Design and Performance Analysis of Hardware Support for Parallel Simulations

Paul F. Reynolds, Jr.,Carmen M. Pancerella,Sudhir Srinivasan

Computer Science Report No. CS-92-20
June 10, 1992

This research was supported in part by NSF Grant #CCR-9108448 and
JPL Contract #957721.

Design and Performance Analysis of Hardware Support for Parallel Simulations

Paul F. Reynolds, Jr.
Carmen M. Pancerella
Sudhir Srinivasan
Department of Computer Science
School of Engineering and Applied Science
University of Virginia

Abstract

It has been established elsewhere [Reyn92] that hardware to support parallel discrete event simulations (PDES) is desirable. We describe the steps leading to the implementation of a hardware-based framework to support PDES. We begin with an exploration of the criteria necessary to make such a framework both practical and useful, concluding that maintenance of sequential consistency is sufficient, while "observable" sequential consistency is more desirable but difficult to attain. We derive a functional design based on these criteria, and from that derive a prototype design. Also, we establish the utility of our design, showing that computation of critical global values, such as global virtual time, can be done in times two orders of magnitude or better than typical event times in discrete event simulations.

1. Introduction

The need for special purpose hardware to support efficient parallel discrete event simulation (PDES) is well established. For example, without support hardware, state saving and rollback costs in aggressive PDES protocols can overwhelm benefits gained from aggressive processing. Fujimoto has developed the state saving and rollback chip to reduce this substantial cost [FuTG92]. Similarly, Reynolds [Reyn92] has established the need for high-speed computation of critical values such as global virtual time (GVT), particularly in support of Fujimoto's rollback chip. In this paper we describe hardware to efficiently compute global reductions, as characterized by the computation of GVT. We describe our framework in the context of correctness criteria; a casual, intuitive approach invariably yields nasty surprises.

High speed hardware that can quickly disseminate globally reduced values, such as GVT, can be useful to PDES in many other respects as well. We have established elsewhere [RePa92] that such a network could support the rapid dissemination of values used in lookahead computations. Similarly, values reflecting processing rates could be disseminated rapidly for adaptive PDES protocols. Iterative PDES protocols, those which select a computation window in a loosely synchronous manner [Ayan89, ChSh89, Luba88, Nico91, SoBW88], can benefit from hardware support for the computation of the window size. Termination detection and global consensus [AbRi91] are other capabilities that can benefit PDES. All of these activities can be carried out in the framework we describe here.

We assume a familiarity with the common approach taken to performing parallel discrete event simulation, namely the partitioning of a discrete event simulation into components generally called logical processes (LP's). Each of the LP's is a discrete event simulation in and of itself except that it must ensure that it executes events, whether internally generated or received from other LP's, so as not to violate causality constraints. Ensuring the observance of causality constraints is typically left to a "PDES protocol." An excellent survey by Fujimoto [Fuji90] explores PDES issues and presents an overview of a wide range of protocols.

Our concern here is with respect to efficient support for PDES synchronization protocols. A "PDES framework," a mix of hardware and software designed to provide that support has been introduced in [Reyn91]. The purpose of this paper is to describe this framework further, to establish criteria for its proper operation, to describe functional characteristics, to describe a nearly complete prototype, and to present results of performance studies.

We use the terms "framework" and "hardware-based framework" to describe the entire software/hardware ensemble meant to support PDES. An integral part of the hardware portion is a parallel reduction network, the PRN.

We begin by establishing the properties required of a framework meant to support reductions on critical simulation values. This occurs in the form of correctness criteria for ensuring that desired results are achieved. We follow with a detailed description of hardware meant to support these criteria, and a detailed analysis of the contributions we expect our framework to make to parallel simulation. This description follows a nine month design effort that will lead to the completion of a prototype available Summer of 1992. We close with an evaluation of the expected performance of the framework we describe.

2. Correctness Criteria

Numerous examples have appeared in the parallel simulation literature demonstrating the need for globally reduced values. Lubachevsky [Luba89] requires global establishment of minimum next event time and other values to compute opaque periods. Global windowing algorithms, such as those proposed by Nicol [Nico91] and Dickens

[DiRe92] require establishment of parameters for the window. Reynolds has demonstrated the utility of values such as global minimum next event time, minimum unreceived message times and others in [Reyn92]. The hardware-based framework we describe in this paper is meant to provide these values correctly and expediently.

When computing globally reduced values, it would be best to allow the computation of these values to proceed asynchronously with simulation. Ideally, we desire a total ordering with respect to this asynchronous computation. That is, if a logical process (LP) submits a new value used in a global reduction and later another new value is submitted by any LP, including the one that submitted the first, the results of the global reduction involving the first submitted value should complete before the results of the global reduction involving the second.

Guaranteeing a total ordering requires consensus or its equivalent among the LP's as we discuss below. That can be quite expensive. We've found instead that sequential consistency is sufficient. In the following discussion we establish the correctness criteria, primarily requirements for computation ordering, for a hardware-based framework meant to support the computation of globally reduced values in support of parallel simulation.

We seek a method for performing global reductions on n-tuples of values submitted by n executing logical processes. Let the state of LP_i be represented by the m-tuple $<V_i^1, V_i^2,..., V_i^m>$. A reduction function, F, operates on n values, one from each LP_i; that is, the n-tuple $<V_1^k, V_2^k,..., V_n^k>$ for a given k *in* $1..m$ is the input to the k^{th} global reduction function, F_k. The reductions we require can be characterized by:

$$F_k := \theta_k <V_1^k, V_2^k, ..., V_n^k> \qquad \text{for } k = 1,...,m$$

where θ_k is an associative, binary operator applied to the k^{th} n-tuple. So, for example, if V_i^1 is LP_i's next event time, then $F_1 = min <V_1^1, V_2^1, ..., V_n^1>$ would be the minimum next event time for all n LP's.

In the course of executing a parallel simulation, LP's can change values in their state tuples asynchronously with respect to the states of other LP's. When LP_i changes V_i^k, the k^{th} global function, F_k, should be invoked in order to account for the new input value. Ideally, global functions should be invoked and completed in the same order in which LP's change the components of their states. Then we would have a total ordering. A total ordering is desirable because it carries sequencing information with it that could be exploited by the LP's. For example, the order in which globally reduced values are computed (along with the ID's of the LP's that caused the computations to occur) can reveal information about which LP's are sending messages to others. That information, in turn, could be used within an LP to determine, for example, its probability of receiving a simulation-related message over some interval of time.

While a total ordering is desirable, it is both expensive and unnecessary. Guaranteeing total ordering requires consensus among LP's with respect to the order in which global functions should be invoked. This kind of consensus can be achieved, but at a cost. The cost can be temporal: LP's must continually execute the equivalent of barrier synchronizations, or the cost can be monetary: specialized networks such as those proposed in [RaBJ88], [ReWW89] and [ReWW92] would be required in addition to the framework we describe in this paper. We have chosen to weaken the desire for total ordering because doing so leads to more efficient and cost-effective solutions and because we can accomplish all we need with a simpler and less expensive approach.

Sequential consistency, applied to the logical process approach to parallel simulation, requires that operations emitted by an LP be performed in the order in which they are emitted. Sequential consistency in this case does not address the ordering of operations *between* pairs of LP's. "Operations" in this case are the resulting global reductions that must be performed whenever an LP changes state values. Thus, if an LP changes its smallest unreceived message time (See [Reyn92]) and then changes its next event time, the corresponding global operations should be applied and completed in the same order. We note that sequential consistency does not require that the effects of the global operations be observed strictly in the order in which they were applied. However, the results of a pair of operations should never be observed in an order opposite that in which they were issued. Sequential consistency guarantees a no-later-than property with respect to observability. The importance of this becomes evident when we discuss data loss.

Even though sequential consistency alone does not require coordination among LP's, the fact that operations are *global* reduction operations reraises the issue of consensus among LP's. For example, consider the case where LP_i changes V_i^k and then changes V_i^l, while LP_j first changes V_j^l and then V_j^k. Independent of the temporal interleaving of these changes, if they all occur in an interval short enough so that all changes are completed before any global operations are initiated, then a consensus problem exists. Sequential consistency requires that F_k be invoked before F_l in support of LP_i's changes, but in the opposite order in support of LP_j's changes. We conclude that consensus, or its equivalent, among LP's is required to safely determine a total ordering of global operations.

An alternative to LP's determining the order in which global operations are invoked is to have global operation ordering be fixed and known. For example, if global reduction operations were applied cyclically, F_1, F_2, ..., F_m, then LP's could submit changes to state variables with temporal spacing between the changes that equaled or exceeded the time required to complete a cycle of m operations. This approach is sufficient to guarantee what we call "observable" sequential consistency as long as no data loss occurs. We discuss the effects of data loss next.

If it can be guaranteed that LP's will process all information produced by the cyclic application of m reduction operations then, as noted, we have achieved observable sequential consistency. However, we do not believe LP's will be able to process a constant flow of globally reduced values. Consider the speed at which such values could be produced. We show in later sections that global operations, with pipelining, can produce new results on the order of every 150 nanoseconds. The absolute timing is not the factor here; rather it is that time relative to a typical processor's instruction cycle time. Given current processor technology at most tens of instructions could be executed during the time a global reduction is performed. We conclude it is not reasonable to expect a processor to keep up with a flow of global reductions.

We could consider slowing the reduction rate so that processors could keep up with the flow of output. However, low latency is critical. When an LP computes a new state value the corresponding global operation should be completed as quickly as possible. The importance of this is established in a subsequent section on performance analysis.

As demonstrated above, cyclic application of the reduction functions is a satisfactory replacement for a dataflow approach; if globally reduced values can be processed without loss then observable sequential consistency can be guaranteed. However, even if loss does occur, we can ensure sequential consistency. Consider treating each m-tuple $<V_i^1, V_i^2, ..., V_i^m>$ for each LP_i as a state vector. Similarly, consider treating the application of $F_1, F_2, ..., F_m$ as a globally reduced state vector: $<F_1, F_2, ..., F_m>$. If, rather than allowing results of the application of individual F_k to be lost, we required the granularity of a loss to be global state vectors, then it is possible to ensure sequential consistency. We elaborate.

At the beginning of a cycle a snapshot is taken of the m-tuples for each of the LP_i. Then $F_1, F_2, ..., F_m$ are applied to these m-tuples. In the meantime the LP_i can change components of local copies of their m-tuples. These changes will not be observed until another snapshot is taken at the beginning of the next cycle. At the end of a cycle the globally reduced state vector $<F_1, F_2, ..., F_m>$ is made available to all of the processors. Some of the processors may succeed in processing the information and others may not, resulting in a loss.

If, preceding one cycle, LP_i changes V_i^k and just preceding the next cycle it changes V_i^l, then the pair of resulting globally reduced state vectors have a desirable property: the first incorporates the change to V_i^k only and the second incorporates the changes to both V_i^k and V_i^l. Observable sequential consistency is maintained for any processors that process both vectors. For those that lose the first but observe the second, sequential consistency is maintained; that is, the ordering of changes to V_i^k and V_i^l is not necessarily preserved, but never violated. Informally, the "precedes" relation maintained by observable sequential consistency is relaxed to the "no-later-than" sequential consistency.

Many of the ordering requirements that arise in parallel simulation can be satisfied with sequential consistency. For example, computation of GVT requires maintenance of smallest next event time and smallest unreceived message time by each LP. When an LP completes an event and sends a message to another LP, both the next event time and the smallest unreceived message time may change. A simulation-wide invariant that must be maintained, as demonstrated in [Reyn92], is that the smallest event time in the system must always be represented in at least one LP's next event time or smallest unreceived message time. An LP just completing an event must not allow its new next event time (which may be infinity) be a part of a global reduction of all minimum event times before its new smallest outstanding message time. However, it is sufficient to allow them to be used in global computations simultaneously. That is, sequential consistency is sufficient to support maintenance of the invariant.

We note that sequential consistency does not require that the effects of a particular v_i^k ever be observed. It is possible for LP_i to emit a sequence of new values for v_i^k at a rate where the effects (possible changes in f_k) of only some members of that sequence are observed. Proofs of correctness must incorporate consideration for LP's that operate this way.

Given the m-tuple $<V_i^1, V_i^2,..., V_i^m>$ for each of n LP_i's, and the desire to produce $<F_1, F_2, ..., F_m>$ in a sequentially consistent manner, we have determined that a framework to accomplish this must do the equivalent of cyclically (1) taking a snapshot of the $<V_i^1, V_i^2,..., V_i^m>$, (2) performing each of the m global reductions, F_k on the captured m-tuples, and (3) presenting the resulting global reductions, $<F_1, F_2, ..., F_m>$, to each of the LP_i atomically. While the global reductions are being performed, each LP_i should be able to update local copies of state variables (members of the m-tuple $<V_i^1, V_i^2,..., V_i^m>$) in preparation for the next snapshot.

In the sections that follow we describe a hardware-based framework that meets the requirements just given. As will be seen, a simple extension enables the framework to produce observable sequentially consistent results as long as no loss of globally reduced m-tuples $<F_1, F_2, ..., F_m>$ can be guaranteed.

3. A Functional View of the Framework Hardware

We now focus on hardware to support parallel simulation [RePa92]. In this section we present a functional view of a hardware design which satisfies the correctness criteria established earlier. Specific details of this design are discussed in Section 4. All design decisions have been made to ensure the correctness criteria. In Section 5 we present a completed prototype design which is a collaborative effort of the Computer Science and Electrical Engineering Departments at the University of Virginia; the scheduled completion date of this hardware is end of Summer 1992.

A high-level view of the system including the *framework hardware* appears in Figure 1. This high-speed hardware consists of a *parallel reduction network* (PRN) augmented with dedicated processors to manage the high frequency I/O from the PRN and execute low-level synchronization algorithms. Throughout this paper, we use the term "framework hardware" to refer to the reduction network, the auxiliary processor unit, the interface between an host processor and its corresponding auxiliary processor, and the interface between an auxiliary processor and the PRN. The host system for the framework hardware is a closely coupled network of high-speed processors with its own network for interprocess communication. This host network is independent of the framework hardware. All interfaces between the host processor and the auxiliary processor (See Section 4.3) and between the auxiliary processor and the reduction network (See Section 4.5) have been designed to ensure the correctness criteria.

Each *host processor* (HP) is paired with a corresponding *auxiliary processor* (AP) which interfaces directly to the reduction network. The general purpose auxiliary processors, one processor per host processor, provide the interface between host processors and the reduction network. A high-speed bidirectional communication channel exists between a host processor and its corresponding auxiliary processor.

The parallel reduction network rapidly computes and disseminates different binary, associative operations across state vectors of values. Each element of a state vector is an input to a binary, associative operation. For example, it can be specified that all first components are to be summed, all second components OR'ed, and the minimum is to be taken on all third components in a three component state vector. As per the correctness criteria, the state vector is the basic unit of operation in an implementation of the framework hardware. Interfaces into and out of the network preserve state vectors: the hardware reads a state vector of size *m*, computes *m* globally reduced values, and writes a globally reduced state vector. The hardware guarantees that a partial or incomplete globally reduced state vector is never seen by an application running on an AP.

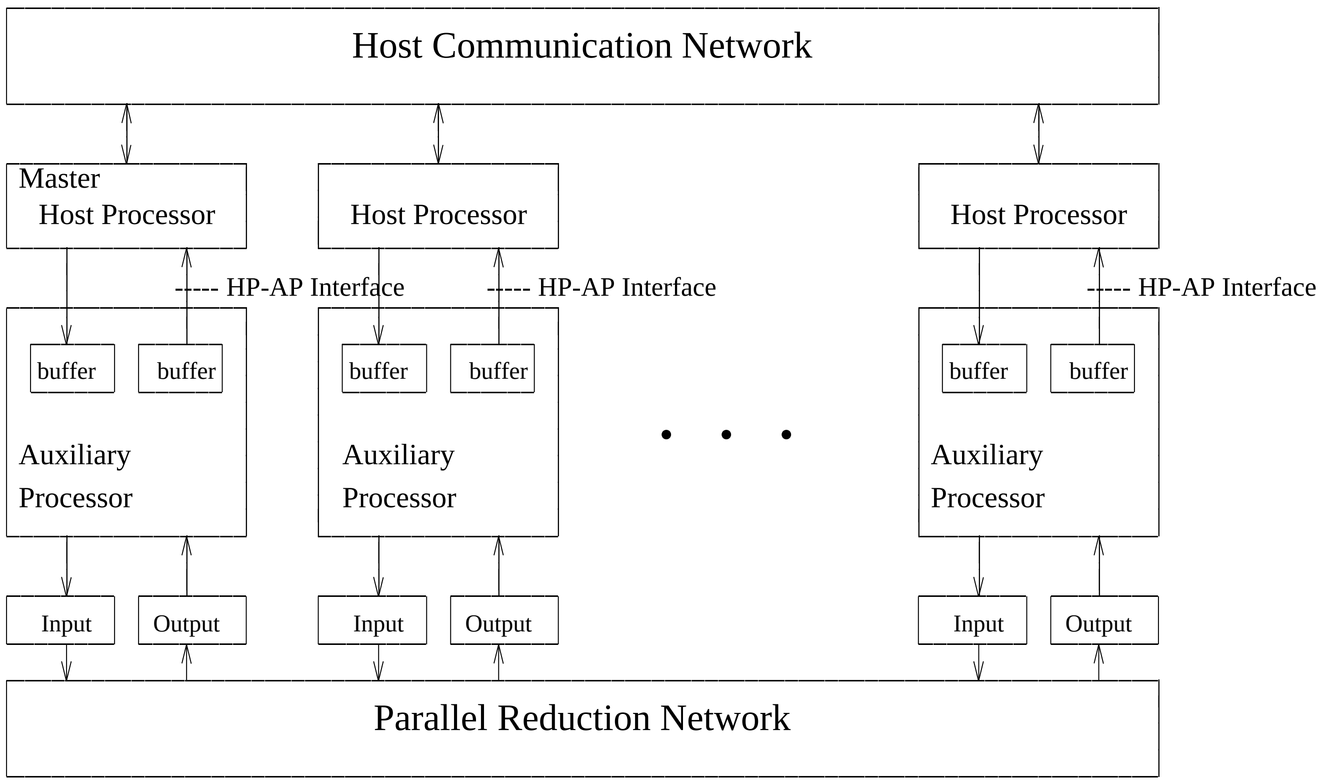

Figure 1. High Level View of Framework Hardware.

The addition of auxiliary processors at the interface to the reduction network facilitates the management of high frequency data coming out of the network. This is motivated by the prototype design expectations, which estimate a new state vector can be emitted from the reduction network every 150 * *m* nanoseconds, where *m* is the number of elements in a state vector. The AP's are responsible for inserting state vectors into the PRN as well but with much lower frequency since this is performed under program control. Hence, our hardware-based framework supports the computation and dissemination of globally reduced values, across all processors, without coordination of host processors, i.e., without barrier synchronization.

Employing auxiliary processors provides a separation of the synchronization activity (performed on auxiliary processors) and the application being simulated (performed on host processors). High speed synchronization activity in the parallel simulation framework (See [Panc92] and [Srin92]) is performed on the dedicated AP's. The logical processes (LP's) execute on the host processors, and all event messages are sent and received at the host processors. When an AP reads a new globally reduced state vector from the network, it writes selected groups of these values into the HP-AP interface readable by the host processor. An LP executing on a host processor can compute GVT, avoid deadlocks, and make processing decisions based on the global synchronization values. Other than simple tests such as these, the execution of the framework algorithms does not interfere with an LP’s event processing. A further advantage of a dedicated processor interfacing with the host processor and the reduction network is that an AP can compute the input reduction values based on multiple LP’s executing on one host processor and coordinate the synchronization activity of multiple LP’s. In sum, our framework offloads all parallel simulation synchronization overhead from host processors and the host network.

4. Details of the Hardware Design

In the sections that follow we discuss the specifics of the hardware design. In this discussion we focus on how we ensure the correctness criteria established earlier. We note that this is not the only hardware design which could guarantee the correctness criteria.

4.1. Auxiliary Processor

The general layout of an auxiliary processor is depicted in Figure 2. Auxiliary processors are fast, general purpose 32-bit microprocessors. Each AP has its own memory to store synchronization programs and related data structures (See [Reyn92], [Panc92], and [Srin92]). Furthermore, each AP has EPROM to store a boot-up monitor which is executed upon reset.

4.2. Setup

Each auxiliary processor boots up in a "listening" state in which it monitors its host processor interface. A host processor sends tagged data to its auxiliary processor representing a program to be loaded and executed by the AP. The physical interface between a host processor and its auxiliary processor is described in the next section.

One of the host processors in the system and its corresponding auxiliary processor is designated as a *master pair* of processors. The master pair communicates PRN programming information to the state machine controlling the PRN. Critical information to be passed to the state machine includes the number of components in a state vector and the operations to be performed on components. For example, it can be specified that all first components are to be summed, all second components OR’ed, and the minimum is to be taken on all third components in a three

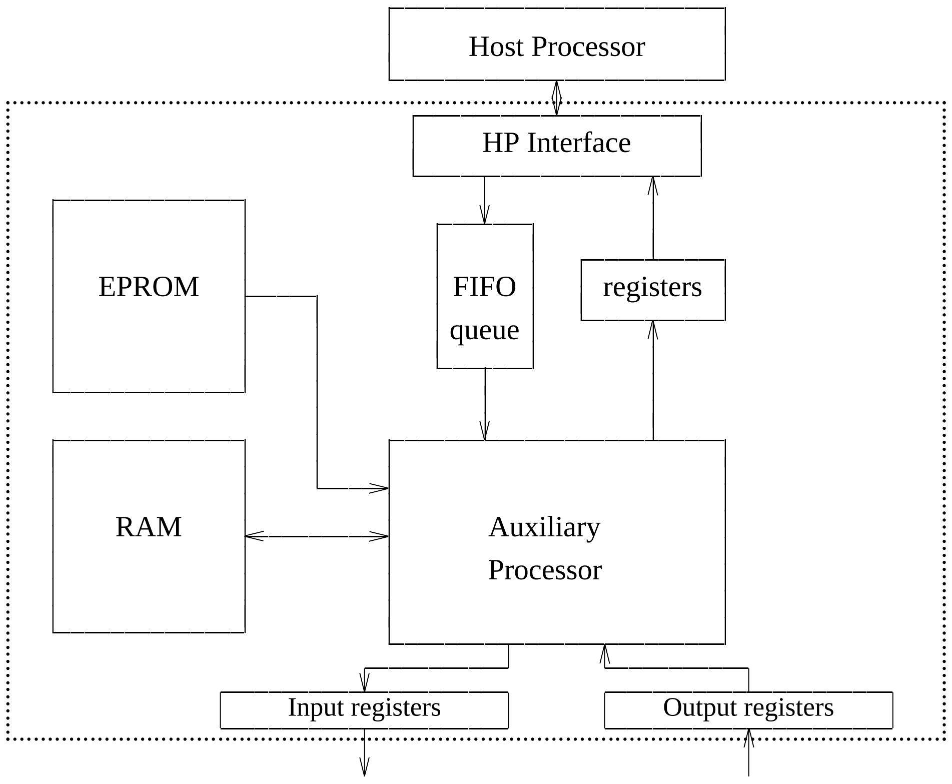

Figure 2. Auxiliary Processor.

component state vector.

The master host processor can send tagged data representing new PRN programming information to its auxiliary processor at any time. Similarly, host processors can send data to their respective auxiliary processors indicating they are to receive new programs to execute. This will permit dynamic reprogramming of the AP's and the PRN. We assume that applications running on the HP's and programs running on the AP's are sufficiently robust to support this dynamic reprogramming.

4.3. Host Processor - Auxiliary Processor Interface

Functionally, there are two data paths between a host processor and auxiliary processor: one from the HP to the AP and the other from the AP to the HP. Each processor is a reader on one data path and a writer on the other path. The host occasionally writes tagged information to the interface which the AP processes, based on the tag, and generates values to input into the PRN. Similarly, the AP writes globally reduced values to the interface which is read by the HP.

The addition of dedicated processors requires correctness criteria to be preserved between a host processor and its auxiliary processor. There are two requirements on the data path from an HP to its AP: (1) no information sent by the HP is lost and (2) the AP processes the data in the order in which it is sent by the HP. Under the established correctness criteria, an application executing on the HP does not need to see all globally reduced values; a recent version of globally reduced values, however, is expected to be available to the HP. This suggests an implementation requires at least a FIFO queue from HP to AP and an atomic set of registers from AP to HP.

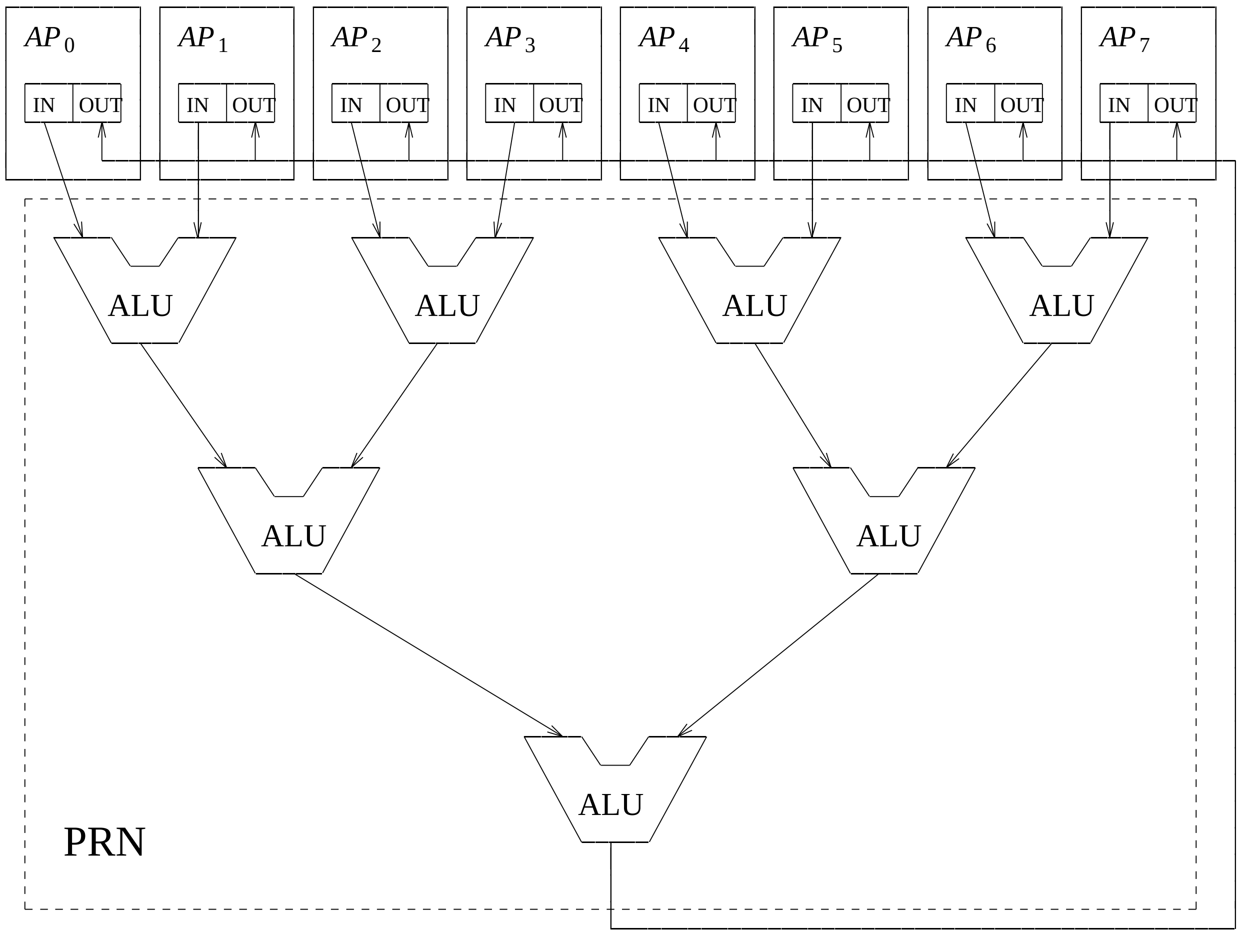

Figure 3. The Parallel Reduction Network.

As seen in Figure 2, the host processor can access the FIFO queue and the registers via an HP interface. The HP interface isolates the particular host processor from the rest of the system. If the host system changes, this HP interface is the only thing that will need to be redesigned. Isolating the HP interface provides adaptability to other parallel processors or closely coupled networks. For example, the HP interface could be changed from a SCSI to a VME interface, and all that would be required is the logic to respond to requests by the HP on the FIFO queue and register bank.

4.4. The Parallel Reduction Network

As seen in Figure 3, the PRN is a binary tree of depth $\log_2 n$, where n is the number of host (and auxiliary) processors. Each stage of the PRN consists of half as many ALU's as the stage above it, with the first stage having $n/2$ ALU's. The PRN's binary tree properties allow a global reduction operation to be computed and disseminated in $O(\log n)$ time. We note that the PRN does not have to be a binary tree; it could, for example, be constructed as a quad tree.

A single ALU node is shown in Figure 4. The ALU's perform reduction operations, i.e., binary, associative operations on two inputs based on a programmed operation code which accompanies the inputs; operations include sum, minimum, maximum, logical AND, logical OR, etc. The ALU's support *tagged selective operations*; in a selective reduction operation, such as minimum or maximum, a tag accompanies the "winning" value of the binary operation. Each input register is paired with a tag register. The PRN propagates the tag of the input that "wins" a selective operation, a minimum or maximum operation, so that the tag of the smallest or largest element emerges from the bottom of the PRN for a minimum or maximum operation. In the case where there is no single choice in a selective operation (i.e., both operands are equal), the PRN selects deterministically the tag which is propagated. A selective operation requires two operations in the ALU: a compare and a select.

As shown pictorially in Figure 4, two inputs and two reduction operation codes arrive at an ALU node. An error check is performed on the reduction operations; if the two operations are not equal, an error flag is placed in the tag, and the tag is propagated through the network. After a reduction is performed, the resultant value and the operation code are propagated through the PRN.

Pipelining is employed in order to use this network efficiently: partial results are pipelined through the $\log n$ stages of the PRN such that each stage of ALU's is always busy. The PRN can pipeline reduction operations at a

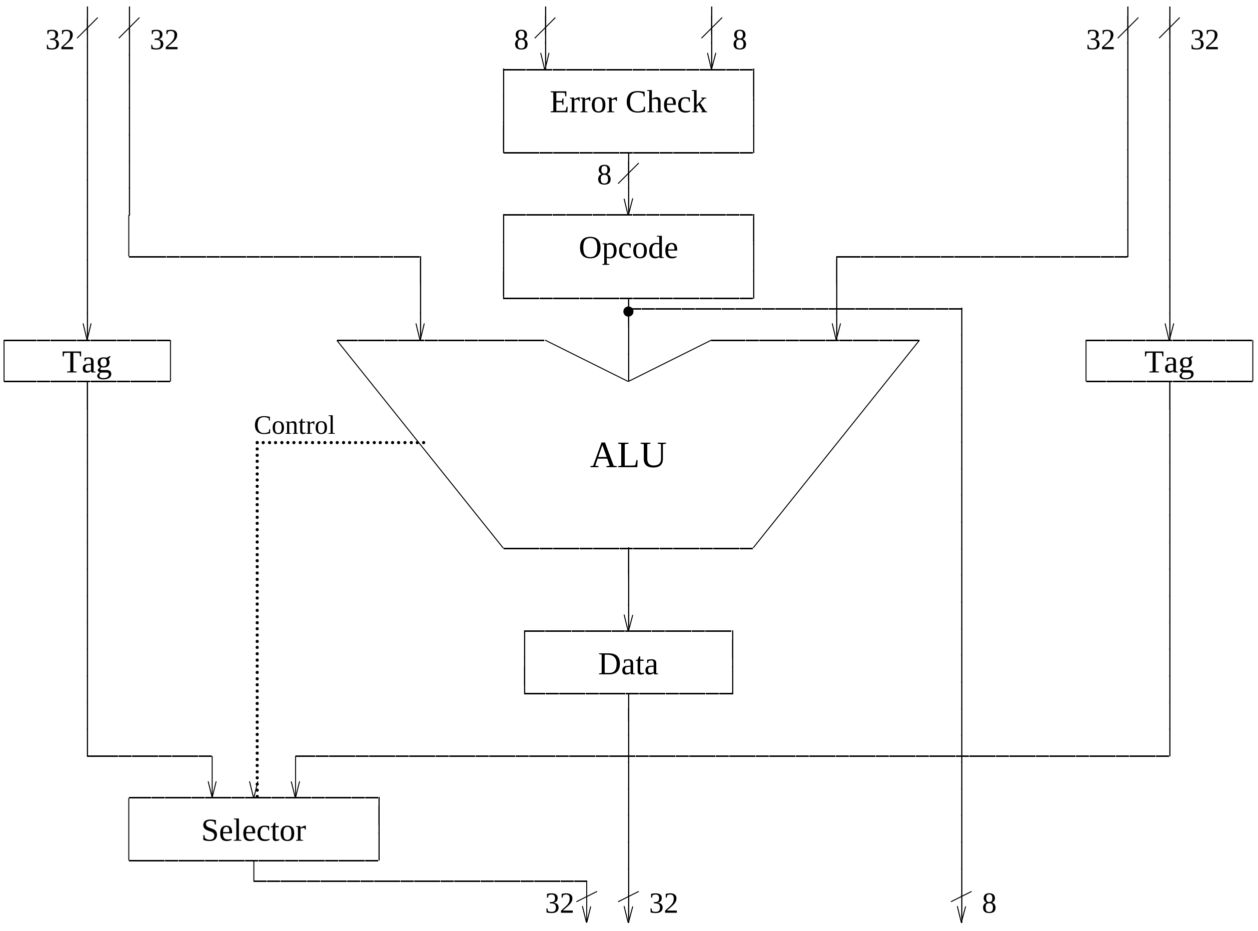

Figure 4. An ALU Node in the Parallel Reduction Network.

rate equal to the delay time of a stage. The time for a value to pass from one level of the PRN to the next is a *minor cycle*. Thus, the time to produce a sequence of values for state vectors of length m is $c * m$ nanoseconds, where c is the minor cycle time (plus the time to fill the pipe which is $c * \log_2 n$ nanoseconds).

As seen in Figure 3, the interface to the PRN from each AP is identical. Each AP has sets of memory-mapped input registers and memory-mapped output registers. A processor can write to the input registers and read from the output registers; the PRN will read values from the input registers and write the corresponding globally reduced results into the output registers. This memory-mapped interface is a possible source of memory contention if both the PRN and the auxiliary processor attempt to access the input or output registers simultaneously. We discuss next how the interface between the processor and the PRN is constructed in order to minimize memory contention, to facilitate atomic writes with and without overwrite capabilities, and to preserve state vectors.

4.5. Auxiliary Processor-PRN Interface

The AP-PRN interface is designed to operate on state vectors in order to support both atomic accesses of globally reduced values and order preservation of input values to the reduction network. From an LP’s point of view, it feeds a *valid* state vector to the PRN, where "valid" is defined by the application using the framework hardware. The PRN reads the state vectors, processes them by performing the corresponding reduction on each element, and writes a globally reduced state vector at each AP. Furthermore, the hardware provides an atomic read access to a single output state vector so than an AP can read an entire state vector. The application software, however, must access whole state vectors, not individual elements.

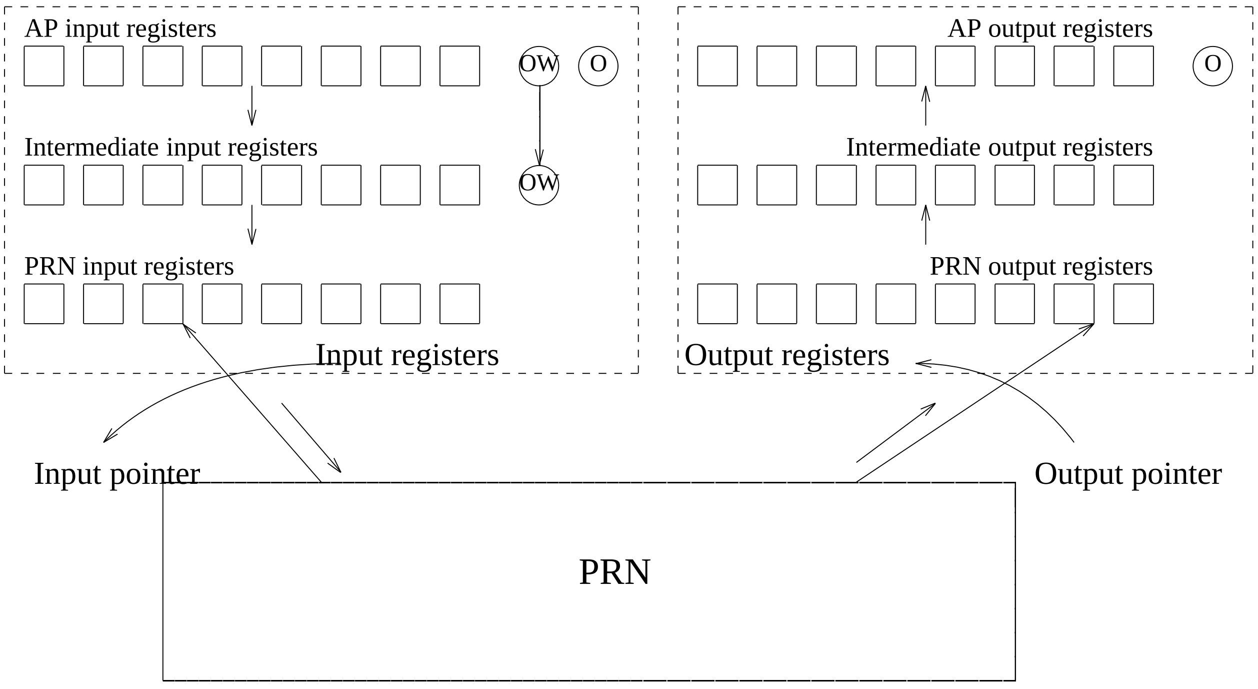

Figure 5. Interface Between an Auxiliary Processor and the PRN.

An auxiliary processor and the reduction network operate asynchronously with respect to one another. As shown in Figure 5, three banks of input and output register pairs provide an interface of isolation, such that both can access the register banks with minimal interference. This interface is designed to guarantee that *the PRN never blocks* while waiting to read a value or write a value. The PRN is expected to read and process state vectors at a rate much faster than an AP produces them; the PRN, therefore, may read and process the same state vector repeatedly. Even if an input value changes with high frequency, it will very likely be used more than one time in the computation of a reduction due to the relative speeds of AP's and the PRN. Similarly, on the output side, the PRN will produce globally reduced state vectors faster than an AP can read and process them, and as a result the AP's may lose some state vectors. Applications executing on the framework hardware will have to tolerate the loss of globally reduced state vectors. All reads to registers from the PRN or an AP are nondestructive. We now discuss the input and output interfaces in greater detail.

4.5.1. Auxiliary Processor-PRN Interface: Input

The interface from an auxiliary processor to the PRN consists of three banks of eight register pairs: the *AP input registers*, the *Intermediate input registers*, and the *PRN input registers*. The AP writes state vectors of size *m* to the top row of registers, the AP input registers, and the PRN reads state vectors of size *m* from the bottom row, the PRN input registers. The state machine which controls the interface transfers state vectors from the AP input registers to the Intermediate input registers and then to the PRN input registers. The transfer is done so as to minimize interference. Intermediate registers facilitate getting snapshots of valid local state vectors to be passed on to the PRN input registers without blocking the PRN.

When an auxiliary processor has completed writing a new state vector, it sets two single-bit control flags: the *overwrite bit* (OW) and the *owner bit* (O). The owner bit is *always* set when the AP has finished writing a valid state vector into the AP input registers; this indicates that the interface controller now owns the top level of registers. When the interface state machine transfers this state vector to the Intermediate input registers, it resets the owner bit indicating that the AP once again owns the AP input registers. If the AP attempts to write to the AP input registers while the owner bit is still set, it will be blocked. However, given the relative speeds of the PRN and the AP, this is not expected to happen often.

The overwrite bit gives the application some control over what values are eventually fed into the reduction network. Specifically, if the AP marks a state vector as "non-overwritable", it is guaranteed that the entire vector will be processed by the PRN. When the control logic transfers the AP input registers to the intermediate level, the overwrite bit is also transferred. If the AP indicates a state vector is overwritable then the state machine controlling the register banks can allow subsequent state vectors written by the AP to overwrite the state vector in the Intermediate input registers. If the AP signals a state vector as non-overwritable and it is transferred to the Intermediate registers, the overwrite bit will prevent the transfer of a newly written AP level state vector until the Intermediate input registers are ultimately transferred to the PRN input registers. The control logic guarantees that AP input registers are only moved to the Intermediate input registers when this process does not cause the PRN to block or when it does not lead to a loss of integrity of a state vector. Finally, we note that due to the relative speeds of an AP and the PRN, it is very unlikely that an overwritable state vector will be overwritten prior to being read by the PRN; however, we have designed the network to provide the guarantee anyway, for future use.

The combination of non-overwrite on input to the PRN and no loss of state vectors on output is sufficient to guarantee the observable sequential consistency introduced in Section 2. If either of these conditions cannot be met then observable sequential consistency cannot be guaranteed. However, neither is required to guarantee sequential consistency. We discuss this further at the end of this section.

The PRN reads state vectors of a specified size cyclically, starting with the mth element and proceeding to the first element. Thus, the PRN reduces the mth element, followed by the $(m-1)$st, and so on. The PRN is pipelined, thus the processing of the $(i-1)$st elements commences as soon as the top level of ALU's completes processing the ith elements. The PRN reads the ith register pair from each of the n input banks simultaneously. The time for the PRN to read an entire state vector is an *input cycle*. An input cycle finishes when the first elements of the state vector are consumed. At the end of an input cycle, the controller transfers the Intermediate input registers to the PRN input registers. The transfer can be overlapped with the last PRN read in the input cycle; thus, the hardware requires a minimum state vector size of two so that this transfer can be performed as efficiently as possible. The transfer from the Intermediate registers to the PRN registers has a higher priority than the transfer from the AP registers to the Intermediate registers so that the PRN never blocks.

We note that $\log_2$ of n and m are not necessarily equal. Therefore, while the PRN is reading from the ith input register pair from all n processors, it is not necessarily writing the ith output register pairs. That is, the PRN may complete reading state vectors from each of n input register banks at a different time than when it completes writing new reduced state vectors. The writing of a reduced state vector for a set of input state vectors will lag by $(((m-1) + \log_2 n) * c)$ nanoseconds, where the minor cycle time is c nanoseconds and there are n processors.

4.5.2. Auxiliary Processor-PRN Interface: Output

As shown in Figure 5, the three banks of output registers are constructed to preserve state vectors and to minimize AP-PRN interference in a similar fashion to the input register banks. Once every m minor cycles (assuming a full pipe in the PRN), the PRN generates a globally reduced state vector, which is written to the *PRN output registers*. This state vector is transferred to the *Intermediate output registers* and finally to the *AP output registers*, which are readable by the AP. Once again the interface controller guarantees that the PRN never blocks, and transfers between output register levels are prioritized to prevent this.

Each time the PRN completes writing a state vector into the bottom row of registers, the values are shifted into the Intermediate output registers. When the bottom row is shifted, the values in the intermediate row are concurrently shifted into the AP output registers unless the AP has locked the top row because it is reading the AP output registers. In that event, the Intermediate output registers are overwritten by the PRN output registers, and the contents in the intermediate registers are lost forever. The AP output registers have a control bit, an owner bit (O), that is set and reset by the auxiliary processor. The owner bit determines whether Intermediate output registers can be written to the AP output registers or are lost; it also ensures an atomic read of a state vector by the AP. The AP may block momentarily if it attempts to set the owner bit to itself while the intermediate values are written in parallel to the registers readable by the AP. Applications using the framework hardware must be robust enough to tolerate the loss of state vectors emerging from the PRN. Consistent with the correctness criteria set forth in Section 2, we note that an AP never sees a partial state vector. State vectors are either seen in their entirety or not at all.

The three levels of registers on the input side guarantee sequential consistency by preserving state vectors, as discussed in Section 2. Observable sequential consistency requires that the overwrite bit be used whenever the values in a state vector must be used in a global operation. Furthermore, there can be no loss of state vectors on the output side of the PRN — that is, AP’s must process every state vector that emerges from the network — if observable sequential consistency is to be maintained. An alternative, which is the equivalent of sequential consistency, is to use two extra input registers and compute tagged selective operations to perform a double handshake [Panc92], as discussed in Section 6. We note, however, that it is expensive (in terms of computation time) to implement this property in the framework hardware and it should be avoided when possible.

5. A Prototype Design of the Framework Hardware

This description is based on a completed design of a four-node prototype expected to be operational Summer, 1992. The host system for which the prototype has been designed is a Sparc Cluster, a group of Sparc-1e's which communicate through a VME backplane.

5.1. Auxiliary Processor

Each auxiliary processor in the prototype framework hardware is a 25 MHz Motorola 68020 microprocessor. This processor operates on 32-bit words. An AP has 256 Kbytes of RAM — expandable up to 1Mbyte — in order to store programs and data structures and 128 Kbytes of EPROM in which to store the boot-up monitor.

5.2. Host Processor - Auxiliary Processor Interface

The interface between HP and AP, i.e., the required FIFO queue and single set of registers, is implemented by a dual-ported RAM, such that the host processor is connected to one port and the auxiliary processor is connected to the other. Each of these ports is memory-mapped into the respective processor's address space. The two data paths are managed in the dual-ported RAM by software resident on the host and auxiliary processors; soft semaphores rely on the exclusive-write support provided by the dual-ported RAM. While the interface between an HP and AP could have been implemented more efficiently with hardware FIFO queues, the dual-ported RAM gives flexibility. For example, the data path from a host processor to an auxiliary processor could be changed to allow for data loss in the same way that the current path from AP to HP operates.

The host processor, a Sparc-1e, accesses the dual-ported RAM via its HP interface, which is the Sun SBus. The SBus has a bandwidth of about 100 megabytes per second for 32-bit words [SBus90]. We expect a potential throughput of 25 megabytes per second from host to auxiliary processor.

5.3. Parallel Reduction Network

The ALU's in the prototype parallel reduction network require 40 nanoseconds to perform a 32-bit fixed-point addition; hence, it will take 80 nanoseconds to perform a tagged selective operation. Fixed-point arithmetic is a good compromise between integer and floating point arithmetic. A minor cycle is projected to be 150 nanoseconds.

Thus, the time to produce values for state vectors of length *m* is 150 * *m* nanoseconds (plus the time to fill the pipe). With a more advanced and customized design, a shorter minor cycle, e.g. 50 nanoseconds, is achievable. We use off-the-shelf components in order to reduce the design time and cost of the prototype.

The pipelining in the prototype is performed synchronously. We selected the synchronous design to reduce potential problems such as race conditions and to extract speed reliably. An asynchronous design for the reduction network has been completed. In the asynchronous design, each ALU node of the PRN computes and outputs a result once it has completed an operation and two input values are available from the preceding stage. Each PRN node operates in a demand-driven manner, where operations are performed as both inputs become available. This asynchronous design is desirable for later versions of the reduction network for two reasons. First, a PRN operating asynchronously is scalable since a hardware handshake can be used to control communication between nodes; this eliminates both a central clock in the PRN and the potential problem of clock skew in a large network. Second, this facilitates the addition of floating point processors at each ALU node. A long operation, such as a floating point operation, forms a one-time "bubble" in the pipeline. With a synchronous network, the minor cycle must allow for the longest operation. Thus, a synchronous design creates wasted time when a shorter binary, associative operation is performed, and an asynchronous design alleviates this problem. We note that the synchronous design is simpler, and it is faster when only operations with uniform execution times are performed.

5.4. Auxiliary Processor-PRN Interface

The prototype hardware limits state vectors to size eight; each of the eight elements is a register pair, one 32-bit data register and one 32-bit tag register. The data register can be a 32-bit integer, a 32-bit fixed point number, or any 32-bit logical value, depending on the reduction operation to be applied. All numeric values are two's complement. The tag register can contain any 32-bit value. The PRN can be programmed to operate on state vectors of size two to eight, depending on the application.

6. Detailed Analysis

The framework hardware described in Section 4 can support a wide range of parallel simulations. In this section we give a summary of how an aggressive PDES synchronization protocol, such as Time Warp, can benefit from this hardware. Details of the algorithms executed on host and auxiliary processors appear in [Srin92].

One of the significant challenges facing aggressive protocols is the rapid dissemination of global virtual time (GVT). (See [Jeff85], [JeSo85], [Sama85], [LiLa89], [Bell90], and [CoKe91]) In an aggressive parallel simulation, GVT is the simulation safe time, the guaranteed time for which all events with timestamps less than or equal to it have been processed accurately. By definition, GVT is the minimum of two values: the minimum of all local clocks and the minimum of the timestamps of all transient messages [Jeff85]. A simulation cannot roll back to a time earlier than GVT. This is important for efficient memory management, which is integral to an aggressive PDES protocol; fossil collection can be performed on any state which is older than GVT. In addition to enhanced fossil collection, GVT is essential to committing to irreversible acts such as I/O and to detecting termination conditions [AbRi91].

We note that efficient GVT computation is most beneficial to a parallel simulation with a limited amount of memory. In a system with sufficient state memory, fossil collection is not as critical. Fujimoto has designed the rollback chip [FuTG92], a memory management unit that offloads state saving and state restoration inherent in aggressive PDES protocols. As reported in [BuRo90] the chip has excellent performance capabilities. However, the chip has one major limitation for which efficient GVT computation can compensate: limited capacity. Efficient GVT computation minimizes the unnecessary state information that must be stored in the rollback chip.

Another contribution our framework can make to an aggressive PDES protocol is the opportunity to limit the degree of aggressive processing. This is necessary if, for example, in the course of state saving, an LP's rollback chip's high-speed memory becomes full. If that LP stops processing aggressively, the risk of deadlock in the simulation is incurred. Saving state to a secondary store has obvious costs. Jefferson's "cancelback" mechanism [Jeff90] requires additional processing and message costs. In contrast, our framework is sufficient to avoid deadlock in a parallel simulation [Reyn92] without overhead from host processors or the host network. The LP whose local simulation time is equal to GVT *must* process its events in order to prevent deadlock. We note that the deadlock-breaking LP would require no state information to be retained prior to its local clock, meaning its rollback chip memory could be emptied.

It has been suggested that a limited aggressive PDES protocol, or *adaptive protocol*, one which is an intelligent combination of aggressive and non-aggressive protocols, may be more powerful than either in a pure form [LuWS91, Reyn88, Reyn92]. Adaptive protocols, however, have been neglected because of the costs associated with gathering and using synchronization information for adaptive decisions. In Section 7, our performance results show that global synchronization information can be efficiently reduced and disseminated using the framework hardware.

6.1. GVT Computation

Within our framework, GVT could be efficiently computed by an LP *at any time*. An LP executing an aggressive protocol on a host processor must communicate the following events to its AP: (1) completion of an event, (2) a rollback, (3) sending of an event message or antimessage, and (4) receipt of an event message or antimessage. For each event, the HP communicates a logical timestamp, and if the event is sending or receiving a message, the HP includes a unique message ID. The framework hardware guarantees that data sent from HP to AP is received in order.

When an AP receives information from its HP, it processes the information, computes inputs to reduction operations, and feeds a valid state vector to the PRN. The process of translating events from the HP into reduction inputs is performed such that correctness criteria of the application are never violated. For example, non-overwritable inputs are preserved.

In order to compute GVT, two globally reduced values are computed in the network: minimum next event time and minimum outstanding message time. When these globally reduced values are emitted from the reduction network, the AP reads these values and writes them into the dual-ported memory, which is readable by an LP executing on the host processor. The LP then calculates GVT as the minimum of these two globally reduced values.

6.2. Message Acknowledgements

The computation of GVT requires a minimum outstanding message time to be correctly maintained. The minimum outstanding message time is the earliest logical time of an event message that has been sent by an LP but not yet received at another LP. Message acknowledgements are needed to determine *when* it is safe to assume that a message is no longer in the host network. It is undesirable to acknowledge messages in the host network because the message traffic doubles. In [Panc92] a scheme for performing message acknowledgements with a *double handshake* in the reduction network was presented. We now discuss this approach.

As mentioned in the previous section, a host processor communicates all messages sent or received to its auxiliary processor. AP's submit messages to be acknowledged to the PRN in a state vector as a pair consisting of a message time and a message ID. A tagged selective reduction operation, i.e. minimum operation, is performed on the message times across all processors. Message ID’s are unique 32-bit values consisting of a message sequence number between a sender-receiver pair concatenated with a sender ID and a receiver ID. In order to support acknowledgements in the reduction network, each AP must maintain two sequence pools, one for messages sent (*outstanding message list*) and one for messages received but not yet acknowledged (*unacknowledged message list*). When a message acknowledgement becomes the global minimum across all processors, it is "acknowledged". The AP that submitted the acknowledgement then removes it so that other messages can be acknowledged.

It is crucial in the acknowledgement of messages that the LP that sent the message also see the message acknowledgement. The framework hardware, however, allows state vector loss at the output of the PRN. Hence, an LP may never see an acknowledgement, and its minimum outstanding message time will never advance. [Panc92] proposes a *double handshake* to guarantee that all acknowledgements are seen. In this scheme, a second tagged selective operation is used to "acknowledge the acknowledgement". In other words, the receiver cannot remove its acknowledgement until the sender echoes it using this second globally reduced value in the state vector.

Because the selective operation used for message acknowledgements is a minimum operation, the received message with the smallest timestamp in the system will eventually be acknowledged. This implies that the minimum outstanding message time will increase. As a result, GVT will increase as the simulation progresses.

This message acknowledgement scheme serializes acknowledgements in the reduction network. In order to optimize this, a contiguous *batch* of messages from the same sender can be acknowledged as sequences by acknowledging the message ID of the largest in-sequence message; the smallest logical timestamp of the message sequence is used as an AP’s input to the minimum operation [Panc92]. We now present simulation results which demonstrate the performance gains of using the framework hardware to acknowledge messages and to compute GVT.

7. Performance

In this section, we present the results of simulations conducted to evaluate the performance of the framework hardware described earlier in this paper. The two main motivations for simulating the hardware were:

(1) to observe its behavior under heavy load conditions, and

(2) to determine the approximate time it takes for a HP to compute globally reduced values using the framework hardware.

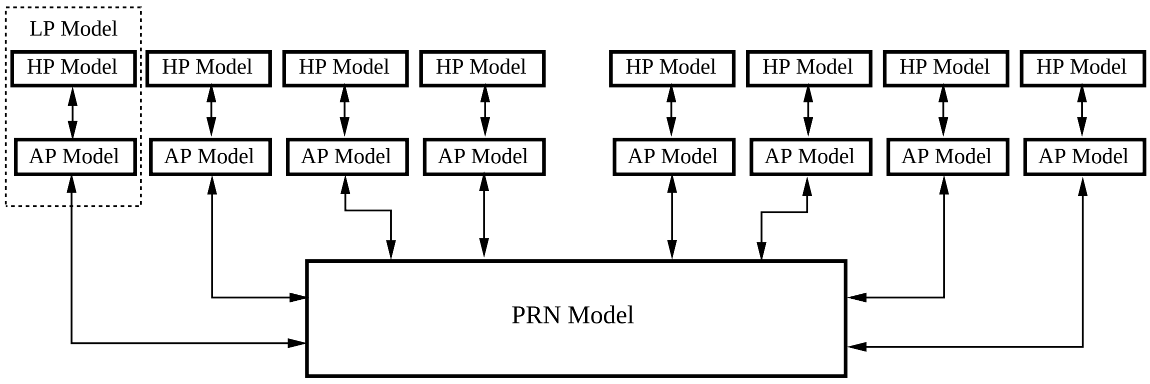

Figure 6. Structure of the Simulation Model.

We conducted two sets of simulations. The first set was aimed at stress-testing the hardware while the second one was to determine the time to perform a global reduction. In both cases we simulated a system of eight HP/AP pairs executing the framework synchronization algorithm described in [RePa92] and employing the framework hardware to compute the globally reduced values used in the algorithm. LP's executing on the HP's maintain two local values: the logical time of the next event and the smallest unreceived message time. Together, the global counterparts of these two values are used to avoid deadlock and to guarantee accurate simulation. In order to maintain the minimum unreceived message time, the LP's acknowledge messages.

The simulation model consisted of detailed functional models of the parallel reduction network and associated interfaces described in Section 4, a model for the AP's which execute the synchronization algorithm, and a model for the HP's which execute a dummy application. The nature of this application differed for the two experiments since they tested different aspects of the hardware. [Srin92] describes the detailed models built to simulate and test the framework hardware. We present an overview here.

The focus of the first experiment was to test the hardware under load. HP's perform only application specific tasks (processing events, sending and receiving event messages, state saving, rollback, etc.) while AP's perform only synchronization tasks (i.e., they execute the synchronization algorithm). In order for the AP's to correctly maintain the local values, this organization requires that HP's communicate with AP's under three circumstances: (1) when an event is processed, (2) when a message is sent, and (3) when a message is received. When an AP

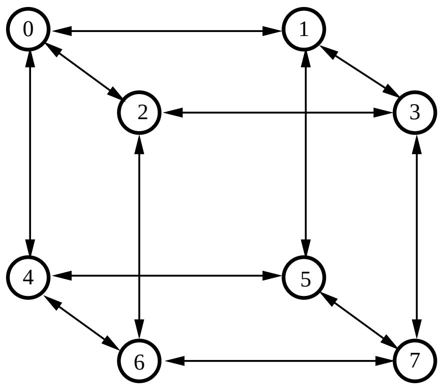

Figure 7. Communication Topology Used in All Simulations.

receives information from its HP, it processes that information, packs it into a valid state vector and then feeds this to the PRN. Messages are acknowledged in the reduction network using the double handshake described in Section 6.2. As a result, the AP is responsible for managing acknowledgements in addition to maintaining the two local values: minimum next event time and minimum logical timestamp of all unreceived messages. Since acknowledgements are performed one at a time in our implementation, each AP maintains a list of messages to be acknowledged, called the unacknowledged message list. We used the mean length of these lists as the metric for determining the behavior of the hardware under loaded conditions. This was done for the following reason. We define the load on the hardware as the rate at which information is presented to the hardware. As the load is increased, the AP would have more messages to acknowledge. Therefore, since acknowledgements are serialized, a reflection of the amount of loading would be the mean length of the unacknowledged message lists. By this token, the system is *unstable* when the unacknowledged lists grow unboundedly, i.e., when the messages to be acknowledged arrive faster than they can be acknowledged. We wished to determine the highest load at which an AP remains stable, by observing the effects of load variation on the lengths of the unacknowledged lists.

The load (rate of information flow from HP to AP) depends on the frequency of HP-AP communication which is controlled by two factors: (1) the rate at which the HP's process events and (2) the message traffic between the HP's. For this experiment, we did not simulate any real application on the HP's. Instead, we focused on controlling processing rate and message frequency. The structure of the model is shown in Figure 6. This figure does not indicate the fact that the HP's communicate with each other. The topology of this communication was parameterized. The specific topology used for the experiment is shown in Figure 7.

Each HP model executes the following loop:

```
loop forever
      Simulate the processing of an event
      Send a message to some other HP with some probability
endloop
```

Simulating an event involves simply delaying the HP for one event processing time period. Of course, the HP also receives messages. Receipt of a message causes no change in the HP's behavior, however. Upon receiving a message, the HP communicates this fact to the AP. The load is controlled by varying the mean time it takes to process an event as well as the probability of sending a message.

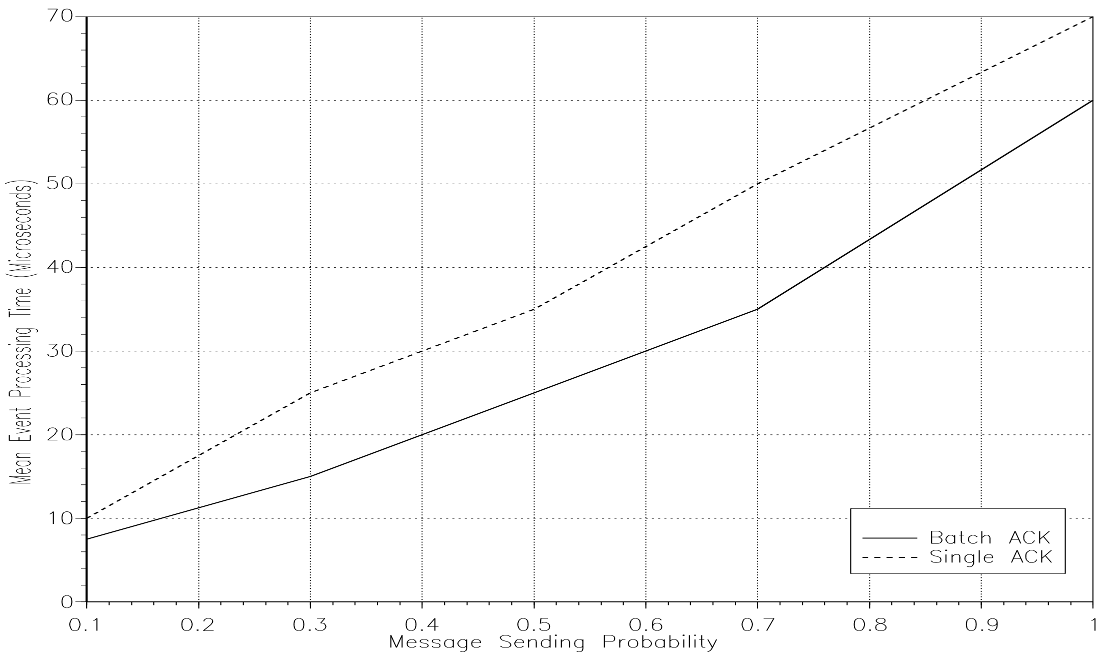

Figure 8. Saturation Points of the Framework Hardware.

We conducted performance tests using two versions of the model. In one, the AP’s acknowledged messages one at a time (the single acknowledgement experiment) while in the other, we implemented the batch acknowledgement enhancement (the batch acknowledgement experiment). We were thus able to observe the effects of this enhancement. We did not simulate the host communication network in great detail. Instead, we assumed that each message communication takes 10 microseconds. Such fast message communications are as yet unavailable. Even so, our simulations predict that the proposed hardware can withstand this order of magnitude improvement in processor interconnection network technology.

The two parameters which were varied during the simulations were mean event processing time and messaging probability. For several other parameters such as instruction execution times of processors, inter-stage data transfer time, etc., we obtained values from the prototype design where available and made realistic, best case estimates otherwise. The model output for each run consisted of the following relevant system-wide statistics:

- the mean length of the unacknowledged lists
- the mean time to complete a double-handshake acknowledgement
- the mean and maximum sizes of batches of acknowledgements
- the mean time between received messages (a measure of granularity)

7.1. Results of the First Set of Simulations

Figure 8 shows the saturation points of the framework hardware. We define a saturation point to be a point at which an increase in the load causes a drastic increase in the mean unacknowledged list length, indicating instability. For example, with 0.1 messaging probability and single acknowledgements, the smallest mean event processing time that is "safe" is 10 microseconds. A value smaller than this will cause the lists to grow unboundedly.

An important observation is the framework hardware is stable even when the granularity of events in the application being simulated is on the order of tens of microseconds. This is very encouraging since real applications typically have granularities on the order of milliseconds. This means our framework design can withstand at least two orders of magnitude performance improvement in host processors. The general shape of the curves is linear, as expected. As message sending probability increases, the message traffic increases and therefore the saturation point will also increase. A second, important conclusion is that since with batched acknowledgements the hardware saturates at smaller event granules, the batched acknowledgement enhancement does indeed improve the stability of

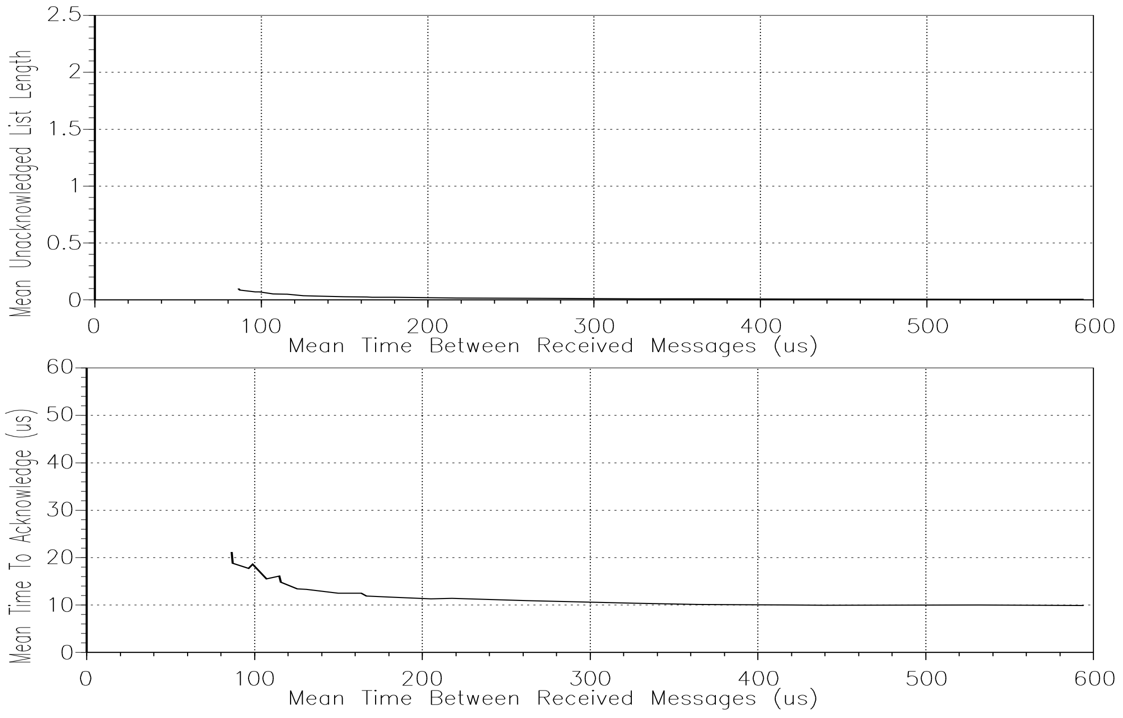

Figure 9. Performance with Single Acknowledgements.

the PRN. This will be more evident in the next set of graphs.

Figures 9 and 10 give another perspective on the saturation points of the PRN. Here, we plot the mean length of the unacknowledged message lists as well as the mean time to complete a double handshake acknowledgement versus the mean time between received messages, which is a function of the mean event processing time and the messaging probability, used in Figure 8. For all of the graphs, data points were obtained up to the point the framework became unstable. This explains the abrupt termination of some of the curves. In fact, we use this nature of the graphs to draw our conclusions.

The first observation is that the smallest granularity of message interarrival times that is "safe" appears to be around 67 microseconds. Of course, this number merely indicates the approximate region in which unstable behavior may be expected. It also appears that under normal load, the mean time to complete a double handshake acknowledgement is around 20 microseconds. The time to acknowledge a message was measured from the moment

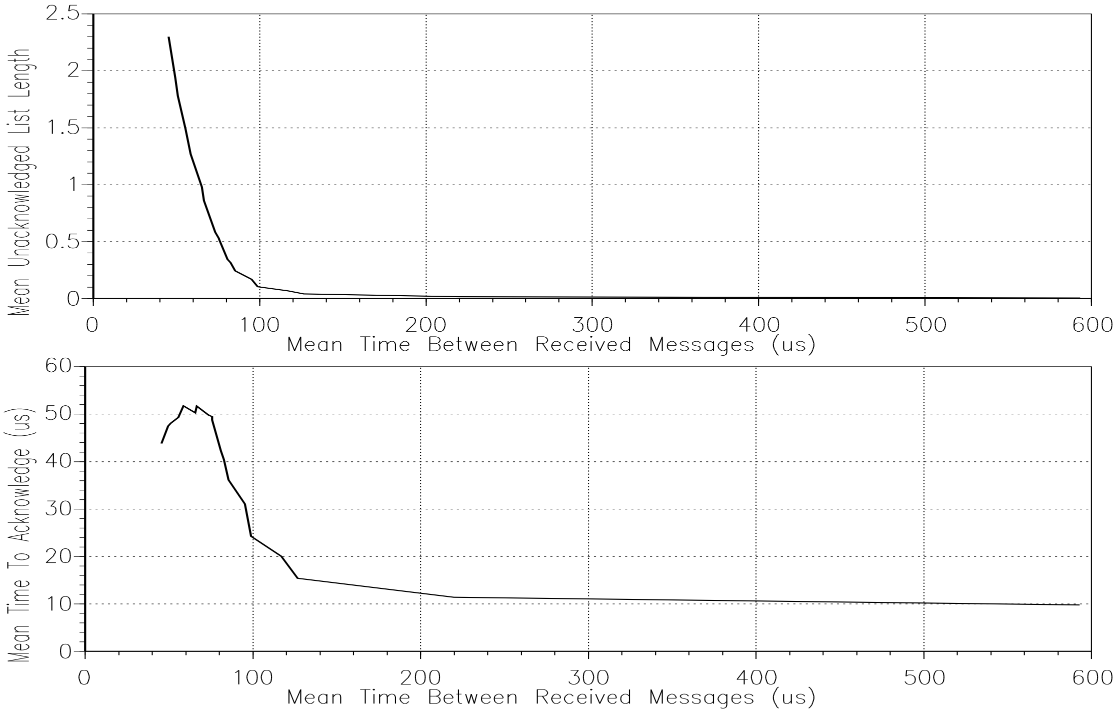

Figure 10. Performance with Batch Acknowledgements.

an AP starts an acknowledgement until it receives a second acknowledgement from the AP whose message it is acknowledging. This is a significant improvement over any existing communication system. Since a double handshake acknowledgement requires the computing of two global reductions, we can safely conclude that a global reduction will not take more than 10 microseconds. Note that here, we are talking about the average time it takes for a change in a local value to move from an AP through the PRN (possibly several times) before becoming visible to the AP at the output of the PRN and not the time it takes for a value to move from the input of the PRN to the output.

Recall that these graphs were obtained with a host communication time of 10 microseconds per message. This is a best case assumption for the host network. The host communication time affects the rate of message generation by the HP's (and therefore the load on the framework) in the following way: the faster messages travel through the host network, the faster they can be generated by the LP's. In effect, we have simulated the host network under low-latency, high bandwidth conditions. A higher host communication time would require a lower message generation rate, resulting in lower load on the framework. Since the host communication time we assumed here is at least an order of magnitude smaller than those provided by current interconnection technologies, we conclude that the framework can withstand at least an order of magnitude improvement in host communication network latency.

The graphs for batched acknowledgements extend up to queue lengths of 2.5. In the single acknowledgements case, the model becomes unstable very soon after the queue lengths approach 0.1. Batching of acknowledgements appears to make the system more robust. The reason for this is that as the load increases on the AP, the unacknowledged lists start growing. As a consequence, contiguous batches of messages form, and therefore, with batching of acknowledgements, the AP's perform more work per unit time than with single acknowledgements.

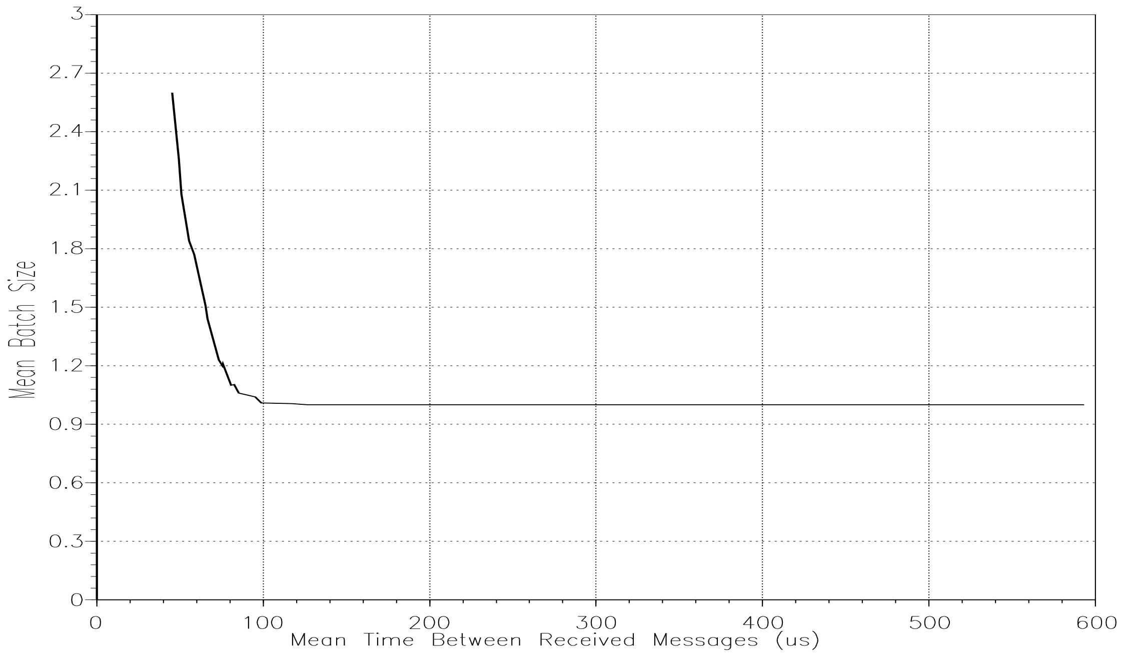

Figure 11. Effect of Load on Batch Size.

Finally, we note that with batched acknowledgements, the mean time to acknowledge a message is computed as:

$$\frac{\textit{mean time to acknowledge a batch of messages}}{\textit{mean size of a batch}}.$$

Therefore, under very heavy loads, since the mean batch size increases rapidly, the mean time to acknowledge a message actually begins to drop as can be seen from Figure 10. The variation of batch sizes with load is shown in Figure 11. As expected, the mean and batch size increases with smaller computation granularity (or higher load).

7.2. Results of the Second Set of Simulations

We used the same structure for the simulation model for the second experiment as that for the first one (Figures 6 and 7). The main difference was that the HP's were programmed to simulate a dummy application using the Time Warp protocol instead of the loop used in the first experiment. Thus, in this model, the HP's performed all the Time Warp specific activities such as state saving and rollback in addition to simulating the processing of dummy events. The model for the AP was also modified to execute an aggressive version of the framework algorithm [Reyn92] which is described in [Srin92]. In this algorithm, the framework hardware is used to rapidly compute GVT. The main focus of this experiment was to demonstrate the correctness of the proposed aggressive synchronization algorithm through simulations. But the model was also designed to compute the amount of simulated time by which the GVT computed by the PRN lagged behind the actual value of GVT. The amount of lag was computed as the average time from the moment an HP makes a change in a local value that causes a change in the actual value of GVT, until the value of GVT computed using the state vectors coming out of the PRN equals the new value of the actual GVT. We observed that under normal load, the GVT computed by the PRN lags behind the actual GVT by about 5-10 microseconds. If we include the time required for values used in the computation of GVT to be available to HP's after they emerge from the PRN, we conclude a HP will see the effects of a change in a value affecting GVT in about 10-20 microseconds.

In summary, our simulations have shown that the proposed hardware indeed satisfies its primary design goal: rapid computation and dissemination of global information. The hardware of course has its limits. Our simulations predict that with these limits, the hardware can withstand one or more orders of magnitude improvement in host processor and interconnection network technologies. Messages can be acknowledged in few tens of microseconds. Batching of acknowledgements makes the system more robust. Finally, the time for a change made by an HP to ripple down through the hardware and back up to the HP is also in the range of few tens of microseconds.

It is easy to estimate the time it will take a value to travel from the input to the PRN to its output. It is more difficult to estimate the time taken for a change made by an HP to be reflected in the globally reduced value. Our simulations predict that this time is very small. All our simulations were conducted for a system of eight LP's. As the number of LP's increases linearly, the raw time to compute a globally reduced value will only increase logarithmically. However, with an increased number of processors, the traffic to the PRN will also increase and it is reasonable to expect acknowledgements to take longer. However, we believe that the numbers for larger systems (for instance 32 or 64 processors) will be the same order of magnitude as those predicted by the simulations discussed here.

8. Related Work

Using a separate synchronization network for improving system performance is not a new idea. The IBM RP3 [PfBG85] was designed as a shared memory multiprocessor that houses both a combining network for synchronization traffic and a low latency network for regular message traffic. Our reduction network is not as complex or expensive as a combining network, yet it performs global synchronization operations very efficiently.

We claim no novelty with respect to reduction networks. Lubachevsky [Luba88] suggests using a binary tree implemented in hardware in order to support synchronization barriers and to compute and broadcast a minimum next event time in a bounded lag PDES. His control synchronization network is presented strictly in support of this PDES protocol. The Finite Element Machine [CrKn85, JoSc79], a NASA prototype, utilizes a binary tree-structured max/summation network to perform the global sum and maximum calculations necessary to support structural analysis algorithms. Like the hardware we propose, the sum and max calculations in the FEM are calculated alternately without processor synchronization. Our hardware design, however, employs a set of input and output registers which are treated as a single state vector, whereas the FEM uses a single input and a single output register.

At about the same time that we introduced our framework, Filoque, *et.al.*, [FiGP91] proposed the use of a processor network with programmable logic for efficient global computations, such as the computation of GVT in a Time Warp simulation. This hardware is not a single network like the PRN; it is, however, a distributed system of *sockets*, one per processor. The reprogrammable sockets are connected in a pipelined ring, forming the computation engine. A token is inserted into the ring by a designated control socket. It travels around the ring, performing partial computations at each socket. When the token returns to the controller, the global computation is complete. Therefore, their proposed hardware performs global computations in $O(n)$ time whereas the PRN performs the same computations in $O(\log n)$ time. Furthermore, the proposed synchronization algorithms for computing GVT in [FiGP91] rely on the host communication network for message acknowledgements and our framework uses the framework hardware for this purpose. The goals of both approaches are similar, but our framework is more efficient, more flexible, and more scalable.

Several researchers have proposed the use of hardware to implement barrier synchronization. Hoshino [Hosh85] has an efficient barrier synchronization in the PAX computer. Stone [Ston90] suggests the use of global busses to compute maximum values and to implement fetch-and-increment. The hardware that we propose, on the other hand, provides support for a larger class of algorithms than barrier synchronization algorithms.

Many parallel architectures provide for global binary, associative operations across all processors. Global operations on the Intel iPSC/2 [Inte89] are provided for arithmetic and logical operations. The Thinking Machines CM-5 [Thin92] contains two separate networks for different types of communication and synchronization: the data network is the primary message-passing network in the machine and the control network provides hardware support for common *cooperative operations*. The CM-5 control network supports "soft" barrier synchronization, arithmetic and logical reduction operations, parallel prefix operations, and segmented parallel prefix operations. The reduction operations on both of these machines require the complete synchronization of all processors. All processors must call global operation functions with a contributed value, and a global operation blocks until all processors enter it. Our framework hardware, on the other hand, computes and disseminates globally reduced values on state vectors *without* the coordination of the host processors; the reduction operations on the PRN are performed continuously. Furthermore, the hardware design employs auxiliary processors to manage the high-speed I/O between from the reduction network and itself.

9. Conclusions

Independent of any preference for a particular PDES protocol, be it Time Warp, blocking, windowing, or the like, rapid computation of synchronization values related to a particular protocol can yield significant benefits. The framework we have described here meets the correctness criteria set forth in Section 2. Also, it supports the computation of critical simulation-wide values with minimal interference with host processors and host communication networks. In fact, it is possible to disseminate all synchronization information through a framework such as ours and not use the host network for the dissemination of any synchronization information at all.

The overall effectiveness of our framework is yet to be determined; we expect, however, that our framework will enhance existing parallel simulation synchronization protocols and will advance the development of adaptive protocols which rely on the efficient dissemination of synchronization information. We have established that its rapid dissemination of GVT will benefit aggressive simulations needing to do fossil collection in limited memory. The prototype described here will be tested extensively beginning in the late Summer of 1992. With a design that has followed well-defined correctness criteria and with extensive simulation experience, we know the framework will operate correctly. What remains is a measure of its overall effectiveness. We expect to report on that important metric soon.

Acknowledgements

This research was supported in part by NSF Grant #CCR-9108448 and JPL Contract #957721. All simulations were performed using SES Workbench.

References

[AbRi91] Abrams, M. and Richardson, D., "Implementing a Global Termination Condition and Collecting Output Measures in Parallel Simulation", *Proceedings of the SCS Multiconference on Advances in Parallel and Distributed Simulation*, Anaheim, California, pp. 86-91, (January 1991).

[Ayan89] Ayani, R., "A Parallel Simulation Scheme Based on Distances Between Objects", *Proceedings of the SCS Multiconference on Distributed Simulation*, Tampa, Florida, pp. 113-118, (March 1989).

[Bell90] Bellenot, S., "Global Virtual Time Algorithms", *Proceedings of the SCS Multiconference on Distributed Simulation*, San Diego, California, pp. 122-127, (January 1990).

[BuRo90] Buzzell, C. A. and Robb, M. J., "Modular VME Rollback Hardware for Time Warp", *Proceedings of the SCS Multiconference on Distributed Simulation*, San Diego, California, pp. 153-156, (January 1990).

[ChSh89] Chandy, K. M. and Sherman, R., "The Conditional Event Approach to Distributed Simulation", *Proceedings of the SCS Multiconference on Distributed Simulation*, Tampa, Florida, pp. 93-99, (March 1989).

[CoKe91] Concepcion, A. I. and Kelly, S. G., "Computing Global Virtual Time Using the Multi-Level Token Passing Algorithm", *Proceedings of the SCS Multiconference on Advances in Parallel and Distributed Simulation*, Anaheim, California, pp. 63-68, (January 1991).

[CrKn85] Crockett, T. W. and Knott, J. D., "System Software for the Finite Element Machine", NASA Contractor Report 3870, NASA Langley, Hampton, Virginia, February 1985.

[DiRe92] Dickens, P. M. and Reynolds Jr., P. F., "State Saving and Rollback Costs for an Aggressive Global Windowing Algorithm ", Computer Science Report No. Technical Report-92-18, Department of Computer Science, University of Virginia, Charlottesville, Virginia, June 1992.

[FiGP91] Filoque, J. M., Gautrin, E. and Pottier, B., "Efficient Global Computations on a Processors Network with Programmable Logic", Report 1374, Institut National de Recherche en Informatique et en Anutomatique, France, January 1991.

[Fuji90] Fujimoto, R. M., "Parallel Discrete Event Simulation", *Communications of the ACM*, Vol. 33, No. 10, pp. 30-53, (October 1990).

[FuTG92] Fujimoto, R. M., Tsai, J. J. and Gopalakrishnan, G. C., "Design and Evaluation of the Rollback Chip: Special Purpose Hardware for Time Warp", *IEEE Transactions on Computers*, Vol. 41, No. 1, pp. 68-82, (January 1992).

[Hosh85] Hoshino, T., **PAX Computer: High-Speed Parallel Processing and Scientific Computing**, Addison-Wesley Publishing Company, Reading, Massachusetts, 1985.

[Inte89] Intel Corporation, **iPSC/2 Programmer's Reference Manual**, Intel Scientific Computers, Beaverton, Oregon, October 1989.

[Jeff85] Jefferson, D. R., "Virtual Time", *ACM Transactions on Programming Languages and Systems*, Vol. 7, No. 3, pp. 404-425, (July 1985).

[JeSo85] Jefferson, D. and Sowizral, H., "Fast Concurrent Simulation Using the Time Warp Mechanism", *Proceedings of the Conference on Distributed Simulation*, San Diego, California, pp. 63-69, (January 1985).

[Jeff90] Jefferson, D. R., "Virtual Time II: Storage Management in Distributed Simulation", *Proceedings of the Ninth Annual Symposium on Principles of Distributed Computing*, Quebec City, Quebec, Canada, pp. 75-89, (August 1990).

[JoSc79] Jordan, H. F., Scalabrin, M. and Calvert, W., "A Comparison of Three Types of Multiprocessor Algorithms", *Proceedings of the 1979 International Conference on Parallel Processing*, pp. 231-238, (August 1979).

[LiLa89] Lin, Y. B. and Lazowska, E. D., "Determining the Global Virtual Time in a Distributed Simulation", Technical Report 90-01-02, Department of Computer Science, University of Washington, Seattle, Washington, December 1989.

[Luba88] Lubachevsky, B. D., "Bounded Lag Distributed Discrete Event Simulation", *Proceedings of the SCS Multiconference on Distributed Simulation*, San Diego, California, pp. 183-191, (February 1988).

[Luba89] Lubachevsky, B. D., "Efficient Distributed Event-Driven Simulations of Multiple-Loop Networks", *Communications of the ACM*, Vol. 32, No. 1, pp. 111-123, (January 1989).

[LuWS91] Lubachevsky, B., Weiss, A. and Shwartz, A., "An Analysis of Rollback-Based Simulation", *ACM Transactions on Modeling and Computer Simulation*, Vol. 1, No. 2, pp. 154-193, (April 1991).

[Nico91] Nicol, D. M., "The Cost of Conservative Synchronization in Parallel Discrete Event Simulation", *to appear in Journal of the ACM*, (1991).

[Panc92] Pancerella, C. M., "Improving the Efficiency of a Framework for Parallel Simulations", *Proceedings of the 1992 Western Simulation MultiConference on Parallel and Distributed Simulation*, Newport Beach, California, pp. 22-29, (January 1992).

[PfBG85] Pfister, G. F., Brantley, W. C., George, D. A., Harvey, S. L., Kleinfelder, W. J., McAuliffe, K. P., Melton, E. A., Norton, V. A. and Weiss, J., "The IBM Research Parallel Prototype (RP3): Introduction and Architecture", *Proceedings of the 1985 International Conference on Parallel Processing*, St. Charles, Illinois, pp. 764-771, (August 1985).

[RaBJ88] Ranade, A. G., Bhatt, S. N. and Johnsson, S. L., "The Fluent Abstract Machine", YALEU/Department of Computer Science/Technical Report-573, Department of Computer Science, Yale University, New Haven, Connecticut, January 1988.

[Reyn88] Reynolds Jr., P. F., "A Spectrum of Options for Parallel Simulation", *Proceedings of the 1988 Winter Simulation Conference*, San Diego, California, pp. 325-332, (December 1988).

[ReWW89] Reynolds Jr., P. F., Williams, C. and Wagner, R. R., "Parallel Operations", Computer Science Report No. Technical Report-89-16, Department of Computer Science, University of Virginia, Charlottesville, Virginia, December 1989.

[Reyn91] Reynolds Jr., P. F., "An Efficient Framework for Parallel Simulations", *Proceedings of the SCS Multiconference on Advances in Parallel and Distributed Simulation*, Anaheim, California, pp. 167-174, (January 1991).

[ReWW92] Reynolds Jr., P. F., Williams, C. and Wagner, R. R., "Empirical Analysis of Isotach Networks", Computer Science Report No. Technical Report-92-19, Department of Computer Science, University of Virginia, Charlottesville, Virginia, June 1992.

[Reyn92] Reynolds Jr., P. F., "An Efficient Framework for Parallel Simulations", *to appear in International Journal on Computer Simulation*, (1992).

[RePa92] Reynolds Jr., P. F. and Pancerella, C. M., "Hardware Support for Parallel Discrete Event Simulations", Computer Science Report No. Technical Report-92-08, Department of Computer Science, University of Virginia, Charlottesville, Virginia, April 1992.

[Sama85] Samadi, B., "Distributed Simulation, Algorithms, and Performance Analysis", PhD Thesis, Computer Science Department, University of California, Los Angeles, January 1985.

[SoBW88] Sokol, L. M., Briscoe, D. P. and Wieland, A. P., "MTW: A Strategy for Scheduling Discrete Simulation Events for Concurrent Execution", *Proceedings of the SCS Multiconference on Distributed Simulation*, San Diego, California, pp. 34-42, (February 1988).

[Srin92] Srinivasan, S., "Modeling a Framework for Parallel Simulations", Master's Thesis, School of Engineering and Applied Science, University of Virginia, Charlottesville, Virginia, May 1992.

[Ston90] Stone, H. S., **High-Performance Computer Architecture**, Addison-Wesley Publishing Company, Reading, Massachusetts, 1990.

[SBus90] Sun Microsystems, **SBus Specification B.0**, Sun Microsystems, Inc., Mountain View, California, 1990.

[Thin92] Thinking Machines Corporation, **The Connection Machine CM-5 Technical Summary** , Thinking Machines Corporation, Cambridge, Massachusetts, January 1992.