Simulation: Transactions of The Society for Modeling and Simulation International (accepted for publication), 26 pages.

A Framework for Formalization and Strictness Analysis of Simulation Event Orderings

Y.M. Teo[1,2] and B.S.S. Onggo[2]
[1]Singapore-Massachusetts Institute of Technology Alliance, Singapore 117576
[2]Department of Computer Science, National University of Singapore, Singapore 117543
email: {teoym, bhaktisa}@comp.nus.edu.sg

Abstract

This paper advocates the use of a formal framework for analyzing simulation performance. Simulation performance is characterized based on the three simulation development process boundaries: physical system, simulation model, and simulator implementation. Firstly, we formalize simulation event ordering using partially ordered set theory. A simulator implements a simulation event ordering, and incurs implementation overheads when enforcing event ordering at runtime. Secondly, we apply our formalism to extract and formalize the simulation event orderings of both sequential and parallel simulations. Thirdly, we propose the relation stricter and a measure called strictness for comparing and quantifying the degree of event dependency of simulation event orderings respectively. In contrast to the event parallelism measure, strictness is independent of time.

1. Introduction

As the size and complexity of simulations grow, the computational demand required is fast becoming a limited factor in solving large and complex real-world problems. Consequently, understanding simulation performance becomes increasingly important. Parallel simulation speeds up simulation execution by distributing the simulation across a number of processors. In parallel simulation, a physical system is viewed as a number of physical processes that interacts in some fashion [15]. In the Virtual Time simulation modeling paradigm, each physical process is modeled by a logical process (LP) [21]. The interactions between physical processes are modeled by exchanging timestamped events between the corresponding logical processes. Parallelism is exploited by simulating LPs concurrently.

Research in parallel simulation in the last decade has resulted in a number of synchronization protocols [15]. These protocols, introduced in an algorithm fashion, are frequently evaluated by comparing its performance among protocols [15]. Moreover, the performance metrics and benchmarks used vary among the different studies. A serious drawback is the lack of performance comparison framework. We proposed a time and space performance evaluation framework based on the concept of event ordering [28, 35]. Event ordering in simulation refers to a set of rules that is used to order a set of events. The framework characterizes simulation performance along the three natural boundaries in simulation modeling and analysis (see Table 1) [28]. The *physical system* layer corresponds to real world systems, the *simulation model* layer corresponds to different simulation event orderings that can be used to simulate a real world system, and the *simulator* layer corresponds to the simulator implemented to enforce a simulation event ordering. The layered approach provides a framework to study the factors affecting simulation performance from the physical system to its simulator implementation.

Layer	Types of Event	Event Ordering
Physical System	Real events	One
Simulation Model	Real events	Many
Simulator	Real events + Overhead	One or more implementations for a given event ordering

Table 1: Layered Simulation Performance Framework

Event ordering (or message ordering) has been studied in the time management component of High Level Architecture (HLA) [15]. The simulation of a physical system in HLA is distributed into a number of federates. Message ordering in HLA time management dictates the ordering of messages within each federate. Fujimoto introduces *five* message orderings which form a spectrum of orderings where at one extreme, messages are not ordered; and at the other extreme, messages are totally ordered based on their timestamp [12]. To exploit the temporal uncertainty in a simulation model, Fujimoto proposed Approximate Time (AT) and Approximate Time Causal (ATC) orders [14]. Recently, Zhou et al. investigate the causality issue in distributed simulation and propose the causal receive ordering [38]. In parallel simulation, event ordering dictates the ordering of events within each LP and across LPs. To produce correct simulation result, events in an LP are executed in non-decreasing timestamp order [15]. This constraint is referred to as local causality constraint (lcc). Different synchronization algorithms impose different ordering rules in executing events. Different runtime event execution schedules produce the same simulation results but with differing execution performance [25, 36].

A particularly vexing problem with event ordering happens when an LP receives a number of distinct events at exactly the same simulation time. Wieland studied this problem, noting that the results of the simulation are sensitive to the particular ordering assigned to simultaneous events. He proposed that the problem be handled statistically: a sampling of all possible simultaneous event orderings is executed, and the resulting distribution would be more meaningful than a single-point estimate derived from an arbitrary temporal tie-breaking mechanism [37].

This paper discusses the formalization of simulation event orderings based on partially ordered set (poset). The formalization provides a theoretical foundation for carrying out performance analysis of simulation. If events with the same timestamp are grouped as an ordered set of simultaneous events, there will be only one event ordering in a physical system. This paper shows that a simulator implementation implements a specific event ordering. Two major benefits may be derived from the separation of simulation event ordering from its implementation. First, this facilitates the understanding of the relationship of different event orderings. We propose the relation *stricter* to compare different event orderings. Second, the performance of different event orderings can be evaluated independent of implementation overheads [28, 35]. The separation between event ordering and its implementation is motivated by research in memory operation orderings in memory consistency model [8, 19] and message ordering in broadcast communication services in distributed system [2, 20]. Memory consistency model recognizes a number of memory operation orderings such as sequential consistency model [23]. Sequential consistency model can be implemented in various ways [1, 18, 24, 33]. Similarly, broadcast communication services recognize a number of message orderings such as causal order [2, 20]. Many algorithms have been proposed to implement causal order [3, 16, 29, 31, 32].

Our work is different from Critical Path Analysis (CPA) that is also used to analyze the performance of parallel simulation [3]. CPA uses an event dependency graph that is based on *happened before* event ordering [22]. Hence, CPA is a subset of our event ordering analysis.

The rest of this paper is organized as follows. Section 2 formalizes the concept of simulation event ordering. We apply this formalism to extract and formalize simulation event orderings in both sequential and parallel simulation. In section 3, we propose the *stricter* relation and apply this concept to analyze a number of event orderings. We show the empirical result in section 4. Our concluding remark is in section 5.

2. Formalization of Event Orderings

Event ordering is an important concept in discrete-event simulation. In this section, we propose to formalize simulation event ordering based on partially ordered set (poset). Research in poset theory was triggered by Dushnik and Miller's publication in 1941 [10]. They propose the definition of partial order as given in Definition 1.

Definition 1. *An order R over S (where S is a set) is called a* ***partial order*** *if R is anti-reflexive* (*i.e.* $(x, x) \notin R$)*, anti-symmetric* (*i.e. either* $(x, y) \in R$ *or* $(y, x) \in R$)*, and transitive.*

For example, an order ″descendant of″ for a given set of people is of partial order. However, an order ″friend of″ for a given set of people may not be a partial order depending on the given set of people. This leads to the concept of *partially ordered set* [10].

Definition 2. *A* ***partially ordered set*** **(*****poset*****)** *is a tuple* (S, R) *where S is a set and R is a partial order on the set S.*

Definition 3. ***A simulation event ordering*** *(or event ordering in short) is a tuple (E, S_R) where E is a set of events and S_R is a set of comparable events based on event order R. Event order R must be anti-reflexive, anti-symmetric and transitive.*

Based on the definition of poset, we formalize simulation event ordering (referred to as event ordering in short) in Definition 3. Just as a poset that has two components, an event ordering also comprises two main components: a set of events *E* and an event order *R*. An event order *R* refers to a set of rules that is used to order events. A pair of events $(x, y) \in S_R$ denotes event *x* is ordered before event *y* in event order *R*. Two events *x* and *y* are *comparable* if either $(x, y) \in S_R$ or $(y, x) \in S_R$, otherwise *x* and *y* are non-comparable (or *concurrent)*.

2.1 Physical System

An event order in the physical system corresponds to how events in the physical system are ordered. Based on the physical time, there is only one event order for any physical system, i.e. an event with a smaller physical time is ordered before an event with a larger physical time (Definition 4). The definitions predecessor and antecedent (Definition 5 and 6) will be used throughput this paper.

Definition 4. *Let x be an event in a physical system and x.ts the physical time when event x happens. The event order in any physical system dictates that for all events x and y (where x ≠ y), x is ordered before y if and only if x.ts < y.ts.*

Figure 1a shows a physical system with four service centers, S_1, S_2, S_3, and S_4. Figure 1b shows the corresponding snapshot of event occurrences. Horizontal axis represents physical time and vertical axis represents service centers. The physical time in Figure 1b is expressed in timestamp unit. Labels a_i^t and d_i^t represent an event arrival and departure of job *i* at time *t* respectively. A shaded circle represents an event arrival and unshaded one represents an event departure. The snapshot shows that at time 0, job 1 arrives at S_1. Since, S_1 is idle, job 1 is processed until time 4. Job 2 arrives at S_1 at time 2. Since S_1 is busy, this job must wait until S_1 completes job 1 and so on. A dashed arrow from x to y shows that x is the predecessor of y; and a solid arrow from x to y shows that x is the antecedent of y. The definition of predecessor and antecedent are given in Definition 5 and 6 respectively. For example, in Figure 1b, event a_2^2 is the predecessor of event d_1^4 and event d_1^4 is the antecedent of event a_5^5.

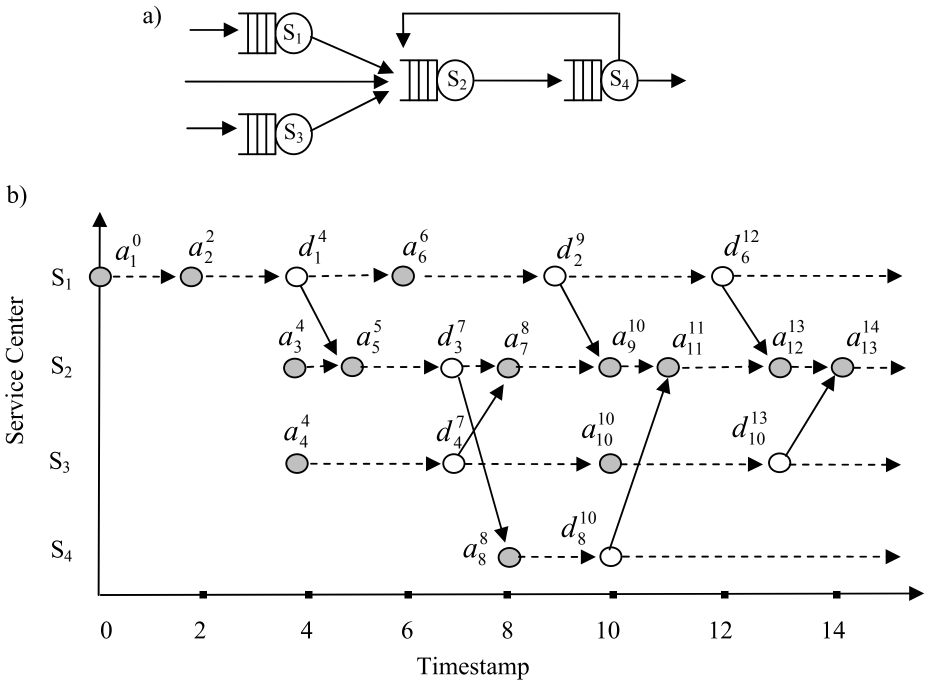


Figure 1: Snapshot of Event Occurrences in a Physical System

Definition 5. *Event x is the* ***predecessor*** *of y* (*denoted by y.pred = x*), *if x and y are at the same service center and there is no other event z that is also at the same service center such that x.ts < z.ts < y.ts.*

Definition 6. *Event x is the* ***antecedent*** *of y* (*denoted by y.ante = x*), *if x spawns y.*

2.2 Simulation Model

Based on the virtual time paradigm, a simulation model emulates a physical system and the interaction among physical processes in the physical system (see Figure 2). Each physical process in the physical system is mapped onto a logical process (LP) in the simulation model. Each event in the simulation model models an event in the physical system. The simulation time of an event in the simulation model models the physical time of the corresponding event in the physical system. The event ordering in a physical system can be modeled and simulated using different event orderings to exploit different degrees of event parallelism.

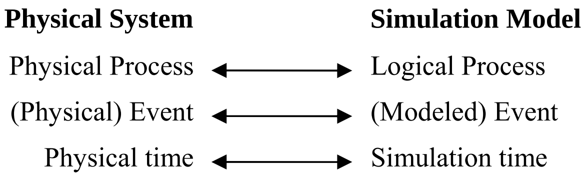


Figure 2: Mapping between Physical System and Simulation Model

Lamport defined *happened before partial order* and *total order* [22], and proved that both orders are anti-reflexive, anti-symmetric, and transitive which match our definition of simulation event order (Definition 3). Hence, we refer to these event orders as partial event order and total event order respectively (see Definition 7 and 8). The priority function in total event order is used to decide which event should be processed when two or more events have the same timestamp. Based on interval order in poset [11], we formalize timestamp event order and time-interval event order [28, 35]. Their definitions are given in Definition 9 and 10 respectively.

Definition 7. ***Partial event order*** *imposes that event x is ordered before event y in if (y.pred = x) or (y.ante = x).*

Definition 8. ***Total event order*** *imposes that event x is ordered before event y in iff (x.ts < y.ts) or (x.ts = y.ts and priority(x) < priority(y)).*

Definition 9. ***Timestamp event ordering*** *imposes that event x is ordered before event y in iff x.ts < y.ts.*

Definition 10. ***Time-interval event ordering*** *imposes that event x is ordered before event y in if: (y.pred = x) or (y.ante = x) or (x.ts + W < y.ts), where W is a constant window size.*

To produce correct simulation result, it is sufficient that each LP executes events in non-decreasing timestamp order [15]. This constraint, commonly referred to as *local causality constraint* (*lcc*) is formalized in Definition 11.

Definition 11. ***Local causality constraint*** *imposes that if for any two distinct events x,y ∈ E, and y.pred = x then x is ordered before y.*

2.3 Simulator

A simulator, written as a sequential program or a parallel program, is an implementation of a simulation model. In parallel simulation, a synchronization algorithm (or simulation protocol) is required for maintaining correct event ordering across processors. Enforcing event ordering at runtime incurs implementation overhead such as null messages in CMB protocol and rollback in Time Warp protocol that results in performance loss.

To show that each simulator implements a certain event order, we extract and formalize the ordering rules of a number of simulator implementations. These include sequential simulation and parallel simulation protocols such as CMB [7], Bounded Lag [25], Time Warp [21], and Bounded Time Warp [36].

2.3.1 Sequential Simulation

The sequential simulation algorithm is presented in Figure 4. Events in sequential simulation are totally ordered (only one event is executed at any time). To enforce this ordering, sequential simulation maintains a future event list (FEL) where events are sorted in chronological timestamp order. In line 3, the function *f* returns the event with the smallest timestamp in the future event list. FEL enables sequential simulation to execute an event with the smallest timestamp (line 12). In case of a tie (i.e. $M \neq \emptyset$ in line 14), an event with the highest priority will be chosen (*z* in line 15). Issues and examples on implementing the priority function have been studied in [30, 37]. Lemma 1 formalizes the event ordering in sequential simulation.

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

Figure 3: Algorithm of Sequential Simulation

Lemma 1. *Sequential simulation implements a total event order.*

Proof. Sequential simulation employs a global event list that is sorted by the smallest timestamp first. This guarantees that event *x* is ordered before event *y* if and only if $x.ts < y.ts$. The use of a priority function when more than one event have the smallest timestamp guarantees that if $x.ts = y.ts$ event *x* is ordered before event *y* if and only if $priority(x) < priority(y)$. ❑

The algorithm presented in Figure 3 does not use LPs. If LPs are used, the priority function only provides a total ordering per LP. Therefore, the priority function could return equal priority for two events from different LPs. In that case, the ordering would depend on the implementation of the event list and the order that the initial events were generated.

2.3.2 CMB Protocol

The algorithm of CMB protocol [7] is given in Figure 5. Each LP maintains a list of LPs that may send events to it (for LP *x*, it is denoted by *SENDER*(*x*)). The ordering rule of CMB protocol imposes that only a safe event can be executed. An event in LP *x* is safe for execution if no other LP ∈ *SENDER*(*x*) will send any event with a smaller timestamp to LP *x*. Therefore, to maintain this ordering, LP *x* must wait for other LP ∈ *SENDER*(*x*) to send their events (see line 5). This could lead to deadlock as all LPs are blocked. To avoid deadlock null messages are introduced. Each null message is stamped with a timestamp, *ts,* which is equal to LP's local simulation clock plus a lookahead value (line 13) to indicate that the sending LP will never transmit any events with a smaller timestamp than *ts*.

```
CMB PROTOCOL
1. initialization
2. run all LPs

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

Figure 4: Algorithm of CMB Protocol

Each LP maintains an event list (EL), a set of input buffers (IB) and a set of output buffers (OB). IB[i] of an LP *x* stores the incoming message from LP$_i$ ∈ *SENDER*(*x*). OB[i] stores the messages that will schedule events in LP$_i$. An LP is blocked if at least one of its IBs is empty (line 4). Function *f* in line 6 is the same function that is used in the sequential simulation shown in Figure 4. The function chooses an event with the smallest timestamp from the IBs and EL for execution. Line 7 removes the chosen event from the corresponding list (one of the IBs or EL). The local clock is updated in line 8. In line 9, an event execution may schedule a set of internal events (IE) and a set of external events (EE). The internal events (i.e. scheduled to happen in the same LP) are saved to EL (line 10) and external events (i.e. scheduled to happen in other LPs) are saved to their respective OB (line 11). Line 12 sets a null message with a timestamp equal to the local clock plus a lookahead value. Line 13 adds a null message to any empty OB. Line 14 sends all the external events and null messages in OBs. Finally line 16 checks the stopping condition.

Lemma 2. *CMB protocol implements an event order whereby event x is ordered before event y if:*

1. *$y.pred = x$, or*
2. *$x.ts + la < y.ts$.*

Proof. The conditional statement in line 4 (Figure 5) ensures that an LP has to wait until all LPs in its SENDER list have sent their events. This ensures that an LP always executes events scheduled in it in timestamp order. Hence, for all events in the same LP, if $y.pred = x$ then x is ordered before y. Further, event y in LP$_j$ is executed only if it has the smallest timestamp among the unprocessed events of all LP $\in$ *SENDER*(LP$_j$). Therefore, event x in any LP $\in$ *SENDER*(LP$_j$) is ordered before event y only if $x.ts + la < y.ts$ where la is the lookahead value. ❑

Researchers have proposed various optimizations such as demand driven protocol [5], flushing protocol [34], and carrier null message protocol [6] to reduce the null-message overhead. These optimizations do not alter the event ordering in the original CMB protocol, but rather, they can be seen as different implementations of the same event order.

2.3.3 Bounded Lag Protocol

Lubachevsky proposed the Bounded Lag (BL) protocol which combines two main rules: bounded lag restriction and minimum propagation delay [25]. Bounded lag restriction imposes that events can be executed concurrently if they are within the same time window. Minimum propagation delay between LPs is used to determine whether an event is safe to execute. The latter is similar to the rule in CMB protocol; however in the implementation BL protocol uses a distance matrix instead of using null messages. To maintain its ordering, BL protocol uses barrier synchronization because the global clock (for imposing bounded lag restriction) and the minimum propagation delay must be broadcasted to all LPs. The algorithm is given in Figure 6.

```
BL PROTOCOL
1.  initialization
2.  run all LPs

LOGICAL PROCESS
3.  while (~stop) {
4.    $\beta \leftarrow \min\{\forall lp \in LP, e=head(lp.EL) \bullet e.timestamp+d(lp, this)\}$
      $\gamma \leftarrow \min\{\forall lp \in LP, e=head(EL) \bullet e.timestamp+d(this,lp)+d(lp,this)\}$
      $\alpha \leftarrow \min\{\beta, \gamma\}$
5.    barrier synchronization
6.    $E \leftarrow \{\forall e \in EL \bullet e.timestamp \le \min(\alpha, global\_clock+W)\}$
7.    $EL \leftarrow EL - E$
8.    while ($E \ne \varnothing$) {
9.        $e \leftarrow head(E)$
10.       $E \leftarrow E - \{e\}$
11.       $\{IE, EE\} \leftarrow execute(e)$
12.       $local\_clock \leftarrow e.timestamp$
13.       $EL \leftarrow EL \cup IE$
14.       Send(EE)
15.   }
16.   $stop \leftarrow g()$
17.   barrier synchronization
18.   $global\_clock \leftarrow \min\{\forall lp \in LP \bullet lp.local\_clock\}$
19.   barrier synchronization
20.}
```

Figure 5: Algorithm of Bounded Lag Protocol

There are two main processes: the nomination of safe events (lines 4-7) and the execution of safe events (lines 8-15). The lookahead between any two LPs is stored in a distance matrix *d*. Based on the distance matrix, an LP (denoted by *this* in Figure 6) determines α, i.e. the earliest time when its system state can be affected by other LP (line 4). The barrier synchronization (line 5) ensures that all LPs receive α before continuing to the next line. Each LP identifies its safe events based on this rule: events with a timestamp less than α and within a time window of W are safe to process (line 6). W is termed as BL size in [25]. Line 7 removes all safe events from EL for execution. BL protocol retrieves a safe event with the least timestamp in line 9 and removes it from the list E in line 10. In line 11, event execution may schedule a set of internal events (IE) and a set of external events (EE). The internal events will be added to the EL (line 13) and the external events will be sent to their respective LPs (line 14). The barrier synchronization in line 17 is used to ensure that all LPs have processed their safe events before the time window is moved. Line 18 computes the global clock as the minimum of all LPs' local clock. This process is repeated until the stopping condition is met.

Lemma 3. *BL protocol implements an event order whereby event x is ordered before event y if:*

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

Proof. In line 7, α returns the smallest timestamp of an unprocessed event x (plus lookahead) that may be sent to a particular LP (Figure 6). Line 6 shows that if event y in LP_i can be executed in parallel with event x from another LP, then $y.ts \leq \alpha$ (i.e. $x.ts + la$) and both x and y must be in the same time window of size W. Therefore, event x is executed before event y only if $x.ts + la < y.ts$ or events x and y are in two different time windows of size W is true (of course the time window of x should be earlier than the time window of y). ❑

2.3.4 Time Warp Protocol

Jefferson proposed Time Warp (TW) protocol which implements a rule that if event x causes event y, then the execution of event x must be completed before the execution of event y starts [21]. The definition of "*x* causes y" follows the relation *happened before* in [22]. To implement this ordering, TW protocol uses what is called local control mechanism (rollback and state saving) and global control mechanism (global clock calculation and fossil collection). The algorithm is given in Figure 7.

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

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

Figure 6: Algorithm of Time Warp Protocol

Each LP stores all incoming events in an input buffer (IB) which is sorted based on the timestamp of the incoming events. Lines 4-11 find the first real event *m*. Line 5 retrieves an event *m* with the smallest timestamp for execution. Line 6 checks if *lcc* is violated. Line 7 detects whether rollback has to be done. If *m* is an anti-message, line 9 will annihilate the associated event that has to be cancelled; otherwise, it will add *m* to a list called input queue (IQ). IQ is used to store the history of all incoming messages (processed and unprocessed). Line 10 removes *m* from IB. Lines 12-15 retrieve an event *e* which has the smallest timestamp from the EL and choose the event with a smaller timestamp, between *m* and *e*. Line 17 executes the chosen event. This execution may produce a set of internal events (IE) and a set of external events (EE). Line 18 updates the local clock and line 19 updates the event list (EL). Line 20 saves the state of an LP. The global clock is updated in line 21. Events with timestamp less than the global clock will never be rollbacked. These events are called *committed events*. Hence, memory allocated to committed events can be reclaimed with the fossil collection process in line 22. Line 23 sends out the external events. Lastly, line 24 checks the stopping condition.

Lemma 4. *Time Warp protocol implements a partial event order.*

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

2.3.5 Bounded Time Warp Protocol

The Bounded Time Warp (BTW) protocol [36] is proposed to limit the degree of optimism in Time Warp protocol by setting a bound on how far an LP can advance ahead of other LPs. This is accomplished by setting a time window (*W*). All LPs are allowed to optimistically process events ahead of the global clock *(GVT)* but bounded by the time window *GVT+W*. No LP can advance beyond *GVT+W* before all LPs have reached this boundary.

Lemma 5. ***BTW protocol*** *imposes that event x is ordered before event y if:*

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

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

We summarize the formalization of the discussed event orderings in Figure 8. The ordering rules of each event order are shown in the form of *x* is ordered before *y* (denoted by $x \Rightarrow y$) if a list of conditions hold. A simulator implements a certain event order. An arrow from an event ordering *R* in simulation model to simulator *S* denotes that *S* implements *R*.

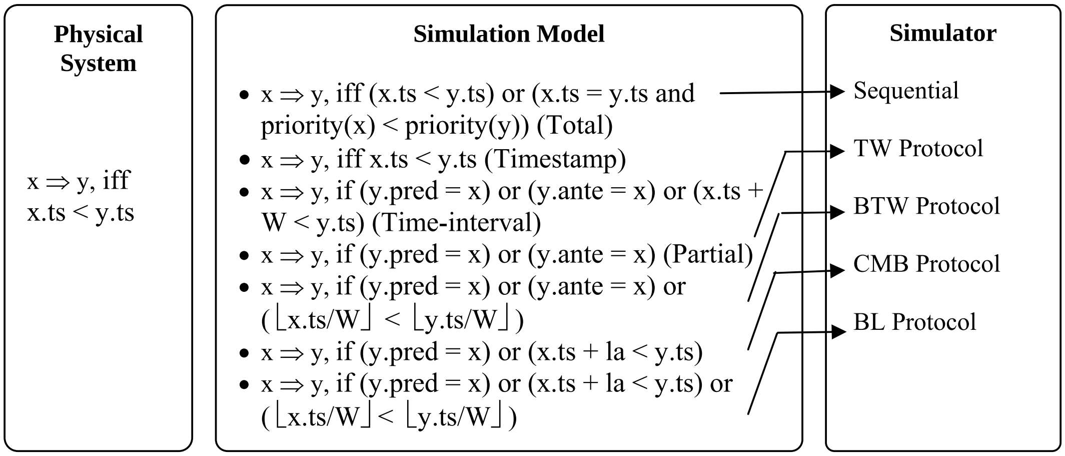

Physical System
$x \Rightarrow y$, iff $x.ts < y.ts$
Simulation Model
• $x \Rightarrow y$, iff $(x.ts < y.ts)$ or $(x.ts = y.ts$ and $priority(x) < priority(y))$ (Total)
• $x \Rightarrow y$, iff $x.ts < y.ts$ (Timestamp)
• $x \Rightarrow y$, if $(y.pred = x)$ or $(y.ante = x)$ or $(x.ts + W < y.ts)$ (Time-interval)
• $x \Rightarrow y$, if $(y.pred = x)$ or $(y.ante = x)$ (Partial)
• $x \Rightarrow y$, if $(y.pred = x)$ or $(y.ante = x)$ or $(\lfloor x.ts/W \rfloor < \lfloor y.ts/W \rfloor)$
• $x \Rightarrow y$, if $(y.pred = x)$ or $(x.ts + la < y.ts)$
• $x \Rightarrow y$, if $(y.pred = x)$ or $(x.ts + la < y.ts)$ or $(\lfloor x.ts/W \rfloor < \lfloor y.ts/W \rfloor)$
Simulator
Sequential
TW Protocol
BTW Protocol
CMB Protocol
BL Protocol

Figure 7: Summary on Simulation Event Ordering Formalization

3. Strictness of Event Orderings

To compare the degree of event dependencies among different event orders, we propose a relation *stricter*. The term stricter is borrowed from memory consistency model [19]. In memory consistency, the stricter relation is used to compare different models by considering the set of possible outcomes that is allowed by each model for a given set of instructions. In simulation event ordering, we consider the set of events that have to be executed one after another due to the ordering rules imposed by an event order for a given set of events.

Definition 12. *An event order R_1 is* **stricter** *than event order R_2 if for any set of events E, $S_{R2} \subseteq S_{R1}$. An event order R_1 is* **incomparable** *to event order R_2 if we can find two sets of events E_1 and E_2, such that $S_{R2} \subseteq S_{R1}$ is true for E_1 but $S_{R2} \subseteq S_{R1}$ is not true for E_2.*

Lemma 6. *Two properties of a stricter relation are:*

1. *if R_1 is stricter than R_2 and R_2 is stricter than R_1, then $R_1 = R_2$ (anti-symmetric).*
2. *if R_1 is stricter than R_2 and R_2 is stricter than R_3, then R_1 is stricter than R_3 (transitive).*

Proof. From Definition 12, event order R_1 is stricter than event order R_2 shows that $S_{R2} \subseteq S_{R1}$. Therefore, if event order R_1 is stricter than event order R_2, and R_2 is stricter than R_1, it means $S_{R2} \subseteq S_{R1}$ and $S_{R2} \subseteq S_{R1}$ are true. Consequently, $S_{R2} = S_{R1}$, which implies $R_1 = R_2$ (Definition 3). Similarly, if event order R_1 is stricter than event order R_2, and R_2 is stricter than R_3, it means $S_{R2} \subseteq S_{R1}$ and $S_{R3} \subseteq S_{R2}$ are true. Consequently, $S_{R3} \subseteq S_{R1}$ is true for any set of events E, which implies that R_1 is stricter than R_3 (Definition 12). ❑

Two events are concurrent in an event order if the event order does not impose any ordering on them. Definition 12 implies that a stricter event order produces less number of concurrent events than a less strict event order (or at most the same number of concurrent events). Since concurrent events can be executed in parallel, a stricter event order produces less event parallelism. To quantify the degree of event dependency, we propose the measure of *strictness*. Since relation stricter is built based on set inclusion, the strictness of event order R is quantified based on the number of elements in S_R as shown in Definition 13.

Definition 13. *The* **strictness** *of an event order R (ς_R) is defined as $\| S_R \| / \| S_{tot} \|$ where $\| S_R \|$ and $\| S_{tot} \|$ is the size of the set of comparable (or non-concurrent) events ordered by R and the total event order respectively.*

Since total event order is the strictest event order, we normalize the number of elements in S_R with the number of comparable elements in total event order (S_{tot}). Hence, the strictness of an event order ranges from zero when $S_R = \varnothing$ and one when R is the total event order. To measure $\| S_R \|$ for a given set of events E, we have to determine for any two events x and $y \in E$ whether $(x, y) \in S_R$ based on the ordering rules of the event order. This process is computationally expensive especially for a large number of events. Since $(x, y) \in S_R$ implies that event y cannot be executed before the execution of event x completes, in our experiments we measure the number of events that are ready for execution but cannot be executed because of the ordering rules imposed by the event order.

3.1 Strictness Analysis

Event order R_2 is stricter than event order R_1 implies that for any two distinct events x and y, if x is ordered before y in R_1 then x is also ordered before y in R_2, but not vice versa. Therefore, to prove whether an event order is stricter than another event order, we show that the ordering rule of one event order is a subset of the other event order. If the ordering rule of event order R_1 is a subset of event order R_2, then definitely if x is ordered before y in R_1 then x is also ordered before y in R_2. Using this approach, in the following theorems, we establish the relationships of various simulation event orderings. The spectrum of various simulation event orderings and its strictness is summarized in Figure 9.

Theorem 1.

a) *Total event order is stricter than TS event order.*

b) *The event order of BL protocol is stricter than the event order of CMB protocol*

c) *The event order of BTW protocol is stricter than partial event order.*

Proof. The proofs are derived by comparing their properties in Figure 8. If the property of an event order R_1 is a subset of the property of event order R_2, then R_2 is stricter than R_1. ❑

Theorem 2. *TS event order is stricter than BL event order.*

Proof. In TS event order, $x \Rightarrow y$, iff $x.ts < y.ts$. On the other hand, in BL protocol, $x \Rightarrow y$, if ($y.pred = x$) or ($x.ts + la < y.ts$) or ($\lfloor x.ts/W \rfloor < \lfloor y.ts/W \rfloor$). These rules can only be true if $x.ts < y.ts$. Therefore, if $x \Rightarrow y$ in BL protocol, then $x \Rightarrow y$ is true in TS event order, but not the converse. Hence, timestamp event order is stricter than the event order of BL protocol. ❑

Lemma 7. $\forall x, y \in E,\ \{y.ante = x\} \subseteq \{x.ts + la \leq y.ts \text{ and } x.lp \in SENDER(y.lp)\}$.

Proof. From the definition of *SENDER* list and lookahead, if $y.ante = x$ then $x.lp$ must be in the *SENDER* list (i.e. $x.lp \in SENDER(y.lp)$) and the timestamp difference between x and y must be greater than the lookahead la (i.e. $x.ts + la < y.ts$). However, it is possible that $x.lp \in SENDER(y.lp)$ and $x.ts + la < y.ts$ is true but $y.ante \neq x$. ❑

Theorem 3. *The event order of CMB protocol is stricter than partial event order.*

Proof. Both have two ordering rules (Figure 8). The first rule is the same, i.e. $x \Rightarrow y$ if $y.pred = x$. In the second rule, partial event order imposes $x \Rightarrow y$ if $y.ante = x$ whereas CMB protocol imposes that $x \Rightarrow y$ if $x.ts + la < y.ts$. Lemma 7 shows that the second rule of partial event order is a subset of the second rule of CMB protocol, therefore, the event order of CMB protocol is stricter than partial event order. ❑

Theorem 4. *The event order of BL protocol is stricter than the event order of BTW protocol.*

Proof. Both have three ordering rules (Figure 8) and two of them are the same, i.e. $x \Rightarrow y$ if $y.pred = x$ or $\lfloor x.ts/W \rfloor < \lfloor y.ts/W \rfloor$. The other rule is different, BTW protocol imposes $x \Rightarrow y$ if $y.ante = x$ whereas BL protocol imposes that $x \Rightarrow y$ if $x.ts + la < y.ts$. Based on Lemma 7, BL protocol imposes a stricter event order than BTW protocol for the same window size W. ❑

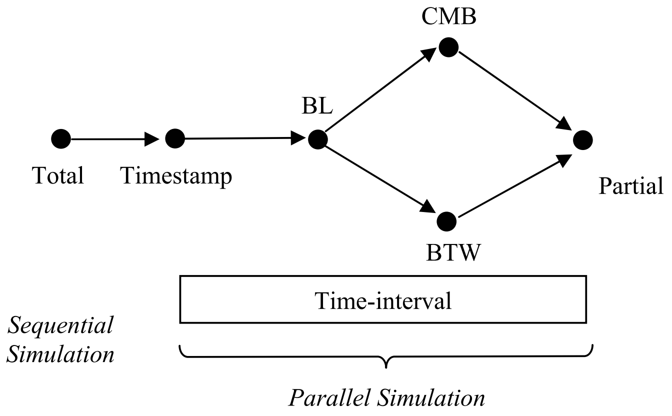


Figure 8: Spectrum of Simulation Event Orders and its Strictness

Figure 9 shows the spectrum of event orders based on our proposed stricter relation. BL, BTW and CMB refers to the event ordering of BL protocol, BTW protocol and CMB protocol respectively. An arrow from event order R_1 to event order R_2 denotes that R_1 is stricter than R_2. Stricter relation is transitive and the arrows can be traversed transitively as well. Sequential simulation implements total event order and the remaining event orders belong mainly to parallel and distributed simulation.

Depending on its window size, the relative position of TI event order can be anywhere between timestamp event order and partial event order. If TI event ordering uses a window size of zero, then it becomes a timestamp event ordering. Similarly, there is a constant c such that $0 < c < W$ where time-interval event ordering becomes partial event ordering (W is the window size) as shown in Theorem 5 This property is useful in strictness analysis because we can create different points (representing different event orderings) between timestamp event ordering and partial event ordering by changing the value of W.

Theorem 5. *For a given set of events E, there is a constant c such that* $0 < c < W$*, where a TI event order will become a partial event order.*

Proof. To prove this, we show that if $0 < c < W$, the third rule of time-interval event order (i.e. $x.ts + W < y.ts$) is redundant. Let a and b be two distinct events in E where $b.pred \neq a$ and $b.ante \neq a$ and $b.ts - a.ts = c$ is the largest. If $W > c$, then the rule $x.ts + W < y.ts$ will produce an empty set. Hence, only the first two rules (*y.pred* = *x and y.ante* = *x*) determines the ordering, resulting in TI event order with $W > c$ and partial event order producing exactly the same event ordering. ❑

4. Empirical Result

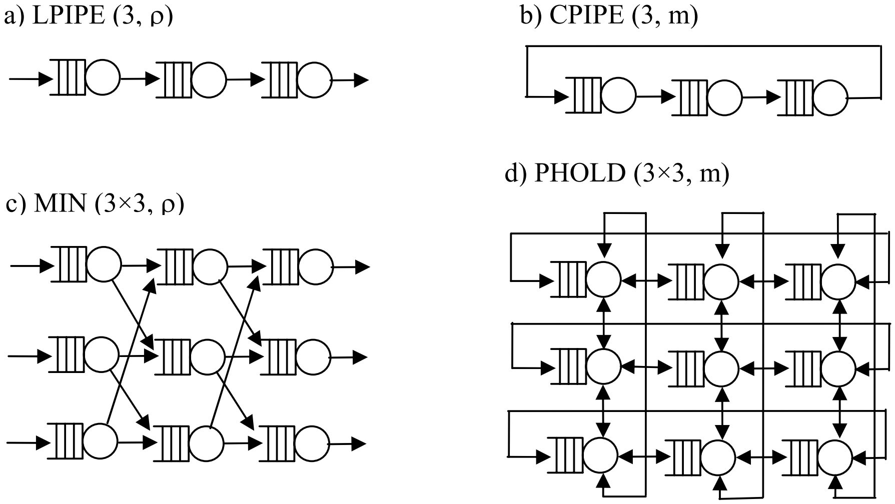
We measure the strictness of five event orders (i.e. total, timestamp, time-interval, CMB and partial) using four benchmarks:

a) *Linear Pipeline (LPIPE)* represents a simple *open system*. It is parameterized by the number of service centers (n), and traffic intensity (ρ) which is the ratio between arrival rate (λ) and service rate (μ).

b) *Circular Pipeline (CPIPE)* represents a simple *closed system*. It is parameterized by the number of service centers (n) and job density (m) which is the average number of jobs in a service center.

c) *Multistage Interconnected Network (MIN)* represents a more complex open system with multiple fork and merge structures [34]. The jobs in any service center (except at the last column) will be sent to one of the two neighbors with equal probability. It is parameterized by the number of service centers ($n \times p$), and traffic intensity (ρ).

d) *Parallel Hold (PHOLD)* represents a closed system with multiple *feedbacks* [12]. A job in any server can move to one of the four neighbors with an equal probability. Initially, jobs are distributed equally among the service centers. It is parameterized by the number of service centers ($n \times p$) and job density (m).



a) LPIPE (3, ρ)

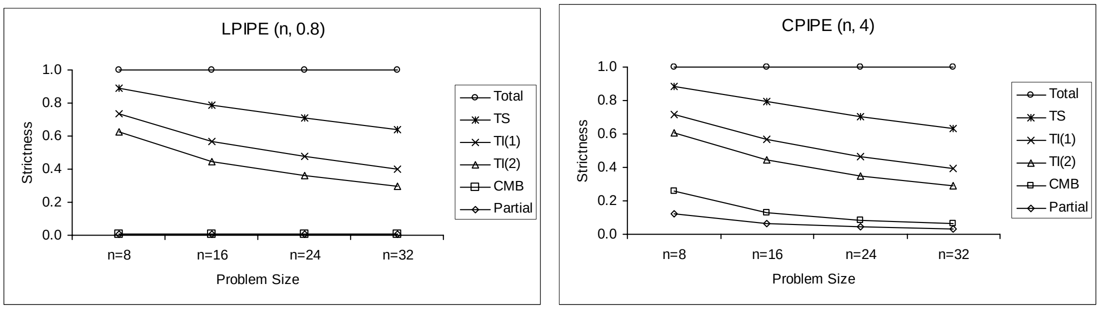
b) CPIPE (3, m)

c) MIN (3×3, ρ)

d) PHOLD (3×3, m)

Figure 9: Benchmarks

We measure the strictness of the event orderings using a Time and Space Analyzer (TSA) that we have developed [28]. The simulation duration is set at 100,000 timestamp units. Figure 10 shows the strictness of event orderings as problem size increases. First, the result shows that the strictness value is between 0 and 1, where total event order is the strictest event order. Second, the figure reveals that partial event order, the event order of CMB protocol, timestamp event order and total event order are in the order of increasing strictness. This confirms their positions on the spectrum of event orders in Figure 9. The time-interval event order with time windows of 1 and 2 are used to represent two event orderings with different degrees of strictness. As we reduce the window size, the curve for time-interval event order moves towards timestamp event order and conversely, when we increase the window size, it moves towards the partial event order.



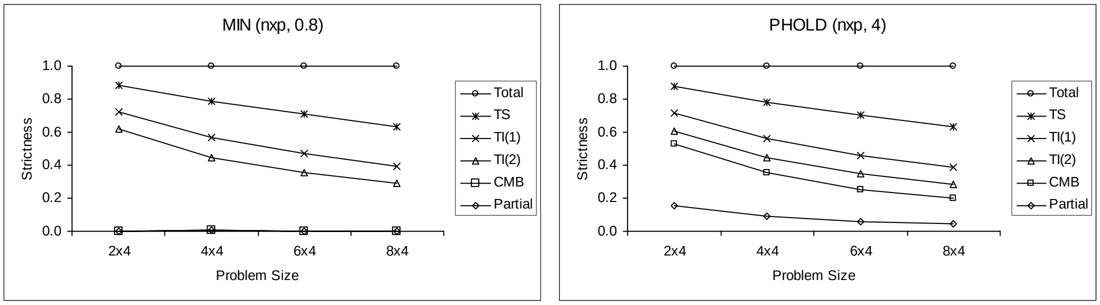


Figure 10: Strictness of Event Orderings

As problem size increases and consequently the number of events, strictness reduces. This is due to the higher probability of concurrent (non-comparable) events in the benchmarks. The strictness measure shown also reflects that the degree of event dependency in closed system is higher than in open system. Misra reported that CMB protocol can achieve optimum performance for a tandem topology and any acyclic topology [26]. Our result confirms this, i.e., strictness of the CMB (and partial) protocols for the open MIN($n \times p$, 0.8) system is lower than in the closed PHOLD($n \times p$, 4) system with multiple feedbacks.

5. Conclusions

The main contribution of this paper is the formalization of simulation event ordering based on partially ordered set theory, and the strictness analysis of various simulation event orderings. First, we characterized simulation performance along the three natural boundaries in simulation modeling and analysis namely: *physical system*, *simulation model* and *simulator*; and formalized the event orderings in each of the layers. Events in a physical system are ordered based on their time of occurrences. In simulation, different event orderings can be used to simulate the physical system. In the implementation, simulator ensures that the chosen event ordering is maintained throughout a simulation run. We extract and formalize the event orderings of both sequential and parallel simulation. To compare the event dependency among different event ordering, we propose the *stricter relation* and to quantify the degree of event dependency a new *strictness measure* is proposed.

References

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

16. Gambhire, P. and Kshemkalyani, A.D. Evaluation of the Optimal Causal Message Ordering Algorithm. *Proceedings of the High Performance Computing, LNCS* no. 1970, Springler-Verlag, pp. 83-95, 2000.

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

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

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

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

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

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

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

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

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

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

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

28. Onggo, B.S.S. and Teo Y.M. Performance Trade-off in Distributed Simulation. *Proc. 6th IEEE International Workshop on Distributed Simulation and Real Time Applications*, IEEE Computer Society Press, pp. 77-84, 2002.

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

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

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

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

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

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

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

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

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

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