Hardware Support for Aggressive Parallel Discrete Event Simulation

Sudhir Srinivasan
Paul F. Reynolds, Jr.

Computer Science Report No. TR-93-07
January 11, 1993

Hardware Support For Aggressive Parallel Discrete Event Simulation

Every Parallel Discrete Event Simulation (PDES) protocol can benefit significantly from the availability of global information: non-aggressive protocols require deadlock detection and recovery; aggressive protocols require Global Virtual Time (GVT), and global and local status information is critical to adaptive protocols. To rapidly compute and disseminate global information, Reynolds proposed a framework for all PDES's. As proposed, the framework could clearly operate with a non-aggressive protocol. To demonstrate its utility to aggressive protocols, we describe an algorithm to compute GVT using the framework in a Time Warp system, and we establish its correctness. Aggressiveness, used in moderation, (i.e. adaptive behavior) will most likely lead to the most efficient protocols. The algorithms described in this paper can be used as a foundation for designing adaptive protocols which utilize the framework to obtain status information. Finally, our approach to computing GVT is novel. Simulation studies show that very accurate GVT can be made available at no cost to the simulation itself.

Categories and Subject Descriptors: D.1.3 [**Programming Techniques**]: Concurrent Programming - *parallel programming*; D.4.1 [**Operating Systems**]: Process Management - *deadlocks*; D.4.1 [**Operating Systems**]: Process Management - *synchronization*; D.4.4 [**Operating Systems**]: Communications Management - *message sending*; D.4.7 [**Operating Systems**] Organization and Design - *distributed systems*; D.4.8 [**Operating Systems**] Performance - *simulation*; I.6.8 [**Simulation and Modeling**] Types of Simulation - *parallel and discrete event*

General Terms: Algorithms, Performance

Additional Key Words and Phrases: Aggressive protocols, GVT computation, reduction network, synchronization algorithm, Time Warp

List of figures

Notation

σ_i	-	*T*-value used to track the logical clock value of LP_i
η_i	-	smallest next event time of LP_i
υ_i	-	smallest unreceived message time of LP_i
ρ_i	-	*T*-value used for primary acknowledgment by LP_i
τ_i	-	*T*-value used for secondary acknowledgment by LP_i
μ_i	-	minimum timestamp of $AP_i = MIN(\sigma_i, \upsilon_i)$

1 Introduction

1.1 Overview

Parallelizing Discrete Event Simulation (DES) is a challenging problem in parallel computing. Since simulated events have causal relations among them, they must be simulated in the order defined by these causal relations. A dependence graph can be constructed using these causal relations, which defines a partial order on all of the simulated events. Events which are not related under this partial order can be executed concurrently. It appears that typical DES's exhibit substantial concurrency in their dependence graphs [9, 15, 19] which makes DES a candidate for parallelization. Ironically though, this concurrency is difficult to extract in practice. The well-investigated approach of extracting concurrency from sequential programs fails with DES due primarily to two reasons: (i) There is a single (main) data structure, the *events list*, which is modified very frequently (ii) The DES algorithm is basically a single loop in which there are several inter-iteration dependencies (because execution of events may schedule more events in the logical future). Consequently, a different method for parallelizing DES has been adopted [4, 19]. The essence of this method is to divide the DES into several *logical processes (LP's)*, each responsible for simulating a part of the physical system. These LP's are sequential DES's by themselves and they synchronize with each other when required using timestamped messages, in order to guarantee accurate simulation (i.e, freedom from causality violations). The method used to synchronize the LP's of a Parallel Discrete Event Simulation (PDES) is generally referred to as a *protocol*. We present a modified version of a previously proposed synchronization algorithm [22], modified to operate with a class of protocols which we call *aggressive* protocols. This class includes the so-called *optimistic* protocols as well as an interesting and as yet relatively unexplored subclass: *adaptive* protocols in which the aggressiveness of the LP's is controlled. Also, we prove the correctness of our algorithm. The algorithm of [22] as well as our algorithm operate on special-purpose hardware designed to desseminate global information rapidly. Thus, the primary contribution of this work is the establishment of a correct method for rapidly disseminating critical synchronization information in support of aggressive (and therefore adaptive) protocols at essentially no cost to the PDES itself. Our algorithm will serve as a foundation for the design of adaptive protocols which use the hardware to gather status information at very low cost.

1.2 Background

In the past, researchers have proposed several protocols [9], most of which belong to one of two categories: *non-aggressive, accurate and without risk* (commonly known as *conservative*) and *aggressive*, *accurate and with risk* (commonly *optimistic*) with some exceptions. In a non-aggressive protocol, an LP generally executes its next scheduled event only after it confirms that no causality constraints can be violated in the future due to this execution. Such protocols tend not to exploit much of the concurrency in the simulation. In an aggressive protocol, an LP executes events without the guarantee of freedom from causality errors. As a result, causality errors may occur. One of the ways of dealing with causality errors is to ignore them, but we do not consider this further here. For the simulation to be accurate, some error recovery must be performed; commonly, *rolling back* to an earlier, correct, saved state is the method of choice. Consequently, aggressive protocols suffer the penalty of saving state periodically and rolling back on errors. The

costs of these actions can become prohibitively high in implementations. Another problem aggressive protocols face is that LP's periodically require a global value called *global virtual time* (GVT) [10]. Computing GVT has thus far been an expensive operation. Note that the general class of aggressive protocols includes optimistic protocols.

Researchers have reported success with both non-aggressive and aggressive protocols [9]. Studies have shown that there are applications for which aggressive protocols will outperform non-aggressive ones [12, 14] and vice-versa [16]. As a result, there is no *optimal* protocol. In [21], Reynolds points out that there are several other categories of protocols, based on a set of *design variables* used to characterize protocols. An interesting category of protocols that has not received much attention is the class of *adaptive* protocols in which the LP's dynamically adapt some aspect or aspects of their processing. An interesting and promising subclass of adaptive protocols are those in which the LP's dynamically control their aggressiveness. Recent research [1, 6, 7, 16, 17] suggests that controlled aggressiveness will most likely be a feature of a universally effi cient protocol. In the rest of this paper, we will use the term *adaptive* to refer to this subclass of adaptive protocols which is characterized by controlled aggressiveness.

Most PDES protocols require global information of some sort [24]. For example, Time Warp LP's require the value of GVT for several reasons. Correspondingly, non-aggressive (or blocking) protocols, which have the potential to deadlock, need to be able to identify the LP with the event that must be executed next, to break a deadlock. Iterative protocols which restrict executable events to those inside a computation window require the LP's to collectively determine the lower and upper bounds of the window. In adaptive protocols as well, there is need for global information, since the amount of aggressiveness of any LP will be controlled by its state relative to the state of others in the system.

The need for global synchronization information motivated a universal *framework* for all parallel simulations, proposed by Reynolds [22]. This framework has the capability of rapidly providing the global information required in any PDES. For instance, it can be used to compute GVT, the minimum lookahead in the system, the floor and ceiling of a computation window or to break deadlocks. We believe that adaptive protocols will benefi t most from this framework because global information required to make adaptive decisions dynamically, which has been very expensive (in time) to compute thus far, can be made available very rapidly using the framework. The framework consists of three parts: (i) A small set of global values required by any PDES protocol (ii) High speed special purpose hardware to rapidly compute and disseminate these global values and (iii) A synchronization algorithm executed by each LP to correctly maintain the global values. (i) and (iii) are described in [23] while (ii) is described in [25]. It must be noted that the synchronization algorithm is not a protocol, but a foundation for PDES protocols. It is intuitively clear that the synchronization algorithm can operate with non-aggressive protocols, in which the logical clocks of the LP's never move backwards. We present an aggressive version of the synchronization algorithm and prove its correctness, thus demonstrating that the framework can be used to support a protocol which allows aggressive processing. In the future, we intend to use our algorithm as a basis for designing effi cient adaptive protocols which use the framework hardware to collect global information. In the rest of the paper we refer to the hardware desgined

for the framework as the *framework hardware* and to the synchronization algorithm as the *framework algorithm*.

In the next section, we briefly describe the hardware and synchronization algorithm of the framework to set the context for our work. We present our algorithm in Section 3 and in Section 4, we show its correctness. Sections 5 and 6 present and establish the correctness of a modified version of our algorithm, which is required due to the organization of the hardware described in Section 2. Section 7 describes three properties of the value of GVT computed by our algorithm. In Section 8, we present the findings of preliminary studies of our algorithm and compare this with previous work in this area. Section 9 concludes the paper.

2 A Framework for PDES

We begin with a brief review of the Reynolds framework. Details can be found in [23]. We first describe the framework algorithm and then the hardware support for the algorithm.

2.1 Framework algorithm

The main purpose of the algorithm is to correctly maintain a set of global values required by the protocol being used in the PDES. At each LP, the algorithm maintains the local counterpart of each global value. For instance, the algorithm could maintain the event processing rate at each LP so that the corresponding global value could identify the fastest LP's. In [22], Reynolds identified two such global values: η', the smallest of the timestamps in the future events list of all the LP's and υ', the smallest of the timestamps of messages that are outstanding (as yet unreceived). Since these values are logical timestamps, we call them *T-values*. As we shall see, these *T*-values are functions of real time. The upper-case "*T*" in "*T*-values" signifies logical time, which is the range of these functions, as against real time, which is their domain. To compute these two global *T*-values, each LP maintains two local *T*-values, one corresponding to each global *T*-value. We define η' and υ' as follows:

$$\eta' = MIN_{LP_i}(\eta_i)$$

$$\upsilon' = MIN_{LP_i}(\upsilon_i)$$

where η_i is the timestamp of the first event in LP_i's events list (which is assumed to be sorted in non-descending order) and υ_i is the smallest of the timestamps of all the messages that LP_i has sent out which have not yet been received by their intended receivers. We call η_i the *next event time* and accordingly, η' the *minimum next event time*. Similarly, υ_i is the *smallest unreceived message time* and υ' is *the minimum unreceived message time*. We emphasize here that the responsibility of the algorithm is only to correctly maintain η' and υ' (or other such global *T*-values). The values are actually used by the PDES protocol working in conjunction with the framework. For completeness, the algorithm presented here uses the two global *T*-values to identify the next *safe* event (or events) in the system, where an event is safe if its execution is guaranteed not to violate any causality constraints in the future.

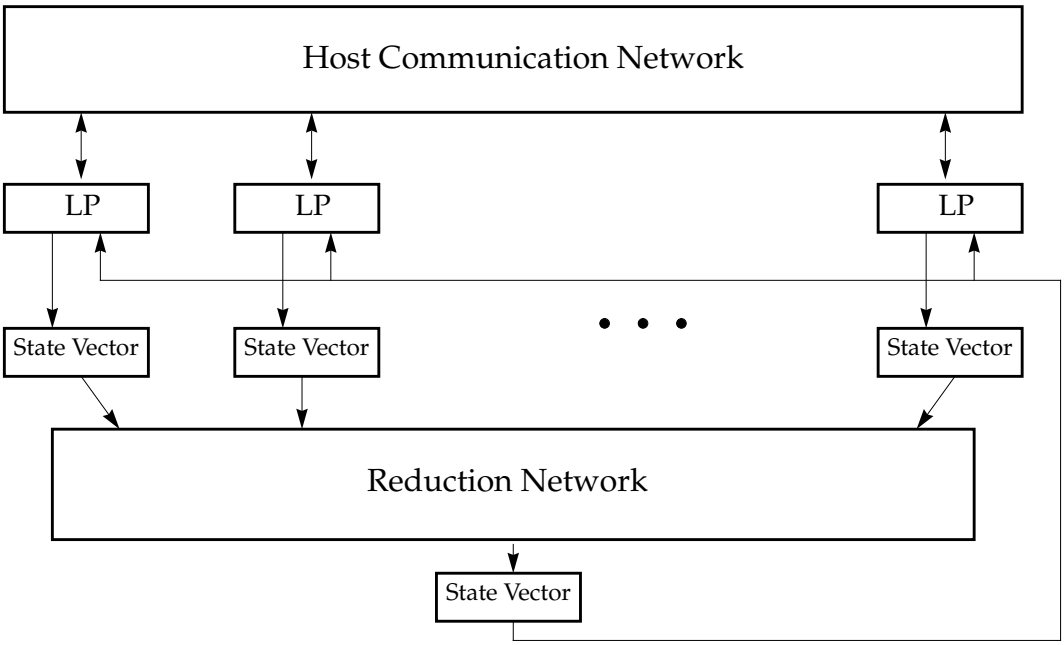


Figure 1 - Global Computation Model

In order to maintain υ_i, when an LP receives a message and incorporates it into its events list, the LP that sent the message must be informed that the message has been received. To achieve this, the algorithm acknowledges each message received by an LP using the two-phase protocol described in [18]. The acknowledgment protocol uses two more *T*-values, ρ_i and τ_i. Each of these *T*-values is actually a triple comprised of the timestamp of the message being acknowledged, a globally unique message identifier (this is required since there may be multiple messages with the same timestamp) and a size field used for batched acknowledgments described later. One way of generating these globally unique message identifiers in a distributed manner with no overhead is by having the identifier consist of three fields: the LP number of the sender, the LP number of the receiver and a sequence number. Each LP maintains one sequence counter for each LP that it sends messages to. When an LP sends a message to another LP, it increments the counter for that receiver and uses its value in the message identifier. Thus, the message identifiers form a contiguous sequence for each ordered pair of communicating LP's. Each LP_i also maintains two lists: a list of sent but not yet received messages called the *outstanding message list* and a list of messages which have been received by LP_i but which it has not yet acknowledged called the *unacknowledged message list*.

The algorithm uses the global computation model shown in Figure 1. For simplicity, we assume here that each LP executes on a separate processor. The four local *T*-values define a *state vector* for each LP. State vectors are the smallest unit of transaction between the LP's and the reduction network. They provide atomic snapshots and guarantee that the hardware satisfies certain correctness criteria defined in [25]. Whenever an LP changes a *T*-value, it constructs a state vector with that change and presents it to the reduction network. The reduction network produces a globally reduced state vector by reducing corresponding elements of each state vector and

distributes this to all the LP's. In the case of *T*-values which have more than one component, the reduction is performed on the time component; all other components simply accompany the time component. Accordingly, we define the global counterparts of ρ_i and τ_i as:

$$\rho' = \rho_j \text{ where } timestamp(\rho_j) = MIN_{LP_i}(timestamp(\rho_i))$$

$$\tau' = \tau_j \text{ where } timestamp(\tau_j) = MIN_{LP_i}(timestamp(\tau_i))$$

where $timestamp(\rho_i)$ and $timestamp(\tau_i)$ refer to the time components of the respective triples.

We note that the LP's and the reduction network operate completely asynchronously with each other. As a result, it is likely that state vectors will not be presented at the same time by all the LP's. The reduction network operates in *reduction cycles*. In each cycle, it reduces a complete set of input state vectors to a single output state vector. The algorithm assumes that once a state vector has been presented to the reduction network, it is used repeatedly in successive reduction cycles until a new state vector overwrites it. Also, it assumes that overwriting of state vectors is performed only between reduction cycles. Finally, it assumes that each LP periodically reads the output of the reduction network, i.e., no LP ignores the output for an arbitrary amount of time. These assumptions are satisfied by the particular hardware design we describe in Section 2.2.

The algorithm is shown in Figure 2*. An instance of this algorithm executes for each LP_i. It consists of three concurrent procedures, `TEST`, `RCVMSG` and `CHKACK`. `SENDMSG` is a subroutine used by `TEST` to send messages to other LP's. At each LP_i, η_i is initialized to the timestamp of the first event to be executed, υ_i to ∞ and ρ_i and τ_i to $\{\infty, \Phi, 0\}$ where Φ denotes a null value

2.1.1 TEST

`TEST` is the procedure which uses the two global *T*-values η' and υ'. At each LP, `TEST` determines whether the next event scheduled for that LP is safe. For an event to be safe, its timestamp must be the minimum of all next event times and there must be no unreceived message with a smaller timestamp (if there is such a message, it may ultimately cause an event with a timestamp smaller than the LP's current next event to be scheduled). If the event is indeed safe, it is executed. As a result, an LP may send out messages to other LP's. Since every message that is sent is initially outstanding, the `SENDMSG` subroutine ensures that the smallest unreceived message time reflects this fact.

2.1.2 RCVMSG

The `RCVMSG` procedure is executed whenever an LP receives a message from another LP. `message_time` is the timestamp of the message. Every message causes an event to be scheduled at the receiving LP, with `message_time` as its timestamp. In the case where the newly created event precedes the scheduled next event, the new event becomes the scheduled next event of the LP. The newly received message is inserted into the unacknowledged message list and remains

* In all of the algorithms presented in this paper, [...] represents actions that must be performed atomically and textual nesting is used to indicate the scope of compound statements such as `IF` statements.

```
TEST:     IF    [ η_i = η' AND η_i ≤ υ']                          -- Identify next event in system
          THEN  LC_i := η_i ;                                     -- Advance local clock
                Execute event; Perform SENDMSG if required;       -- May send zero or more messages
                Compute new η_i ;

SENDMSG:  [IF   message_time < υ_i                                -- Update smallest unreceived
           THEN υ_i := message_time];                             -- message time if necessary
          Send the message;
          Add {message_time, message_id} to outstanding message list;

RCVMSG:   [IF   message_time < η_i                                -- Message creates new next event
           THEN η_i := message_time];                             -- update next event time
          Add {message_time, message_id} to unacknowledged message list;

          [IF   ρ_i = {∞, Φ, 0}                                   -- No acknowledgment in progress
           THEN ρ_i := {message_time, message_id}];               -- acknowledge the new message

CHKACK:   IF    [ τ' = ρ_i AND ρ_i ≠ {∞, Φ, 0} ]                  -- Sender has seen acknowledgment
          THEN  Set ρ_i to acknowledge next batch of messages;    -- Possibly {∞, Φ, 0}
                Remove acknowledged batch from unacknowledged message list;

          [IF   ρ' has been sent to this LP                       -- sender sees the acknowledgment
          THEN  τ_i := ρ';
                IF   ρ' is in outstanding message list]
                THEN Remove the acknowledged batch                -- seeing ack for the first time
                           from outstanding message list;
                     IF  timestamp of acknowledged batch = υ_i
                     THEN υ_i := smallest timestamp in outstanding message list;
          ELSE  τ_i := {∞, Φ, 0} ;
```

Figure 2 - Framework Algorithm

there until it is acknowledged. If the LP is not acknowledging any previous messages, RCVMSG immediately initiates the acknowledgment of the newly received message.

2.1.3 CHKACK

CHKACK is responsible for performing acknowledgments of messages through the reduction network. It is executed periodically, as often as possible. Since acknowledgments are performed through the reduction network, they do not interfere with the simulation message traffic in the host communication network. We first briefly present the two enhancements described in [18] which have been incorporated into the CHKACK procedure, viz., the two-phase protocol for acknowledging messages and batched acknowledgments.

Two-phase acknowledgment protocol

In the framework, messages are acknowledged through the reduction network. Since LP's may not observe all of the state vectors emerging from the reduction network, an LP sending an acknowledgment to another cannot assume that the LP to which the acknowledgment is being sent observes the acknowledgment when it emerges from the reduction network. To overcome

this, the protocol uses two phases: in the first phase, the sender of the message being acknowledged determines that the receiver of the message is acknowledging the message and in the second phase, the receiver determines that its acknowledgment has been observed by the sender.

To acknowledge a message sent to it by LP_s, LP_r initiates the first phase by setting its ρ_r to the timestamp and message identifier of the received message. We refer to some ρ_i having a value other than $\{\infty, \Phi, 0\}$ as a *primary acknowledgment* (or simply *acknowledgment*) and the act of setting ρ_i as *submitting a primary acknowledgment*. The timestamp of a primary acknowledgment is the same as that of the message which it is acknowledging. For now, assume that the timestamp of ρ_r is the smallest among all other acknowldegments submitted simultaneously. Thus, ρ' will equal ρ_r. When this is observed by LP_s (and it is guaranteed to ultimately do so, since LP_r does not change ρ_r), it knows that its message to LP_r is being acknowledged and the first phase is complete. LP_s initiates the second phase by setting its τ_s to ρ'. As with ρ_i, we refer to some τ_i having a value other than $\{\infty, \Phi, 0\}$ as a *secondary acknowledgment* and the act of setting τ_i as *submitting a secondary acknowledgment*. Since an LP submits a secondary acknowledgment only after observing a primary acknowledgment, every secondary acknowledgment corresponds to a primary acknowledgment and the timestamp of a secondary acknowledgment is equal to that of its primary acknowledgment. After some time, τ' will equal τ_s and therefore ρ_r. This is ultimately observed by LP_r (because LP_s does not change τ_s) which then knows that LP_s has seen its acknowledgment. LP_r then removes its acknowledgment by changing ρ_r. After some delay, this change is reflected in the output of the reduction network as a change in ρ'. When LP_s observes this change, it knows that LP_r has removed its acknowledgment and so it *relinquishes* the secondary acknowledgment by setting τ_s to $\{\infty, \Phi, 0\}$ and the second phase is complete.

At any time during the two phases described above, a new acknowledgment with a timestamp smaller than ρ_r may be submitted. As a result, ρ' will equal this new acknowledgment. This change in ρ' may occur either before or after LP_s observes the old ρ'. In the first case, the first phase of the protocol is preempted; this situation is the same as having several acknowledgments with the new acknowledgment having the smallest timestamp. When the new acknowledgment completes, ρ_r again becomes the acknowledgment with the smallest timestamp and the first phase is restarted. In the second case, LP_s has observed the primary acknowledgment but the second phase is preempted. LP_s observes that its message is no longer being acknowledged and consequently relinquishes the second acknowledgment by setting τ_s to $\{\infty, \Phi, 0\}$. After the new acknowledgment completes, ρ' reverts to ρ_r. When LP_s observes this, it resumes the second phase by resubmitting its secondary acknowledgment and the second phase ultimately completes. Thus, the algorithm effectively nests acknowledgments.

Batched acknowledgments

From the description above, it is clear that when multiple acknowledgments are submitted at the same time, only the one with the smallest timestamp completes. This serialization of acknowledgments is alleviated by the second enhancement in which a batch of messages is acknowledged using a single physical acknowledgment. This is done by adding a third component to each of ρ_i, τ_i, ρ' and τ'. These *T*-values now consist of a message timestamp, a message identifier and a size field. An acknowledging LP searches its unacknowledged message

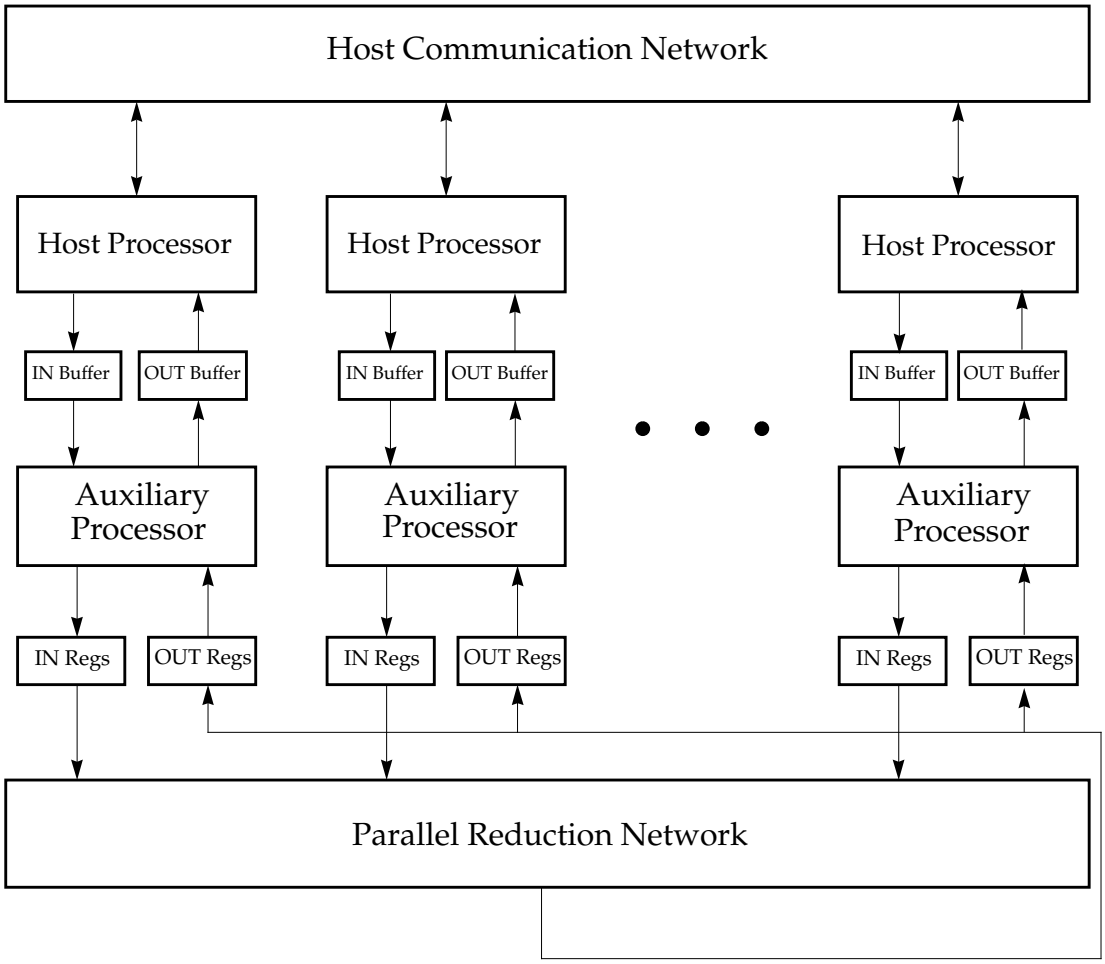


Figure 3 - Hardware Confi guration

list for a batch of received messages with contiguous sequence numbers and sets its ρ_i to {the smallest timestamp in the batch, the message identifi er of the starting message in the batch, the number of messages in the batch}. This enhancement is expected to make the system more robust under heavy loads, as corroborated by simulation studies [28].

The `CHKACK` procedure consists of two parts. The fi rst`IF` statement implements the part of the two-phase protocol executed by an LP which receives a message while the second `IF` statement implements the part executed by an LP which sends a message.

2.2 Framework hardware

We briefly describe an implementation of the reduction model shown in Figure 1. A detailed description can be found in [25]. The hardware confi guration is shown in Figure 3. The main components of this hardware are the *parallel reduction network* (PRN) and the *auxiliary processors* (AP). For each *host processor* (HP), there is an AP which is a general purpose processor such as the HP itself. The HP's perform all simulation specifi c tasks (executing events, sending

and receiving messages, saving state, etc.) while the AP's perform all of the synchronization tasks (i.e., they are responsible for submitting state vectors to the PRN, reading the output of the PRN and executing synchronization algorithms). When an HP causes its state to change (by executing an event, receiving a message, etc.) it communicates this change to its AP. The AP incorporates such changes into state vectors which it submits to the PRN for reduction. The AP also reads the globally reduced output state vector produced by the PRN and makes selected subsets of this available to the HP.

The interface between the HP and the AP consists of two unidirectional channels (implemented as buffers). For information flowing from the HP to the AP, it is required that none of it be lost and that the AP process the information in the order sent out by the HP. For these two reasons, the IN Buffer is a FIFO. In the other direction, the only requirement is that the most recent version of the AP's output be available to the HP. Accordingly, the OUT Buffer consists of a single cell buffer, which is repeatedly overwritten by the AP each time it presents new data to the HP. We note here that this confi guration is one of many possible implementations of the global computation model of Figure 1. In particular, the choice of having dedicated processors for simulation and synchronization tasks and their asynchronous operation with respect to each other introduces latencies in the data path which are absent in the model of Figure 1. This requires minor adjustment of synchronization algorithms which are based on the model of Figure 1, as will be seen in Section 5.

As mentioned earlier, the PRN and the AP's operate asynchronously. Occasionally, the AP's present state vectors (not necessarily all at the same time) to the PRN for reduction. The PRN operates in reduction cycles, reducing a set of state vectors, one from each processor, to a single output state vector in each cycle. Subsequent reduction cycles re-use state vectors for those AP's which do not present new state vectors. To satisfy these requirements, the design includes a custom interface between the AP's and the PRN. The AP's write new state vectors to the IN registers and read globally reduced state vectors from the OUT registers. Likewise, the PRN reads state vectors from the IN registers, reduces them and writes the global output to the OUT registers. The IN and OUT registers are comprised of three sets of registers each which provide the isolation between the AP and the PRN. The detailed operation of these register interfaces is described in [25].

The PRN is a binary tree of height $\log_2 n$, where n is the number of processors. Figure 4 shows a PRN for eight processors. Each node of the tree contains a general purpose ALU which performs reduction operations on its two operands. The reductions are performed in parallel across all the ALU's. The PRN interfaces with each processor through the IN and OUT registers. At each LP, the IN registers present a state vector (which may be the same as the one used in the previous cycle) to the PRN. The PRN starts the reduction by reading the fi rst elements of each state vector and reducing them pair-wise at the top row of ALU's. These ALU's then pass the reduced values down to the second row of ALU's. While the second row of ALU's reduces these values, the top row reduces the second elements of each state vector. Thus, the PRN operates in a pipelined fashion. The time required for a set of values to pass through a single PRN stage is called the *minor cycle time*, while the time required for the top row ALU's to read all the elements of the state vectors is called the *input cycle time*. Note that an input cycle consists of m minor cycles, where m is

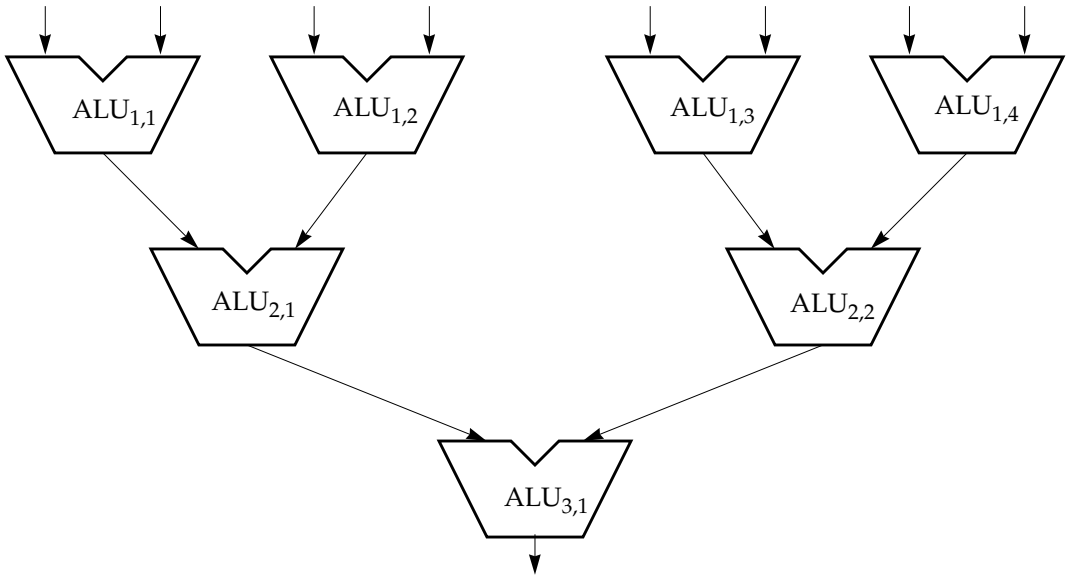


Figure 4 - A Parallel Reduction Network

the size of the state vectors. Because of the pipelined nature of the PRN, it takes $\log_2 n$ minor cycles for the global reduction of the first element of a set of state vectors and thereafter, reductions of the remaining m-1 elements emerge from the PRN one minor cycle apart. Thus, a reduction cycle, which is the total time required to produce a globally reduced output state vector from a set of n input state vectors of size m each, is $(\log_2 n + (m\text{-}1))*c$, where c is the minor cycle time. We note that in this implementation, a new reduction cycle is initiated at the end of every input cycle (or m minor cycles). For $n > 2$, this means that a new reduction cycle is started before the current one completes. Since speed of reduction is the primary design goal of this hardware, it is important that c be small. In the prototype hardware described in [24], c is expected to be around 150 nanoseconds giving a reduction cycle time of 1.2 microseconds for 32 processors with 4-element state vectors.

Besides the ALU, each PRN node has some additional logic to accommodate T-values which have multiple components (such as ρ_i and τ_i). Each value submitted to the PRN for reduction is accompanied by a *tag* which is expected to carry the other components of a T-value. At each ALU, the tags of the two operands are brought to a selector switch. The choice of which tag is propagated down to the next stage is made by a control signal from the ALU. For example, if the ALU performs a selective operation such as a *minimum* on the values, the tag of the smaller of the two values will proceed downward. In non-selective operations, the choice can be arbitrary but should be deterministic.

3 Making the framework aggressive

It has been established in [22] that the framework algorithm will operate correctly with non-aggressive protocols. Aggressive protocols have two attributes which are not seen in non-aggressive protocols: (i) the logical clocks of the LP's may move backward and (ii) LP's use

antimessages (when they propagate the results of aggressive processing to other LP's). In this section, we present our algorithm which is a modified version of the algorithm of Figure 2, modified to incorporate the considerations for the two attributes just mentioned. For exposition, our choice of aggressive protocol is Time Warp [10].

The most important synchronization value required by the LP's in Time Warp is GVT. GVT is required by the LP's during a simulation for several reasons: fossil collection in systems with limited state saving space, performing I/O, collecting statistics and detecting termination conditions. To date, GVT computation has been an expensive (time consuming) operation. This is an area in which the framework can significantly benefit aggressive protocols, by rapidly computing and providing the value of GVT. In order to correctly compute GVT using the framework hardware, we have modified the synchronization algorithm of Figure 2 to correctly maintain two local *T*-values at each LP, the global reduction of which provides GVT.

3.1 Aggressive Framework Algorithm

The Aggressive Framework Algorithm (AFA) listed in Figure 5 is designed to asynchronously compute GVT in a Time Warp system. It assumes the computation model of Figure 1. As mentioned in Section 2.2, the framework hardware differs slightly from this model. We will present AFA adapted for the framework hardware in Section 5 and prove its correctness in Section 6.

We assume familiarity with the Time Warp protocol and associated terminology. By definition, GVT is the minimum of two*T*-values: the smallest of the logical clocks of all the LP's and the smallest unreceived message time among all the LP's. The original framework algorithm maintains the minimum unreceived message time, υ', but does not maintain the smallest logical clock value. Instead, it maintains the smallest next event time, η'. In a non-aggressive system, we can distinguish between two local *T*-values: σ_i, which is the timestamp of the event being executed (or just executed) by LP_i (i.e., the logical clock value of LP_i) and η_i, which is the timestamp of the next event that LP_i intends to execute. In such a system, LP_i will not execute its next event unless it can ascertain its safety. Each LP_i submits its η_i for reduction rather than its σ_i, so that the LP's can identify the next safe event(s). In an aggressive system, since LP's do not wait to ascertain the safety of an event before executing it, we consider only σ_i, the logical clock value of LP_i. Thus, the algorithm of Figure 5 maintains the following two *T*-values:

$$\sigma' = MIN_{LP_i}(\sigma_i) \quad \text{: the minimum of all the logical clocks}$$

$$\upsilon' = MIN_{LP_i}(\upsilon_i) \quad \text{: the minimum unreceived message time}$$

and GVT is defined as $GVT = MIN(\sigma', \upsilon')$.

AFA has two concurrent procedures, `PROCESS` and `DO_ACK`. The `PROCESS` procedure invokes three subroutines, `SEND_MSG`, `RCV_MSG` and `ROLLBACK` as required. One change in AFA from the framework algorithm of Figure 2 is that `RCV_MSG` is now called as a subroutine by `PROCESS` instead of being a concurrent procedure invoked when a message is received. This is

```
PROCESS:   IF     there are events in the events list
           THEN   σ_i := timestamp of next event;
                  Execute event; Perform SEND_MSG if required;
                  Optionally save state;
           WHILE  there are newly received messages with timestamps < σ_i
                  Perform ROLLBACK;
           FOR    each new message
                  Perform RCV_MSG;
           Optionally collect fossils;

SEND_MSG:  [IF    message_time < υ_i
           THEN   υ_i := message_time]
           Add message to outstanding list;
           Send the message;
           IF     message_sign > 0
                  Add antimessage to output list;

RCV_MSG:   IF     message_sign > 0
           THEN   Insert event into events list;
           ELSE   Delete the positive event;

           [IF    ρ_i = {∞, Φ, 0}
           THEN   ρ_i := (message_time, message_id, 1)
           ELSE   Add message to unacknowledged message list];

ROLLBACK:  σ_i := rollback time;
           Roll back the execution of all events with time > rollback time;
           Restore state from the last time it was saved before rollback time;
           Rebuild the state up to rollback time if required;
           FOR    each antimessage with timestamp > rollback time
                  Perform SEND_MSG;
                  Delete it from the output list;

DO_ACK:    IF     [τ' = ρ_i AND ρ_i ≠ {∞, Φ, 0}]          -- sender has seen ack
           THEN   Remove next batch to be acknowledged
                      from unacknowledged message list;   -- if one exists
                  Set ρ_i to acknowledge this batch;

           [IF    ρ' has been sent to this LP
           THEN   τ_i := ρ';
                  IF     ρ' messages are in outstanding message list]
                  THEN   Remove the acknowledged batch from outstanding message list;
                         IF   timestamp of acknowledged batch = υ_i
                         THEN υ_i := smallest timestamp in outstanding message list
           ELSE   τ_i := {∞, Φ, 0};
```

Figure 5 - Aggressive Framework Algorithm

because we assume that messages do not preempt event execution. We show later how to incorporate event preemption into this algorithm.

The PROCESS procedure implements all Time Warp specifi c operations. It unconditionally executes events on the events list. The state of the LP is saved periodically. After the execution of each event, the procedure checks for the arrival of new messages during this execution. If any new

messages have arrived in the LP's logical past (i.e. they have to be inserted in the events list before the event just executed - such messages are called *stragglers*), then PROCESS calls ROLLBACK, to perform the rollback. On a rollback, the logical clock of the LP is rolled back to the timestamp of the straggler. In the ROLLBACK subroutine, we have chosen to employ aggressive cancellation [26]. We show later that lazy cancellation is easily accommodated in this algorithm. At the end of a rollback, the PROCESS procedure checks to see if any new stragglers have arrived during the rollback process which will cause the LP to roll back further. If so, ROLLBACK is called again. Thus, ROLLBACK is invoked repeatedly until all of the newly received messages are in the LP's logical future (i.e., the LP has rolled back far enough). These messages are now incorporated into the events list of the LP in the RCV_MSG subroutine. The DO_ACK procedure is the same as the CHKACK procedure in Figure 2; it performs acknowledgments of messages using the PRN.

In order to maintain υ_i correctly, the timestamps of antimessages must also be part of the GVT computation. To see why, consider a system of two LP's with LP_0 having σ_0 at 1000, LP_1 having σ_1 at 1500 and no messages in transit. Therefore, GVT has a value of 1000. Now, LP_0 sends LP_1 an antimessage with timestamp 1000 and then proceeds to execute its next event with timestamp 2000. While the antimessage is in transit, if antimessages are not acknowledged, we have σ_0 at 2000, σ_1 at 1500 and both υ_0 and υ_1 at ∞, giving a GVT of 1500. When the antimessage is finally received by LP_1, σ_1 falls to 1000, bringing GVT down to 1000. This is an error since GVT must be strictly non-decreasing. The error occurs because the smallest timestamp in the system, which is the timestamp of the antimessage, is momentarily absent from the GVT computation algorithm. The problem is solved by requiring antimessages also to be acknowledged. To do this, the ROLLBACK procedure calls SEND_MSG to send the antimessages. This guarantees that the timestamps of antimessages are accounted for in the value of υ_i. In addition, we see that the received message is marked for acknowledgment independent of its sign in RCV_MSG (i.e. antimessages are also acknowledged).

3.2 Event preemption and lazy cancellation

We indicate how two optional features of Time Warp (event preemption and lazy cancellation) can be incorporated into AFA. In AFA as presented in Figure 5, the event processing procedure (PROCESS) checks to see if new messages have arrived after the execution of each event. When one or more of the messages received during the execution of an event have a smaller timestamp than that of the event, an efficient implementation will preempt the execution of the event (since it will be rolled back anyway) rather than wait for the execution to complete. To incorporate this into AFA, the RCV_MSG subroutine is activated as a concurrent procedure with PROCESS and DO_ACK whenever the LP receives a message. In addition, at the start of the RCV_MSG procedure, a check is made to see if the received message is in the LP's logical past and if so, the ROLLBACK procedure is called. In effect, each iteration of the WHILE and FOR loops in PROCESS is transformed into a concurrent invocation of RCV_MSG.

With lazy cancellation, upon rolling back, an LP sends out antimessages only as required. In Figure 5, the ROLLBACK subroutine implements aggressive cancellation, wherein antimessages are sent out during a rollback regardless of whether they are needed or not. To implement lazy cancellation, the FOR loop is removed from the ROLLBACK subroutine. Instead, the PROCESS

procedure must now perform a check after every event execution, to see if any antimessages need to be sent out. If so, the `SEND_MSG` subroutine is called to actually send the antimessage.

4 Correctness

In this section, we establish the correctness of AFA shown in Figure 5. Throughout this paper we are concerned only with temporal correctness. We assume that the simulators under discussion are functionally accurate. By temporal correctness we mean that none of the committed events violate causality constraints. The following criteria defi ne (temporal) correctness of a parallel simulation:

i) each LP must ultimately execute events in strictly non-decreasing timestamp order [9]

ii) the simulator must make progress if the application being simulated makes progress

The second criterion incorporates the concepts of freedom from deadlock as well as termination.

4.1 Correctness of a Time Warp system

For a Time Warp system, it can be proven that the value of GVT never decreases, by considering all of the possible ways in which GVT may change [10]. Hence, GVT defi nes a commitment horizon, i.e. events with timestamps lower than GVT can be committed. Other uses of GVT are fossil collection, gathering statistical data and detecting the occurrence of termination conditions. It is therefore reasonable to assume that every Time Warp LP needs to know the value of GVT periodically. In a typical Time Warp implementation, this is done by actually computing it during the simulation.

An *ideal time warp system* is one in which the LP's obtain all the information they need instantaneously. For instance, in an ideal system, when an LP receives a message sent to it by another, the sending LP becomes aware of the fact that the message has been received at the same instant the receiving LP receives the message. Obviously, no implementation can be ideal but every implementation has a corresponding ideal system. Timestamps often remain in an implementation for some time after they have disappeared from the ideal system. Returning to our previous example, typically, the sending LP is notifi ed of the receipt of the message by some form of acknowledgment. Since this acknowledgment takes time to accomplish, the message remains outstanding for a longer period of time in the implementation than in the ideal system. This *lag* is especially true of implementations in which the GVT computation proceeds asynchronously with the simulation, i.e., the simulation is not suspended when GVT is computed. As a result, the GVT computed by an implementation is usually an approximation of the value of GVT in the corresponding ideal system. At any instant of real time t, we distinguish between the actual value of GVT, $GVT_a(t)$, which is the value of GVT in the ideal system at time t (i.e. the smallest timestamp in the ideal system at time t), and the computed value of GVT, $GVT_c(t)$, which is the value of GVT made available by the Time Warp implementation at time t.

Time Warp has the property that all of the LP's are guaranteed to have correctly simulated the system until simulated time equal to GVT. All of the simulation beyond GVT is speculative. In terms of our notation, at real time t, all events with timestamps less than GVT_a (t) have been correctly simluated at each LP. Since GVT_c (t) is used to approximate GVT_a (t) (and therefore to commit events), its value must not exceed that of GVT_a (t) for all times t. The correctness criteria for a Time Warp implementation (actually any GVT computation algorithm) can now be stated as below:

Criterion 1: $GVT_c(t) \leq GVT_a(t)$ for all times t.
Criterion 2: As t increases, $GVT_c(t)$ increases if $GVT_a(t)$ increases.

4.2 Useful properties

Before proving the correctness of AFA, we present some properties of the framework hardware and AFA. These will be used in the proofs that follow. A short name is associated with each property and will be used along with the property number to refer to the property.

4.2.1 Properties of the framework hardware

P1 *no loss*

No communication from the HP to the AP is lost.

P2 *reduction operation*

Let $\mathbf{V}_1, \mathbf{V}_2, \ldots \mathbf{V}_n$ be the state vectors submitted by the n LP's such that $\mathbf{V}_i = \langle V_i^1, V_i^2, \ldots, V_i^m \rangle$ where m is the size of each LP's state vector. Let θ_i $1 \leq i \leq m$ be binary associative operators. Then in one reduction cycle the PRN computes $\mathbf{G} = \langle G^1, G^2, \ldots, G^m \rangle$ where $G^i := \theta_i \langle V_1^i, V_2^i, \ldots, V_n^i \rangle$, $1 \leq i \leq m$ and $\mathbf{G}$ is the output state vector of the PRN. For the algorithms in this paper, $m = 4$ and $\theta_1 = \theta_2 = \theta_3 = \theta_4 = MIN$.

P3 *input cycle time*

A reduction cycle (i.e. a fresh computation of GVT) is started every δ time units.

P4 *reduction cycle time*

Each reduction cycle takes Δ time units to complete and $\Delta \geq \delta$*.

* Strictly speaking, due to the asynchronous design of the PRN, a reduction cycle is started once every *approximately* δ time units. Similarly, each reduction cycle takes *approximately* Δ time units. However, this variation does not affect the proofs.

4.2.2 Properties of AP software

The software executing on the AP's should have the following properties:

P5 *periodic read*

Each AP (and therefore each LP) reads the output of the PRN periodically, i.e. no LP will ignore the output of the PRN for an arbitrary amount of time.

P6 *finite delay*

No communication in a FIFO remains unprocessed by the AP for an arbitrary amount of time. This assumes that the communications are enqueued at an average rate which is lower than the average rate at which the communications are processed.

4.2.3 Properties of AFA

P7 *set clock on event*

An LP sets its σ_i to the timestamp of an event before it executes that event.

P8 *set unreceived message time*

When an LP sends a message, if its υ_i is greater than the timestamp of the message, its υ_i will be set to the timestamp of the message; otherwise, there is no change in its υ_i.

P9 *update unreceived message time*

When LP_i receives the primary acknowledgment for a message which has timestamp υ_i, that message is removed from its outstanding message list and its υ_i is set to the smallest among the timestamps of the remaining messages in its outstanding message list.

P10 *unreceived is minimum*

At any LP_i, υ_i is equal to the smallest of the timestamps of the messages in its outstanding message list. This follows directly from P8 (set unreceived message time) and P9 (update unreceived message time).

P11 *maintain unreceived message time*

When an LP sends a message *M*, its υ_i is less than or equal to the timestamp of *M* (from P8 (set unreceived message time)). In addition, it will remain so until the LP observes the primary acknowledgment for *M*.

P12 *set clock on message*

When an LP receives a message, its σ_i is made less than or equal to the timestamp of the message before the acknowledgment of the message is initiated.

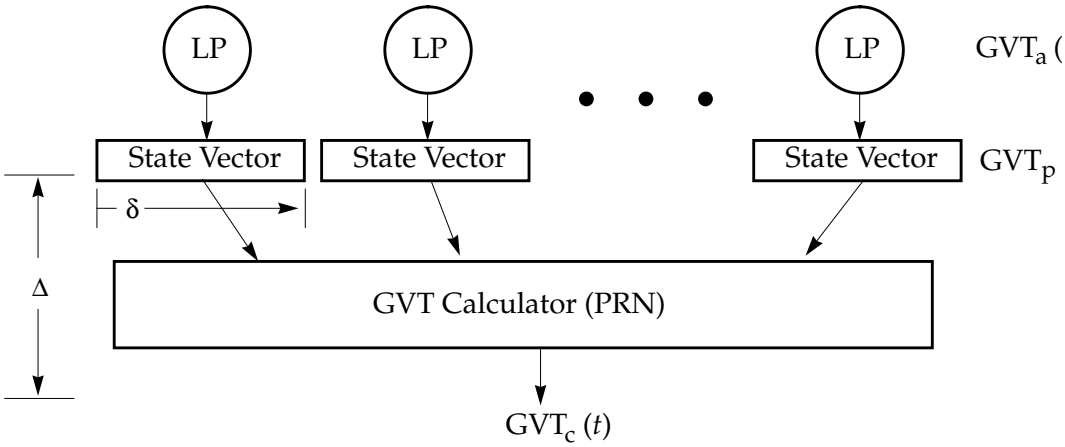


Figure 6 - GVT Computation Model for AFA

4.2.4 Properties of the `DO_ACK` procedure of AFA

P13 *unique selection*

Given a set of primary acknowledgments submitted to the PRN, there is one among them (which also has the smallest timestamp among them, by P2 (reduction operation)) that is selected by the PRN.

P14 *secondary acknowledgment*

If an LP observes a primary acknowledgment being sent to it, it submits the corresponding secondary acknowledgment within a fi nite, bounded amount of time. Otherwise, its ρ_i is set to $\{\infty, \Phi, 0\}$.

P15 *persistent primary acknowledgment*

Once an LP submits a primary acknowledgment to the PRN, that acknowledgment is removed only when the acknowledging LP observes the corresponding secondary acknowledgment.

4.3 Correctness of AFA

AFA basically maintains two *T*-values at each LP_i, corresponding to the two component values of GVT: σ_i, the logical clock and υ_i, the smallest unreceived message time. In addition, AFA also performs acknowledgments of messages using two more *T*-values: ρ_i and τ_i. These four *T*-values defi ne a*state vector* for each LP. The model of GVT computation employed by the algorithm is shown in Figure 6. Each LP presents a new state vector to the GVT Calculator whenever an event occurs which causes a change in GVT_a (*t*) (for example, executed event, received message, sent message, etc.). The GVT Calculator (which is the PRN described in Section 2.2) takes these state vectors and computes GVT_c (*t*+Δ). We show next that AFA satisfi es the two correctness criteria of Section 4.1. Note that properties P2 through P4 are true for the model of Figure 6. In addition, recall that this model assumes that each LP periodically reads the output of the reduction network (i.e., the GVT Calculator). Consequently, P5 is true as well.

4.3.1 Criterion 1

Define $GVT_p(t)$ to be the value of GVT that can be *potentially* calculated at the input side of the GVT Calculator (Figure 6), i.e., given the set of state vectors presented to the GVT Calculator at time t, $GVT_p(t)$ takes on the value of GVT defined by these state vectors. To show that AFA satisfies criterion 1, we prove three lemmas below.

LEMMA 1 If a reduction cycle is started at time s, and if $GVT_p(s) \le GVT_a(s)$, then $GVT_c(t) \le GVT_a(t) \forall t$.

PROOF If a reduction cycle is started at time s, its result becomes available at the output of the GVT Calculator at time $s+\Delta$ (P4 (reduction cycle time)). By P3 (input cycle time), the next reduction cycle is started at time $s+\delta$. Since this cycle also takes Δ time units to complete, a fresh result becomes available at time $s+\delta+\Delta$. Continuing thus, we see that a fresh (not necessarily different) value of $GVT_c(t)$ is computed every δ units of time. Thus, $GVT_c(t)$ is a step function with the separation in time between steps being an integral multiple of δ. Consider the two (arbitrarily chosen) consecutive steps above which form the time interval $[s+\Delta, s+\delta+\Delta)$. $GVT_a(t)$ cannot decrease in this interval (because of its non-decreasing nature, by assumption of the functional correctness of AFA). It follows that $GVT_c(t) \le GVT_a(t)\ \forall t$ if $GVT_c(s+\Delta) \le GVT_a(s+\Delta)$ for those times $s+\Delta$ at which a fresh result is computed at the output of the GVT Calculator. By design of the GVT Calculator, $GVT_c(s+\Delta) = GVT_p(s)$ and by assumption, $GVT_p(s) \le GVT_a(s)$. Thus, $GVT_c(s+\Delta) \le GVT_a(s)$. By the non-decreasing property of $GVT_a(t)$, $GVT_a(s) \le GVT_a(s+\Delta)$. Hence the claim. ■

We digress briefly to introduce some notation. All of the T-values considered here (i.e., σ_i, υ_i, etc.), as well as $GVT_a(t)$, $GVT_p(t)$ and $GVT_c(t)$ are functions of real time which is continuous. However, these entities do not change continuously with time, but rather at discrete points in time. If an entity, f, changes its value at time instant t, we use $f(t)$ to refer to the new value of f after the change at time t and $f(t^-)$ to refer to the old value of f before the change at time t. Also, we use the time parameter throughout the paper only for $GVT_a(t)$, $GVT_p(t)$ and $GVT_c(t)$. For the other entities (especially σ_i and υ_i), we use the time parameter when it is required for clarity and omit it otherwise.

AFA is designed such that any action by an LP which may cause $GVT_a(t)$ to change is preceded by a change in the T-values being presented to the GVT Calculator which reflects that change. For instance, before sending a message, the timestamp of the message is incorporated in the υ_i of the sending LP. Similarly, when an LP receives a message, the timestamp of the message is incorporated into its σ_i before the message is acknowledged. Such being the case, the following invariant is maintained by the algorithm:

LEMMA 2 At any instant t, there is at least one LP_k with $\sigma_k(t)$ or $\upsilon_k(t)$ less than or equal to $GVT_a(t)$.

PROOF We use an induction proof here. At the start of the simulation (time t_0), assuming the state vectors are initialized correctly, the LP (LP_k) which has the event with the smallest timestamp (G_0) scheduled as its first event will have its σ_k equal to $G_0 = GVT_a(t_0)$ and the basis is

true. For the inductive step, consider the different ways in which $GVT_a(t)$ changes from G_i to G_{i+1} at time t_{i+1} i.e., $GVT(t_{i+1}^-) = G_i$ and $GVT(t_{i+1}) = G_{i+1}$.

i) LP_l executes the event with timestamp G_i at time t_{i+1} and now G_{i+1} is the timestamp of an event at LP_k (note k may equal l). By P7 (set clock on event), $\sigma_k(t_{i+1}) = G_{i+1}$.

ii) LP_k executes the event with timestamp G_i at time t_{i+1} and sends a message with this timestamp so that G_{i+1} is the timestamp of a message in transit (i.e., the message has not yet been scheduled as a future event at the intended receiver). By P11 (maintain unreceived message time), $\upsilon_k(t_{i+1}) = G_{i+1}$

iii) A message from LP_l is received by LP_k and becomes the event whose timestamp is G_{i+1}. At t_{i+1}, the two-phase acknowledgment of the message completes so that $\upsilon_l(t_{i+1})$ may be greater than G_{i+1}. By P12 (set clock on message) and the fact that G_{i+1} is the smallest timestamp in the ideal system at t_{i+1}, σ_k becomes equal to G_{i+1} before the message is acknowledged. Thus $\sigma_k(t_{i+1}) = G_{i+1}$.

iv) The message with timestamp G_i is received by some LP and at t_{i+1}, some other message in transit has timestamp G_{i+1}. This case is similar to case (ii) above. ■

LEMMA 3 $GVT_p(t) \le GVT_a(t)\ \forall t$

PROOF $GVT_p(t)$ is defined as the GVT computed from the state vectors presented to the GVT Calculator at time t, i.e.:

$$GVT_p(t) = MIN_{LP_i}(\sigma_i(t), \upsilon_i(t))$$

which implies that $GVT_p(t) \le \sigma_k(t)$ and $GVT_p(t) \le \upsilon_k(t)$ for all LP_i. The lemma follows immediately from the truth of the invariant of Lemma 2 and these inequalities. ■

THEOREM 1 $GVT_c(t) \le GVT_a(t)\ \forall t$

PROOF From Lemma 1, Theorem 1 is true if $GVT_p(t) \le GVT_a(t)$ for those times t, at which the GVT Calculator starts a reduction cycle. From Lemma 3, $GVT_p(t) \le GVT_a(t)\ \forall t$. ■

Note the proof above covers the case when old state vectors are overwritten by new ones before they are processed by the PRN. In this case, the set of values of $GVT_p(t)$ processed by the PRN is a subset of the set of all the values taken on by $GVT_p(t)$. Since Lemma 2 is valid for all values of $GVT_p(t)$, in the case of overwrites it will be valid for those values which are processed by the PRN and Theorem 1 is true.

4.3.2 Criterion 2

Since acknowledgments are key to the progress of υ' and therefore $GVT_c(t)$, we first show that the `DO_ACK` procedure acknowledges messages correctly.

LEMMA 4 The `DO_ACK` procedure acknowledges messages such that if $GVT_a(t)$ increases with t, every message is eventually acknowledged.

PROOF We consider acknowledgments under two situations:

Case I: Fixed set of acknowledgments

Assume first that a non-empty set of acknowledgments (A) submitted to the GVT Calculator does not change over the entire time taken to perform a single two-phase acknowledgment (Section 2.1.3). By P13 (unique selection), one of these (call it P_W, an acknowledgment from LP_R to LP_S) will appear in the state vector output by the GVT Calculator. In addition, P_W will have the smallest timestamp in the set. By P15 (persistent primary acknowledgment) and our assumption of an unchanging acknowledgment set, P_W will continue to appear in the GVT Calculator's output. P5 (periodic read) guarantees that LP_S will observe P_W and by P14 (secondary acknowledgment), will submit the corresponding secondary acknowledgment (call it S_W from LP_S to LP_R) in finite time. P14 (secondary acknowledgment), P15 (persistent primary acknowledgment), P5 (periodic read) and our assumption of an unchanging acknowledgment set guarantee that after some finite time, the only secondary acknowledgment being submitted to the GVT Calculator will be S_W. After time Δ, S_W will appear in the output of the GVT Calculator and will continue to do so because P_W also appears in the output. P5 (periodic read) guarantees that LP_R will observe S_W and by P15 (persistent primary acknowledgment), will remove its primary acknowledgment, P_W. After time Δ, P_W will no longer appear in the GVT Calculator's output. When this fact is observed by LP_S, it will stop sending its secondary acknowledgment, S_W (P14 (secondary acknowledgment)). Thus, when a set of acknowledgments is submitted to the GVT Calculator and this set does not change for the duration of one acknowledgment, then a unique one of these (having the smallest timestamp) will complete in finite time. In the rest of this proof and the paper, if P_X is a primary acknowledgment, we refer to the two-phase process initiated by the submission of P_X as the *P_X acknowledgment*.

Case II: Changing set of acknowledgments

Consider the case where the set of acknowledgments (A) changes when the two-phase protocol is in progress. By P15 (persistent primary acknowledgment), the only change can be the submission of a new acknowledgment (call it P_N) by some LP_N. Thus we have a new set of acknowledgments, $A' = A \cup P_N$. This new arrival can have one of two effects: it either replaces P_W as the winning acknowledgment or it does not. The second case is uninteresting as it does not affect the two-phase acknowledgment described earlier (i.e., we can apply the argument of Case I to show that the P_W acknowledgment will complete in finite time). When P_N replaces P_W in the GVT Calculator's output, we say the P_W acknowledgment is *preempted*. P_N may preempt either the first phase or the second phase of the P_W acknowledgment. We consider each separately.

Case II(a): First phase preempted

The first phase of the P_W acknowledgment completes when LP_S observes P_W. If the first phase is preempted, then LP_S has not observed P_W. The effect of this is that the first phase has not been started at all. Since P_N is the winning akcnowledgment now, the P_N acknowledgment goes through to completion in some finite time. When LP_N stops sending P_N, the set of acknowledgments reverts to A from A'. Now the situation is exactly that before the arrival of P_N. Thus, the P_W ackowledgment will restart and complete in finite time.

Case II(b): Second phase preempted

If P_N replaces P_W after LP_S has observed P_W (and hence submitted S_W), the second phase of the P_W acknowledgment is preempted. This may happen either after or before LP_R has observed S_W in the output of the GVT Calculator. In the first case, all that remains of the second phase is for LP_S to stop sending S_W. By P14 (secondary acknowledgment), this will indeed happen in a bounded amount of time since P_W has been replaced by P_N. Since LP_R has already seen S_W, it will no longer submit P_W and the P_W acknowledgment is complete. In the second case, after LP_S observes that P_N has replaced P_W (which it eventually will, by P5 (periodic read)) it will stop sending S_W (P14 (secondary acknowledgment)). The P_N acknowledgment now goes to completion. At the end of this, when LP_N removes P_N, P_W again becomes the winning acknowledgment (the set of acknowledgments is again A). After some finite time, LP_S observes this and resubmits its secondary acknowledgment, S_W (by P14 (secondary acknowledgment)). The P_W acknowledgment now completes in finite time.

Note the arguments of Case II may be applied recursively (i.e., if a preempting acknowledgment is itself preempted). Thus the two-phase acknowledgment protocol allows acknowledgments to be preempted such that they are resumed later (i.e., the acknowledgments are properly nested). A necessary condition for an acknowledgment to preempt another is that its timestamp must be less than or equal to that of the one in progress. Consequently, for an acknowledgment to be infinitely delayed, there must be an infinite number of messages with timestamps less than or equal to that of the acknowledgment in progress. Since at any time t, the logical clocks of LP's (and therefore the timestamps of messages) can only go back as far as $GVT_a(t)$, we conclude that a necessary condition for infinite nesting is that $GVT_a(t)$ does not advance with t. We have thus proved that if $GVT_a(t)$ advances with t, every acknowledgment will complete in a finite amount of time. ■

THEOREM 2 As t increases, $GVT_c(t)$ increases if $GVT_a(t)$ increases

PROOF We prove this theorem by contradiction. Assume that $GVT_c(t) = GVT_c(t_0)$ for all $t > t_0$ while $GVT_a(t_1) > GVT_a(t_0)$ for some $t_1 > t_0$. This must be due to a timestamp that persists indefinitely in the implementation even after it no longer exists in the ideal system. Since the timestamp of an event being executed by an LP is always reflected in $GVT_c(t)$ through its σ_i prior to the execution of the event (P7 (set clock on event)), it follows that this residual timestamp cannot be that of an event and must consequently be the timestamp of an unreceived message.

The fact that GVT_a (t) advances with t means that there are only a finite number of unacknowledged messages at any LP with a given timestamp. Lemma 4 guarantees that all of these messages are acknowledged in finite time. P9 (update unreceived message time) then ensures that an LP's υ_i can remain at any one value for only a finite amount of time. The assumption is thus contradicted. ■

In summary, we have proved that AFA satisfies the two correctness criteria for any GVT computation algorithm:

i) The computed GVT is always less than or equal to the actual value of GVT

ii) The computed GVT advances if the simulation advances.

We have thus described a correct method of rapidly computing and disseminating critical synchronization information using the framework hardware to support aggressive protocols.

5 AFA on the framework hardware

AFA as shown in Figure 5 assumes that all of the procedures (synchronization as well as application-specific) for an LP are executed on a single processor (recall we assume that each LP executes on its own processor) and the processors communicate directly with the PRN. The framework hardware replaces a single processor with two processors: an HP to execute application specific tasks and an AP to execute the synchronization tasks. In order to execute AFA of Figure 5 on the framework hardware, the synchronization aspects have to be separated from the application specific ones. Figure 7 shows AFA2P, which is basically AFA separated into two algorithms, one to be executed on each HP and one on each AP. We make the assumption that each LP is allocated its own HP-AP pair of processors in order to simplify the description of the algorithm as well as the correctness proofs in the next section. The algorithm can easily be extended to allow more than one LP per pair of processors. We use HP_i and AP_i to refer to the HP-AP pair on which LP_i is executing. AFA2P is derived from AFA by off-loading all of the synchronization tasks from the HP to the AP. In AFA2P, the HP performs only Time Warp specific operations such as event execution, sending and receiving messages, state saving, rolling back, etc., while the AP performs all of the operations required to maintain the *T*-values. Accordingly, the HP maintains the logical clock, the events list and the output list (list of antimessages) while the AP maintains the outstanding message list and the unacknowledged message list. As mentioned in Section 2.1, each message in the system is given a globally unique identifier. These identifiers are constructed such that the identifiers form contiguous sequences between each ordered pair of communicating LP's. The AP maintains its lists sorted in ascending order of these sequence numbers in the message identifiers. The AP also maintains the four *T*-values: σ_i, υ_i, ρ_i and τ_i.

Recall that the communication channel between an HP and its AP is functionally a FIFO. An HP communicates with its AP by inserting a tagged communication into the FIFO with the tag indicating the nature of the communication. In the algorithm for the HP, this is indicated by the statement `Enqueue (tag, value1, value2, ...)`. The AP is responsible for three major tasks:

```
HOST_PROC: IF     there are events in the events list
           THEN   local_clock = timestamp of next event
                  Enqueue (NEW_CLOCK, local_clock);
                  Execute event; Perform SEND_MSG if required;
                  Optionally save state;
           WHILE  there are newly received messages with timestamps < local_clock
                  Perform ROLLBACK;
           FOR    each new message
                  Perform RCV_MSG;
           Optionally collect fossils;

SEND_MSG:  Enqueue (SENT_MSG, message_time, message_id);
           Send the message;
           IF     message_sign > 0
                  Add antimessage to output list;

ROLLBACK:  local_clock := rollback time;
           Enqueue (NEW_CLOCK, local_clock);
           Roll back the execution of all events with time > rollback time;
           Restore state from the last time it was saved before rollback time;
           Rebuild state up to rollback time if required;
           FOR    each antimessage in output list with time > rollback time
                  Perform SEND_MSG;
                  Delete it from the output list;

RCV_MSG:   IF     message_sign > 0
           THEN   Insert corresponding event into events list;
           ELSE   Delete the positive event;
           Enqueue (RCVD_MSG, message_time, message_id);
```

Figure 7 - AFA2P: Host Processor Algorithm

- it must read the global output of the PRN and pass on a portion of the global state vector to the HP
- it must process as soon as possible the tagged communications that its HP inserts into the FIFO (since speed of reduction is critical)
- it must perform acknowledgments of messages through the PRN

It is important that each of these tasks is performed frequently and none gets delayed arbitrarily. To satisfy these requirements for the AP, we chose to organize the AP's algorithm as a single loop. The AP begins each iteration by reading the PRN output and forwarding a portion of the global state vector to the HP if the global state has changed since the last time it was forwarded to the HP. Having obtained the new global values, the AP checks for acknowledgments by executing the `DO_ACK` procedure. Finally, it removes one tagged communication from the FIFO (if there are any) and processes it. Note with this organization of the AP's algorithm, properties P5 and P6 of Section 4.2.2 are true.

```
AUX_PROC:  Read the PRN output;
           Write global state vector to HP interface if global state has changed;
           Perform DO_ACK;
           IF     FIFO is not empty
           THEN   Get next entry from FIFO;
                  CASE (entry_type):
                      NEW_CLOCK:  σ_i := new_clock_value;

                      SENT_MSG:   IF message_time < υ_i
                                  THEN υ_i := message_time;
                                  Add message to outstanding message list;

                      RCVD_MSG:   [IF ρ_i = {∞, Φ, 0}
                                  THEN ρ_i := (message_time, message_id)]
                                  ELSE Add message to unacknowledged message list;

DO_ACK:    IF     [ τ' = ρ_i AND ρ_i ≠ {∞, Φ, 0}]
           THEN   Remove next batch to be acknowledged
                      from the unacknowledged message list;
                  Set ρ_i to acknowledge this batch;

           [IF    ρ' has been sent to me
           THEN   IF  messages in ρ' batch are in outstanding message list
                  THEN τ_i := ρ' ];
                      Mark them as acknowledged;
                      Remove any other batches marked as acknowledged
                                  from outstanding message list;
                      IF    timestamp of acknowledged batch = υ_i
                      THEN  υ_i := smallest timestamp in outstanding message list;
                  ELSE τ_i := {∞, Φ, 0};              -- ignore primary ack
           ELSE   τ_i := {∞, Φ, 0};
```

Figure 7 (continued) - AFA2P: Auxiliary Processor Algorithm

6 Correctness of AFA2P

Recall that AFA2P is derived from AFA by distributing the work between the HP and the AP. Synchronization tasks have been off-loaded from the HP to the AP in AFA2P. This organization introduces latency in the data path for synchronization data (Figure 8), i.e., when an HP changes state so that GVT_a (t) changes, this change is not reflected immediately in the state vector presented to the GVT Calculator. Moreover, the HP and the AP are asynchronous. This means that after an HP enqueues a communication in its FIFO, it continues processing without waiting for the AP to process that communication. This is the main difference between the AFA2P and AFA computational models. For instance, in AFA, the time of the next event is always reflected in σ_i (and therefore GVT_p (t)) before the HP starts executing the event, while in AFA2P, σ_i may be set to the time of the event much after the event has been completely executed. In this section, we show that except for a minor modification to the`DO_ACK` procedure, the algorithm remains correct despite the latency. Note that properties P1 through P6 of Section 4.2 apply to the computation model of Figure 8. As with AFA, the correctness of AFA2P is determined by the two correctness criteria of Section 4. We show next that AFA2P satisfies these two criteria. To do so, we define the *minimum timestamp* at time t, $\mu_i(t)$ of AP_i as:

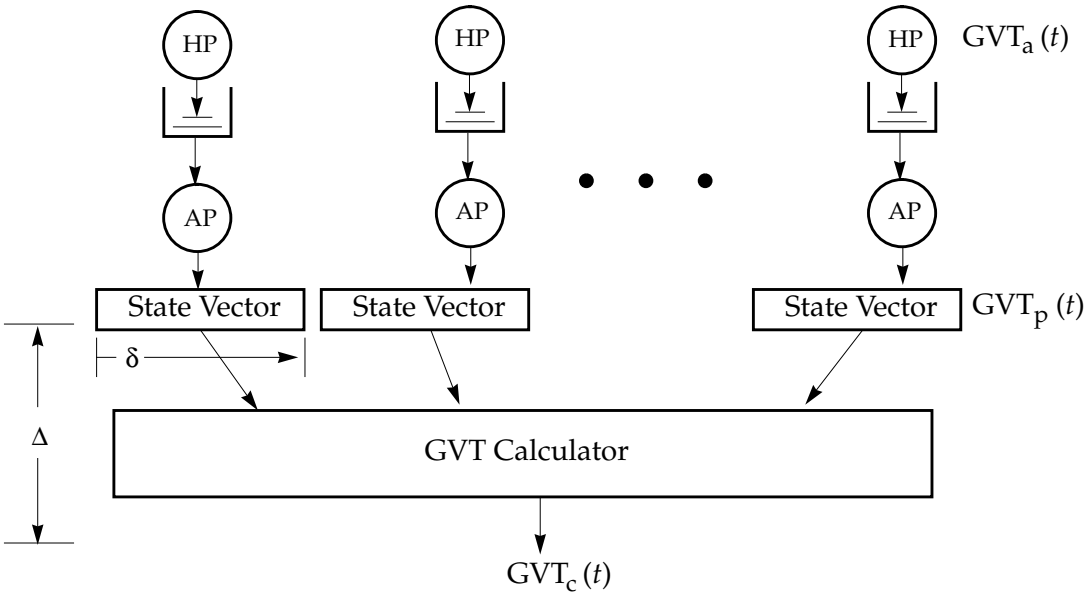


Figure 8 - GVT Computation Model for AFA2P

$$\mu_i(t) = MIN(\sigma_i(t), \upsilon_i(t))$$

THEOREM 3 $\text{GVT}_c(t) \le \text{GVT}_a(t)\ \forall t$

PROOF Comparing Figure 8 with Figure 6, we see that the relation between GVT_c (t) and GVT_p (t) is the same, i.e. Lemma 1 holds here as well: if a new reduction cycle is started at time s, and if $GVT_p(s) \le GVT_a(s)$ then $GVT_c(t) \le GVT_a(t)\ \ \forall t$. We are thus required only to prove that $GVT_p(s) \le GVT_a(s)$ for those times s at which a new reduction cycle is initiated. Once again, we do this by showing that AFA2P maintains the invariant of Lemma 2, which is restated here using our definition of μ_i above:

I: At any instant t, there is at least one LP_i such that $\mu_i(t) \le GVT_a(t)$.

For any t, GVT_a (t) is the timestamp of one or both of the following: (i) an event being executed (or just executed) at some HP (ii) a message in transit. We see in AFA2P that an HP always enqueues a `NEW_CLOCK` communication with the timestamp of the next event before executing that event. Similarly, an HP enqueues a `SENT_MSG` communication before sending a message. As a result, at any time t, we see that the FIFO of at least one LP (LP_i) will have a tagged communication with timestamp GVT_a (t). With regard to this communication, there are two possibilities: either the AP of the corresponding HP has processed the tagged communication or it has not. In the first case, we see from AFA2P (the `NEW_CLOCK` and `SENT_MSG` cases of the `CASE` statement in the AP's main procedure) that $\mu_i(t) \le GVT_a(t)$ and the invariant will hold. In the remainder of this proof, we analyze the second case wherein the FIFO of some LP contains a communication with timestamp GVT_a (t) and this communication has not yet been processed by the AP.

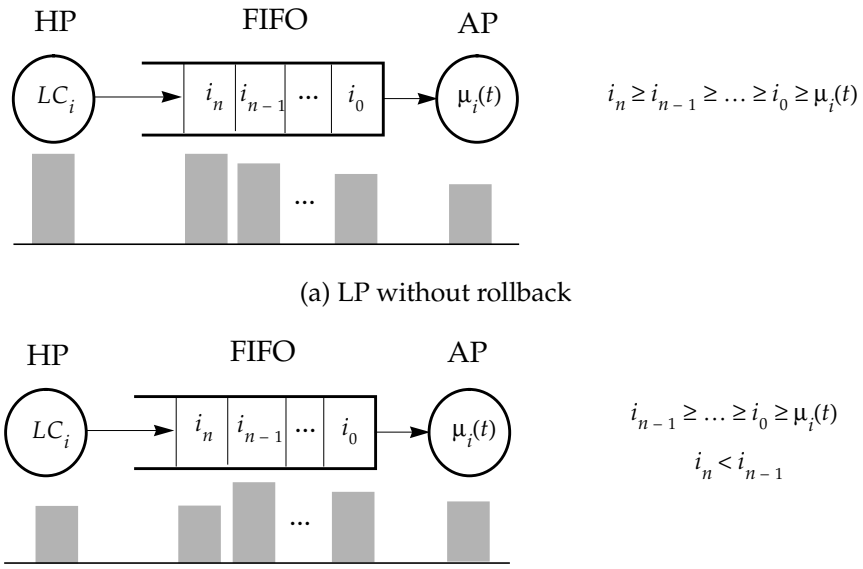


Figure 9 - Timestamp profi le of LP_i

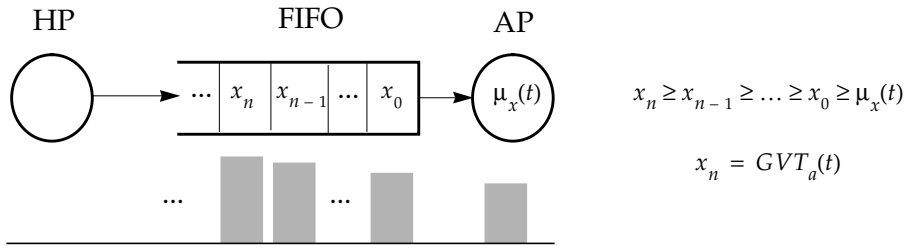
We use the pictorial notation of Figure 9 to show the timestamps present in an HP-AP pair and the FIFO between them. In this notation, vertical bars are used to indicate the relative values of the timestamps of the corresponding entities (HP, FIFO element or AP). The height of the bar is proportional to the value of the timestamp, i.e. the larger the timestamp, the taller the bar. The bar for the HP depicts the value of the HP's logical clock (LC). For each element in the FIFO, the bar depicts the timestamp of the communication enqueued (we consider only those elements which have a value enqueued in them and use i_j to symbolically denote the timestamp of the *j*'th communication from the head of the FIFO of LP_i). Finally, the bar for the AP depicts the AP's minimum timestamp, μ_i. Figure 9a shows an LP which has been processing forward in logical time in the recent past, while Figure 9b shows an LP which has recently suffered a rollback. We refer to a dip in the timestamps in a FIFO (such as i_{n-1} to i_n) as a *valley*. The effect of a rollback is to create a valley in the LP's FIFO. With regard to an element in the FIFO, we say that all of the elements closer to the AP are *ahead* of it. When a particular timestamp is irrelevant, it is simply not depicted.

At some time *t*, consider the LP (LP_x) with a communication with timestamp GVT_a (*t*) enqueued in its FIFO. The FIFO of LP_x can have one of the two profi les shown in Figure 10, with x_n = GVT_a (*t*) (strictly speaking, the FIFO can have more than two profi les, depending on the number of valleys in the profi le; however, there are only two interesting types: those with valleys ahead of the communication with timestamp GVT_a (*t*) and those without). If LP_x does not have any valleys ahead of this communication, then we see from Figure 10a that $\mu_x(t) \leq GVT_a(t)$ and the invariant will hold. The interesting case arises when the communication with timestamp GVT_a (*t*) is present in LP_x's FIFO due to a recent rollback (Figure 10b). In this case, it may happen that $\mu_x(t) > GVT_a(t)$. For the invariant to be true, there must be some other LP_r, such that $\mu_r(t) \leq GVT_a(t)$. We show now that this is the case.

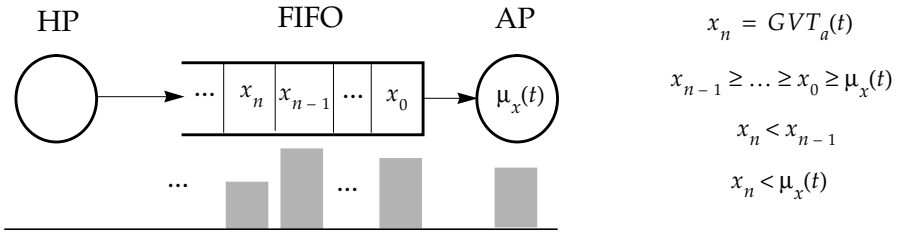
Consider the LP (LP_y) that sent the straggler message which caused the rollback in LP_x corresponding to the valley in Figure 10b (in the case where LP_x's profile has more than one valley, consider the valley closest to the communication with timestamp $GVT_a(t)$, but ahead of it in the FIFO). HP_y will have enqueued a `SENT_MSG` communication with timestamp $GVT_a(t)$ into its FIFO before sending the straggler. If this communication has been processed by AP_y, then $\upsilon_y(t) \le GVT_a(t)$ (since AP_x has not yet acknowledged the straggler message) and thus $\mu_y(t) \le GVT_a(t)$. If not, there are two possible situations: (i) LP_y is not the root of the rollback chain to LP_x or (ii) it is the root of the rollback chain to LP_x. We consider the two cases separately.

Case (i): Assume LP_y is not the root of the rollback chain to LP_x (i.e., LP_y has rolled back recently) and the `SENT_MSG` communication has not been processed by its AP so that its profile is similar to that shown in Figure 10b. Then LP_y's situation is similar to that of LP_x. Therefore, as we did for LP_x, we examine LP_y's predecessor in the rollback chain and continue up the rollback chain. This procedure terminates in one of two ways: (a) we reach a pair of LP's in the chain, LP_i and LP_j such that LP_i is before LP_j in the chain and AP_i has seen the `SENT_MSG` communication from HP_i or (b) we reach the root of the rollback chain. In case (a), as explained earlier, since AP_j has not acknowledged the straggler message, LP_i will have $\mu_i(t) \le GVT_a(t)$. Case (b) is the same as case (ii) above, which we consider next.

Case (ii): If the root of the rollback chain (LP_r) has a FIFO profile as shown in Figure 11*, then the invariant holds since $\mu_r(t) \le GVT_a(t)$. On the other hand, if its FIFO profile is as shown in Figure



(a) Profile without valleys



(b) Profile with a valley

Figure 10 - Possible profiles

* A dashed arrow is used to connect the tagged communication in a FIFO corresponding to a rollback-causing straggler message, with the valley due to that rollback in the succeeding LP. For instance, r_l is the `SENT_MSG` communication corresponding to the message sent by LP_r which started the cascading rollback. A vertical dotted line is used to indicate that intermediate LP's have been omitted for compactness.

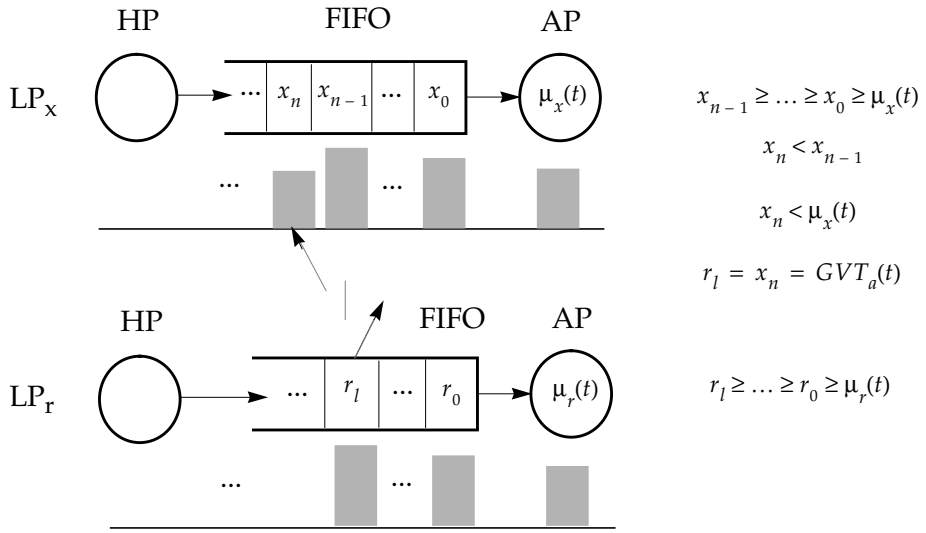


Figure 11 - Profile of the root of the rollback chain

12, then it has suffered a rollback recently, whose effects are still in its FIFO (r_p) and consequently, the invariant may not be satisfied by LP_r. However, LP_r is now in a situation very similar to that of LP_x and we apply our logic again, traversing backwards along the second rollback chain to LP_r. We show next that this process will terminate.

Consider the process again. We start with the LP that has the minimum timestamp in its FIFO. As we proceed up the rollback chain, we mark each LP that we visit. Note that each time a marked node is revisited, we move closer to the AP in the FIFO (because we are following a causal chain and 'A caused B' implies 'A occurred earlier than B'). Assume now that we reach the root of this chain and the root has a valley ahead in its FIFO, due to another rollback chain. We proceed up this new rollback chain, once again marking visited nodes. Here again, if we revisit previously

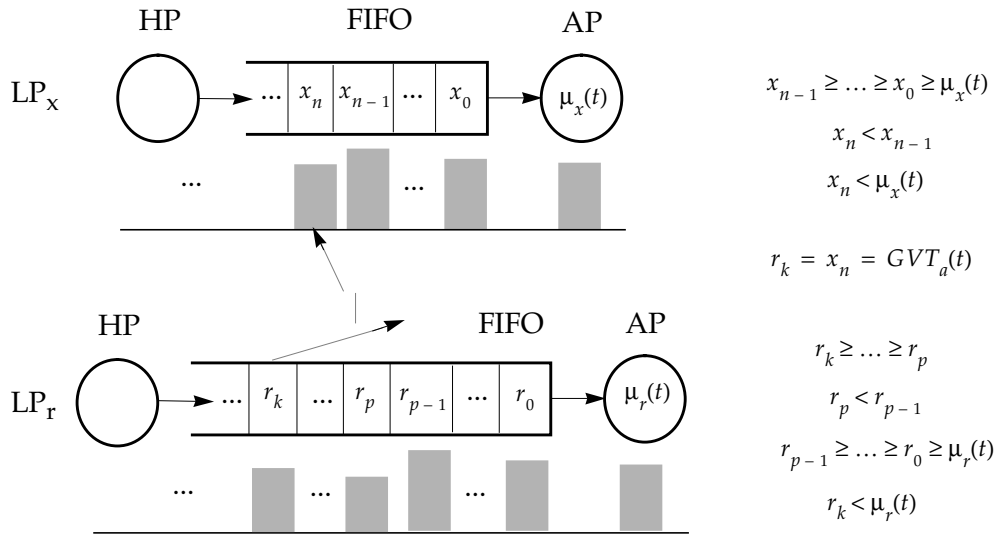


Figure 12 - Profile of the root of a rollback chain with a valley

marked nodes, we will visit them at a point closer to the AP, each time. Since the number of processors is finite (we cannot keep marking unmarked nodes indefinitely) and the number of cells in any FIFO ahead of the first element visited is finite, this procedure must ultimately terminate in one of the following ways:

i) We reach an LP_i whose AP has seen the `SENT_MSG` communication from its AP and therefore has $\mu_i(t) \leq GVT_a(t)$.

ii) We reach the root of a rollback chain (LP_i) whose FIFO profile has no valleys (Figure 11) ahead of the communication corresponding to the rollback-causing straggler, in which case LP_i has $\mu_i(t) \leq GVT_a(t)$.

Note that the entire argument for the truth of the invariant relies upon property P1 (no loss) of Section 4.2. Given that the invariant is true, Lemma 3 holds for AFA2P as well and the proof of Theorem 3 parallels that of Theorem 1. ■

As mentioned previously, the addition of the FIFO requires a modification to the `DO_ACK` procedure as can be seen by comparing the corresponding procedures of Figure 5 and Figure 7. In AFA, the timestamp and identifier of a message are always inserted into the outstanding message list prior to sending the message. In AFA2P, the HP sends the messages while the AP maintains the outstanding message list and the two processors are asynchronous. This leads to a race condition in the following situation.

It is possible (though extremely unlikely given the speed of the framework hardware) that HP_i sends out a message to LP_j and enqueues a `SENT_MSG` communication to AP_i such that the message is received by LP_j and the primary acknowledgment for the message is initiated by AP_j before AP_i processes the `SENT_MSG` communication. As a result, it may happen that AP_i observes AP_j's primary acknowledgment emerging from the PRN, but does not have the corresponding message identifier in its outstanding message list. With AFA, the `DO_ACK` procedure executing at AP_i assumes that the message was in the outstanding message list earlier but has been removed because its primary acknowledgment has been observed previously (i.e., it simply assumes that the acknowledgment for that message had been preempted). Therefore it submits the corresponding secondary acknowledgment. When AP_j observes this secondary acknowledgment, it removes its primary acknowledgment and the two-phase acknowledgment is completed. Finally, when AP_i processes the `SENT_MSG` communication, it inserts the message identifier into its outstanding message list where it remains indefinitely since AP_j will never acknowledge that message again. In such a case, υ_i (and hence $GVT_c(t)$) will never advance beyond the timestamp of that message.

To avoid this problem, in AFA2P a sending LP retains a message in its outstanding message list until both the phases of the acknowledgment of that message are complete (whereas in AFA, the message would be removed as soon as the first phase is completed). Thus, even if an acknowledgment is preempted, upon resumption, the sending LP will still find the message in its outstanding list. This allows us to distinguish between a preempted acknowledgment and the situation described above where the sending LP has not processed the `SENT_MSG` communication at all. Now, if a sending LP receives a primary acknowledgment for a message which it does not

find in its outstanding message list, it ignores the acknowledgment until such time when the corresponding message is indeed inserted into the outstanding message list. Apart from this, the two procedures are identical. Consequently, properties P13 and P15 of Section 4.2.4 apply to the `DO_ACK` procedure of AFA2P as well. Also, we see that P14 applies when the message being acknowledged is already in the sender's outstanding message list. Once a message has been sent by an HP, however, P1 (no loss) and P6 (finite delay) guarantee that this message will be inserted into the corresponding AP's outstanding message list within finite time. Thus, P14 applies to the `DO_ACK` procedure of AFA2P as well. Since the proof of Lemma 4 uses properties P5 and P13 through P15, Lemma 4 holds for AFA2P as well.

THEOREM 4 As t increases, GVT_c (t) increases if GVT_a (t) increases

PROOF Assume that $GVT_c(t) = GVT_c(t_0)$ for all $t > t_0$ while $GVT_a(t_1) > GVT_a(t_0)$ for some $t_1 > t_0$. This can happen only if some timestamp (either that of an event or of a message) remains indefinitely in the implementation even after it no longer exists in the ideal system.

We see in AFA2P that an HP always enqueues a communication in the FIFO with the timestamp of an event before executing it. By P6 (finite delay), the σ_i of the AP will be set to this timestamp in finite time. Therefore, the constant GVT_c (t) cannot be due to an unchanging σ_i.

The advance of GVT_a (t) implies that there are only a finite number of messages with a given timestamp. Lemma 4 and P6 (finite delay) guarantee that all of these messages are acknowledged in finite time. Note that P9 (update unreceived message time) is a statement of the way any LP_i updates its υ_i and is true for AFA2P also. This property ensures that an LP's υ_i will remain at any one value for only a finite amount of time. Thus, the assumption that GVT_c (t) remains constant while GVT_a (t) increases with t is contradicted. ■

7 Properties of GVT_c (t)

In this section, we describe three interesting properties of the GVT computed by AFA using the PRN. For brevity, we discuss these only for AFA, but each of these properties can be shown to be true for AFA2P as well.

7.1 GVT_c (t) is strictly non-decreasing

Criterion 1 for the correctness of a GVT computation algorithm requires only that $GVT_c(t) \le GVT_a(t)\ \forall t$. Here, we show a stronger property about GVT_c (t), namely that in addition to being less than or equal to GVT_a (t), GVT_c (t) is strictly non-decreasing (i.e., $t_1 > t_2 \rightarrow GVT_c(t_1) \ge GVT_c(t_2)$). The GVT computation model for AFA is shown in Figure 6. We see from the figure that GVT_c (t) = GVT_p (t-Δ). It is clear that if the values of GVT_p (t-Δ) at those times t-Δ when a reduction cycle is started are strictly non-decreasing, then GVT_c (t) will be strictly non-decreasing for all times t. We now show that this is the case.

Consider a change in GVT_p (t) from one reduction cycle (GVT_p (t_0) = G) to the next ($GVT_p(t_0 + \delta) = \hat{G}$). For exposition, we refer to T-values contributing to $\hat{G}$ using the ^ symbol and to those contributing to G without it (i.e., $\sigma_i(t_0 + \delta) = \hat{\sigma}_i$, $\upsilon_i(t_0 + \delta) = \hat{\upsilon}_i$, $\sigma_i(t_0) = \sigma_i$ and

$\upsilon_i(t_0) = \upsilon_i$). By definition, $\hat{G}$ is the minimum of the smallest $\hat{\sigma}_i$ and the smallest $\hat{\upsilon}_i$. In each case, we consider all possible ways in which these two *T*-values may be assigned.

<u>*Case (i)*</u>: $\hat{G} = \hat{\sigma}_i$ for some *i*

- LP_i finished processing an event and computed $\hat{\sigma}_i$ from the events list. $\hat{\sigma}_i$ must be at least as large as σ_i since the events list is sorted in non-decreasing order. We have $G \le \sigma_i \le \hat{\sigma}_i = \hat{G}$.

- LP_i received a straggler from LP_j, causing a rollback. In this case, $\hat{\sigma}_i < \sigma_i$, but since antimessages are also acknowledged, the straggler will be outstanding at LP_j. Thus $G \le \upsilon_j = \hat{\sigma}_i = \hat{G}$.

<u>*Case (ii)*</u>: $\hat{G} = \hat{\upsilon}_i$ for some *i*

- LP_i received an acknowledgment, and therefore set its $\hat{\upsilon}_i$ to the new smallest timestamp in its outstanding list. By P10 (unreceived is minimum) (Section 4.2.3), we have $\upsilon_i \le \hat{\upsilon}_i$ and since $G \le \upsilon_i$ we have $G \le \hat{G}$.

- LP_i sent a message and set $\hat{\upsilon}_i$ to the timestamp of the message. Here $\hat{\upsilon}_i < \upsilon_i$. However, since a message is sent only after executing an event (i.e., after setting σ_i), we have $\hat{\upsilon}_i = \sigma_i$ and $G \le \sigma_i$. Thus $G \le \hat{G}$.

7.2 GVT_c (*t*) tracks GVT_a (*t*)

A necessary condition for GVT_c (*t*) to be correct is that it is never greater than GVT_a (*t*). An algorithm that estimates a lower bound on GVT satisfies this condition. The GVT computed by AFA exhibits a much stronger property. From Figure 6 and the discussion on the operation of the reduction network in Section 2.1, it is clear that if GVT_a (*t*) (and hence GVT_p (*t*)) takes on at most one new value between successive reduction cycles, GVT_c (*t*) will take on exactly the same values as GVT_a (*t*), albeit lagging behind it. If GVT_a (*t*) takes on more than one new value between successive reduction cycles, then only the last one will be seen in GVT_c (*t*). In this case, GVT_c (*t*) takes on a subset of the values of GVT_a (*t*), once again lagging behind it. In either case, the values of GVT_c (*t*) are numerically the same as those of GVT_a (*t*) and not mere lower bounds.

7.3 GVT_c (*t*) approaches GVT_a (*t*)

The progress criterion (Section 4.3.2) guarantees only that if GVT_a (*t*) increases with *t*, GVT_c (*t*) also ultimately increases. Here, we make a stronger statement about GVT_c (*t*). Assume that after the simulation has run for some time, the LP's stop their processing and retain the values of their logical clocks. Assuming a reliable host communication network, all messages in transit will eventually be received and incorporated into the events lists of the receiving LP's. Now, since there are only a finite number of messages to be acknowledged, after some bounded time, the υ_i of each LP will become ∞. At this point, GVT_p (*t*) will equal the minimum of the σ_i of all the LP's and after time Δ, GVT_c (*t*) will equal GVT_a (*t*).

8 Performance of the aggressive framework

So far, we have described special purpose hardware to support parallel simulation and presented a synchronization algorithm which uses this hardware for computing GVT in an aggressive system. In this section, we provide some quantitative evidence of the benefi ts of the aggressive framework for parallel simulations and compare our work with previous efforts.

8.1 Speed

The primary design goal of the framework hardware is speed of operation. It is critical that synchronization values be computed and disseminated rapidly. Based on timing values obtained from the prototype of the framework hardware described in [24], the PRN is expected to compute a globally reduced state vector of four elements from 32 input state vectors in 1.2 microseconds. We note that in the design of the prototype, signifi cant effort was directed towards proof of concept in addition to speed of operation. Future designs of the hardware are expected to operate at higher speeds. Since another important design goal is generality (i.e., the capability to program the hardware for use in different kinds of applications), some speed must be traded for this.

Sometimes, a global computation may require multiple reduction cycles (for instance, the two-phase acknowledgment). In addition, the introduction of the interface between the HP and the AP introduces delays in a global computation. It is diffi cult to analytically estimate the time taken for such multi-cycle global computations. [28] describes two simulation experiments conducted to estimate the performance of the framework hardware and AFA2P. The fi rst of these was directed at determining the loading limits of the framework hardware. The two signifi cant results of this experiment are:

- The PRN is very fast - a two-phase acknowledgment takes few tens of microseconds to complete, with eight processors
- The framework hardware can support very fi ne grained simulations (event execution times under 100 microseconds)

The second experiment simulated eight LP's performing a busy-work application using AFA2P. The model was designed to compute the average amount of logical time for which GVT_c (t) lags behind GVT_a (t). Note that this takes into account the delay of the HP-AP interface as well. The observation that the average lag was around ten microseconds (an order of magnitude smaller than event execution times) clearly shows that the framework hardware (and AFA2P) can enhance any aggressive protocol by providing almost current values of GVT (and other such synchronization values) at very low costs. We note that the speed with which acknowledgments can be performed using the framework (as shown by the simulations) compensates for the fact that acknowledgments are serialized in the framework.

8.2 Scalability

The main benefit of using a tree structure for the PRN is that it scales well with the number of processors. As the number of processors, n, is increased, the time to compute a globally reduced state vector increases as $\log_2 n$. For instance, if the number of processors in the example of the previous section is doubled to 64, the time to compute a global state vector increases from 1.2 microseconds to 1.35 microseconds. With more processors, the time to perform an acknowledgment is expected to increase, since more messages will be submitted for acknowledgment per unit time. However, this problem can be alleviated by using several pairs of T-values to perform several acknowledgments concurrently.

8.3 Previous work

Since the introduction of the Time Warp protocol, several algorithms have been proposed to compute GVT in a message passing system [2, 3, 5, 8, 11, 20, 27]. With the exception of [8], all of these algorithms share the property that they use the host communication network of the message passing computer to compute GVT. Since inter-processor communication is relatively expensive in current computer systems, the main focus of research in this area has been to minimize the number of messages used to compute a value of GVT while keeping the computed value of GVT as close to the actual value of GVT as possible. Thus, computing GVT has been an expensive operation. This has had two side effects: (i) simulations tend to have comparably large computation granules (on the order of milliseconds, so that the cost of computing GVT becomes less significant) and (ii) GVT is not computed as often as it should be, resulting in wastage of memory. In this respect, our algorithm is most efficient, as it simply does not use the host communication network to compute GVT. Even though acknowledgments are used to maintain v_i at each LP, they are performed in the PRN. We noted in Section 8.1 that a message can be acknowledged using the PRN in few tens of microseconds as compared to hundreds of microseconds to milliseconds to do the same in a typical host network. GVT is computed by the framework asynchronously with the simulation computation and this is done very rapidly using the PRN. An LP may obtain the value of GVT by simply reading some memory locations and performing a MIN operation. As mentioned earlier, simulations indicate that under normal load conditions with eight processors, the value of GVT thus obtained will lag behind the actual value of GVT by few tens of microseconds. Even with event granules of tens to hundreds of microseconds, the GVT provided will be very accurate (possibly less than one event old). The work of Filloque et. al. is similar to ours in that the host communication network is not used to compute GVT. However, in their system, messages are acknowledged through the host network. In that sense, the host network is indeed used in the GVT computation. Also, the actual GVT computation algorithm proposed by them takes O(n) time where n is the number of processors, while ours takes O(log n) time. Thus, we believe our system will provide more accurate values of GVT and also scale better.

The first GVT computation algorithm was proposed by Samadi [27]. This algorithm uses message acknowledgments in the host network and is quite straightforward. It has the added disadvantage that one processor is designated the initiator of the GVT computation. It receives messages from all other processors as well as broadcasts the computed value of GVT to all.

Obviously, this processor becomes the bottleneck. A hierarchical scatter-gather scheme can be used to alleviate this situation. However, whenever a processor needs the value of GVT, all other processors are involved in the operation. Bellenot's algorithm [3] also uses acknowledgments to maintain the smallest unreceived message time. Here again, there is a designated initiator processor. The optimization in this algorithm is the way in which GVT is computed, so that the initiator does not become the bottleneck. Preiss [20] proposed a ring algorithm using a token which is passed around to compute GVT. Here again, all processors are involved in every GVT computation and the host communication network is used to compute GVT. A similar approach was followed in [13] where GVT is computed in a networked environment using *active messages* in the network. Active messages differ from passive messages in that the arrival of an active message at a network interface causes an arithmetic or logical operation to be performed on the value being carried by the message and a value stored at the interface. The result of this operation is carried further by the message. Thus GVT computation involves O (n) messages. Moreover, [13] do not address the issue of maintaining υ_i, which will require more messages through the host network. Lin and Lazowska [11] proposed an algorithm in which they eliminated acknowledgments of messages. This cuts the message traffic in the host network by half. However, Concepcion and Kelly [5] pointed out that the network traffic in the worst case for this algorithm could be O (n^2). In effect, this algorithm suffers the penalties of acknowledging messages. Concepcion and Kelly proposed the first asynchronous approach to GVT computation, called the Multiple Level Token Passing algorithm (MLTP). MLTP uses dedicated processors responsible only for GVT computation (and other Time Warp specific operations), one for each processor which does event processing. This approach is very similar to our HP-AP combination. However, this organization requires two messages for every message in a typical Time Warp system. MLTP also uses message acknowledgments to maintain local minimum timestamps. Acknowledgments as well as the GVT computation using a hierarchical token passing algorithm are done in the host network. In our system, the AP's use the high-speed PRN to do the above. An issue to be addressed when computing GVT asynchronously (since simulation computation continues while GVT is being computed) is how close the computed value of GVT is to the actual value. For MLTP, this lag is computed to be less than or equal to $14 \cdot (\tau_i + \delta_M)$ where the dominating factor is δ_M, which includes the time for at least one message communication through the host network. This is considerably larger than the few tens of microseconds projected for our algorithm. Bauer and Sporrer [2] also proposed an asynchronous approach to GVT calculation. Their method uses a dedicated processor for computing GVT (besides other functions). All other processors communicate periodically with this GVT Calculator. Obviously, the GVT Calculator is a potential bottleneck. Their algorithm also uses the host communication network to compute GVT. They claim that the computed value of GVT is "very close" to the actual value but do not provide quantitative evidence to support this. We believe the need to send messages to compute GVT will make this lag much larger than that in our system.

In summary, the main reasons why we believe our algorithm computes very accurate GVT are:

- high-speed, scalable PRN used to compute GVT
- GVT computed asynchronously

- messages acknowledged in the PRN
- dedicated processors manage GVT computation

9 Conclusion

All PDES protocols require some form of global information (for instance, breaking deadlock in non-aggressive systems and GVT in aggressive systems). Gathering this information through the host communication system is very costly as it usually requires several messages. To solve this problem, Reynolds [22] proposed the use of special purpose hardware for rapidly computing and disseminating critical synchronization information to all the LP's in a PDES. This hardware (the Parallel Reduction Network or PRN) basically provides a low-latency communication medium with limited communication facilities (i.e., the type of information that can be relayed is limited). To use the PRN, each LP must execute a synchronization algorithm. The hardware, synchronization algorithms and the synchronization values are together called the framework. The framework computes and distributes synchronization information to all the LP's and the PDES protocol uses these values. The problem of obtaining non-local information is especially acute in adaptive systems, where LP's dynamically control their aggressive processing, because each LP requires information only from its set of predecessors. Computing this kind of *target-specific* information [22] is generally more expensive than computing completely global information. Simulations [28] have demonstrated the speed of the framework hardware. It is expected that future versions of this hardware will have the capability to provide target-specific information at such high speeds.

While it was fairly clear that the Reynolds framework could work with a non-aggressive protocol, it was not so with protocols that display aggressive behavior. In this paper, we have presented an aggressive version of the framework synchronization algorithm and established its correctness. This algorithm will serve as a base to be used by aggressive and adaptive protocols. Since we chose Time Warp as our aggressive protocol, the framework was used to compute GVT. However, we note that this is just an example of the use of the framework in computing global values and demonstrates the possibility of integrating the framework with aggressive protocols. We have also presented a restructured version of the algorithm to suit the framework hardware organization and argued that this restructuring does not alter its correctness. We have described three properties of the GVT computed by our algorithm, which we believe will benefit adaptive protocols. Finally, we have presented results of preliminary studies of the performance of the algorithm and compared this with previous work in this area.

Acknowledgments

This work was supported in part by the National Science Foundation (grant CCR-9108448, Aug. 91, number 48) and MITRE Corporation (Academic Affiliates Program). All simulations were performed using SES/Workbench.

REFERENCES

1. Ball, D. and Hoyt, S., "The adaptive Time-Warp concurrency control algorithm", *Proceedings of the SCS Multiconference on Distributed Simulation*, January 1990, pp 174-177.

2. Bauer, H. and Sporrer, C., "Distributed logic simulation and an approach to asynchronous GVT-calculation", *Proceedings of the sixth workshop on parallel and distributed simulation*, January 1992, pp 205-208.

3. Bellenot, S., "Global virtual time algorithms", *Proceedings of the 1990 SCS Multiconference on Distributed Simulation*, January 1990, pp 122-127.

4. Chandy, K.M. and Misra, J., "Distributed Simulation: A case study in the design and verifi cation of distributed programs", *IEEE Transactions on Software Engineering*, Vol. SE-5, No. 5, September 1979, pp 440-452.

5. Concepcion, A.I. and Kelly, S. G., "Computing global virtual time using the Multiple-Level Token Passing algorithm", *Proceedings of the SCS Multiconference on Advances in Parallel and Distributed Simulation*, January 1991, pp 63-68.

6. Dickens, P.M., Reynolds, P.F., Jr. and Duva, J.M., "State saving and rollback costs for an aggressive global windowing algorithm", Technical Report number TR-92-18, Computer Science Department, University of Virginia, July 1992.

7. Dickens, P.M. and Reynolds, P.F., Jr., "SRADS with local rollback", *Proceedings of the 1990 SCS Multiconference on Distributed Simulation*, January 1990, pp 161-164.

8. Filloque, J.M., Gautrin, E. and Pottier, B., "Effi cient global computations on a processor network with programmable logic", Resear ch Report number 1374, Institut National de Recherche en Informatique et en Automatique, January 1991.

9. Fujimoto, R.M., "Parallel discrete event simulation", *Communications of the ACM*, Vol. 33, No. 10, October 1990, pp 30-53.

10. Jefferson, D.R., "Virtual time", *ACM Transactions on Programming Languages and Systems*, Vol. 7, No. 3, July 1985, pp 404-425.

11. Lin, Y-B. and Lazowska, E.D., "Determining the global virtual time in a distributed simulation", Technical Report number 90-01-02, Department of Computer Science and Engineering, University of Washington, December 1989.

12. Lin, Y-B. and Lazowska, E.D., "A study of Time Warp rollback mechanisms", *ACM Transactions on Modeling and Computer Simulations*, Vol. 1, No. 1, January 1991, pp 51-72.

13. Livny, M. and Manber, U., "Distributed computation via active messages", *IEEE Transactions on Computers*, Vol C-34, No. 12, December 1985, pp 1185-1190.

14. Lipton, R.J. and Mizell, D.W., "Time Warp vs. Chandy-Misra: A worst-case comparison", *Proceedings of the 1990 SCS Multiconference on Distributed Simulation*, January 1990, pp 137-143.

15. Livny, M., "A study of parallelism in distributed simulation", *Proceedings of the 1990 SCS Multiconference on Distributed Simulation*, January 1985, pp 94-98.

16. Lubachevsky, B., Weiss, A. and Shwartz, A., " Rollback sometimes works . . . if filtered", *Proceedings of the 1989 Winter Simulation Conference*, December 1989, pp 630-639.

17. Lubachevsky, B., Weiss, A. and Shwartz, A., " An analysis of rollback-based simulation", *ACM Transactions on Modeling and Computer Simulation*, Vol. 1, No. 2, April 1991, pp 154-193.

18. Pancerella, C.M., " Improving the efficiency of a framework for parallel simulations", *Proceedings of the 6th Workshop on Parallel and Distributed Simulation*, January 1992, pp 22-27.

19. Peacock, J.K., Wong, J.W. and Manning, E.G., " Distributed simulation using a network of processors", *Computer Networks*, Vol. 3, No. 1, February 1979, pp 44-56.

20. Preiss, B., " The Yaddes distributed discrete event simulation specification language and execution environments", Proceedings of the SCS Multiconference on Distributed Simulation, January 1989, pp 139-144.

21. Reynolds, P.F., Jr., " A spectrum of options for parallel simulation", *Proceedings of the 1988 Winter Simulation Conference*, December 1988, pp 325-332.

22. Reynolds, P.F., Jr., " An efficient framework for parallel simulations", To appear in the *International Journal in Computer Simulation*, Vol. 3, No. 4, 1992. An earlier version appeared in *Proceedings of the SCS Multiconference on Distributed Simulation*, January 1991, pp 167-174.

23. Reynolds, P.F., Jr. and Pancerella, C.M., " Hardware support for parallel discrete event simulations", Computer Science Technical Report No. TR-92-08, University of Virginia, March 1992.

24. Reynolds, P.F., Jr., Pancerella, C.M. and Srinivasan, S., " Making parallel simulations go fast", *Proceedings of the 1993 Winter Simulation Conference*, December 1992, pp 646-656.

25. Reynolds, P.F., Jr., Pancerella, C.M. and Srinivasan, S., " Design and performance analysis of hardware support for parallel simulations", Technical Report number TR-92-20, Computer Science Department, University of Virginia, June 1992. To appear in the special issue of the *Journal of Parallel and Distributed Computing* on Parallel and Distributed Simulation, 1993.

26. Reiher, P., Fujimoto, R., Bellenot, S. and Jefferson, D., " Cancellation strategies in optimistic execution systems", *Proceedings of the 1990 SCS Multiconference on Distributed Simulation*, January 1990, pp 112-121.

27. Samadi, B., " Distributed simulation: algorithms and performance analysis", Ph.D. Thesis, Computer Science Department, University of California, Los Angeles, January 1985.

28. Srinivasan, S., " Modeling a framework for parallel simulations", Master of Science Thesis, Computer Science Department, University of Virginia, May 1992.