing by decrease by 1, reset flag>>
                if (<<Frame can be sent to right>>){
                    wait(neighbour_delay);
                }else{
                    terminate();
                }
            }

            if (<<Channel is clear after this departure>> and
               <<This station waits a clear channel>>){
                 reactivate(<<this station>>);
            }

            break;
        }
        }
    }
```