On-the-Fly Calculation and Verification of Consistent Steering Transactions*

David W. Miller, Jinhua Guo, Eileen Kraemer, Yin Xiong
{miller, jinhua, eileen, xiong}@cs.uga.edu
Department of Computer Science
The University of Georgia

Abstract

Interactive Steering can be a valuable tool for understanding and controlling a distributed computation in real-time. With Interactive Steering, the user may change the state of a computation by adjusting application parameters on-the-fly. In our system, we model both the program's execution and steering actions in terms of transactions. We define a steering transaction as consistent if its vector time is not concurrent with the vector time of any program transaction. That is, consistent steering transactions occur "between" program transactions, at a point that represents a consistent cut. In this paper, we present an algorithm for verifying the consistency of steering transactions. The algorithm analyzes a record of the program transactions and compares it against the steering transaction; if the time at which the steering transaction was applied is inconsistent, the algorithm generates a vector representing the earliest consistent time at which the steering transaction could have been applied.

Keywords: *Program Transaction, Steering Transaction, Consistent Transaction, Consistent Steering, Program Event, Steering Event, Happened Before, Consistent Cut*

1. Introduction

Distributed computing is a powerful tool for performing large, computationally intensive tasks quickly by sharing the work across a network of workstations. However, the complexity of these programs can make it difficult for users to fully understand the runtime behavior of many of these computations [15]. The ability to observe controlled executions, interact with, and "steer" a running computation can aid understanding. We have developed an Exploratory Visualization (EV) system that allows a user to pose queries and visualize program data in a real-time fashion. Through this system, the user may monitor attributes and variables of the distributed computation. In addition, we are developing an Interactive Steering (IS) environment through which users may dynamically manipulate program variables or adjust resource allocation.

Consider a system for performing molecular mechanics calculations that contains modules for functions including distance geometry calculation, energy minimization, and free energy perturbation calculation, with the overall goal of 3D protein structure determination. The user might begin with an input file containing the approximate pairwise distances between atoms, as produced by an NMR study. An initial pass with a distance geometry model would produce a rough 3D structure. This structure would then be refined

* This material is based on work supported by the National Science Foundation under Grant No. 9996082.

SC2001 November 2001, Denver (c) 2001 ACM 1-58113-293-X/01/0011 $5.00

during a long-running energy minimization phase.

The standard protocols involved in this calculation may produce a structure that is overconstrained in one area and underconstrained in another area, or become trapped in a local energy minimum and fail to produce a reasonable 3D structure. The user might wish to visualize the intermediate results of these calculations and interact to turn constraints off or down in some areas of the molecule, and to apply or increase constraints in another area. It might be desirable to phase in the constraints rather than apply them all at once. (Note: a variety of ad-hoc protocols exist for the phasing-in of forces, but no one protocol is universally accepted.) In addition, the user might wish to load balance or make other performance adjustments during this long-running energy minimization phase.

The IS environment would allow the user to view the 3D structures, the state of the constraints, graphs of the change in total energy as the minimization runs, as well as performance statistics, and permit the user to select from a variety of protocols, to write a "steering agent" to implement a new protocol based on the data gathered from the energy calculations, or to rebalance the computational load by moving the computation associated with some atoms from one processor to another.

Both the visualizations and the steering activities rely on consistency. In the case of visualization, consistency guarantees that the visualization represents a valid state of the computation. In the case of steering, consistency guarantees that the steering operations are applied in a way that maintains the correctness of the computation(i.e., are causally consistent). For example, when moving an atom from one processor to another, consistency would guarantee that the energy calculation included each atom once and only once, that the atom being moved would not be either excluded or double-counted in the energy calculation. More subtly, consistency would guarantee that the new constraints would be phased in at all targeted atoms in the same way, unaffected by differences in speed of communications or processing at the participating processes. In this paper, we address the problems of detecting inconsistency, verifying consistency, and calculating consistent states.

2. Exploratory Visualization (EV) Environment

Given the size and complexity of distributed computations, it is important to have a presentation that provides a simple, accurate, and flexible view of an execution. To simplify the problem, in the EV environment, the overall computation is abstracted to an interleaving of atomic state changes involving one or more processes – by analogy to databases, such state changes are referred to as *transactions*. From a computational standpoint, transaction processing applications are a natural choice for obtaining global state information, since their structure matches the logical actions performed by the application [6]. For example, a money transfer may involve two processes located at different points of the network. It is desirable to treat the debit to one account and credit to another account as an atomic operation on the state of two bank accounts. Many multi-phased computations also fall into this category of applications whose structure reflects the logical computation. For example, the N-body problem employs alternating phases of communication and computation in which information about the state of neighboring regions is exchanged, and the forces on each body and the body's new position in space is then calculated. These phases may be modeled as transactions. Further, the transaction concept can often be superimposed on computations that otherwise execute in an unstructured manner.

The formal definition of transaction can be defined as follows:

Definition A transaction relation is an equivalence relation satisfying the following conditions:

1. A local transaction boundary is specified explicitly by end of transaction (EOT) annotations in an application program. A local transaction is a sequence of events between EOTs (or between the start of the program and the first EOT).
2. Two local transactions of different processes belong to the same global transaction if one local transaction sends at least one message to the other local transaction.
3. A global transaction consists of all local transactions in the same equivalence class.

We then view the local computation of a process as a sequence of local transactions and a distributed computation as a set of partially ordered global transactions.

The EV environment is a monitoring system that allows users to view and control a distributed computation in real-time. At present, the system supports PVM, MPI, and socket communication. Use of the EV system with a distributed computation requires that all communication calls be replaced by their EV counterparts and that the code be annotated with two types of statements: *watch* and *end-of-transaction*. The *watch* statement is used to indicate the variables of possible interest for monitoring or steering, and the *end-of-transaction* statement is used to logically group the code into blocks of computation, transactions, that represent logical actions performed by the application. Currently, this annotation is performed manually; interactive tools that largely automate this process could be implemented in a straightforward manner.

3. The Pathfinder Architecture

The EV environment is viewed as a three part architecture, known as the Pathfinder Architecture [5], composed of Interaction Managers (IM), a Snapshot Manager (SM), and a User Interface (UI), seen in figure 1.

The IM is composed of communication layers between the process and its communication environment. The IM implements a transaction labeling protocol. The IM collects information (local snapshots) from the processes that participate in each transaction, as well as the communication relationships between processes (membership), and forwards that information to the SM, which then calculates the dependence relationships between transactions (ordering). Processes that communicate with one another during the execution of a logical block of code that forms a transaction are considered members of the same transaction. However, each process typically knows only of its neighboring processes, those with which it directly communicated. Based on the transitive communication patterns, the complete membership within a transaction can be determined [18]. An example is seen in figure 2.

At the SM, globally consistent snapshots are generated based on the local snapshots from the IMs and the transaction labeling information. This labeling information may consist of neighbor lists, message counts, vector clocks, or other means [18]. In this paper, we use the vector clock approach as our basis for discussion. The transaction labeling information relies on a set of vector clocks that encode the inter-process communication that occurred during the represented transactions. This information can be used to group local snapshots into global snapshots that represent the global state of the computation at one instant in time [18]. Note that the existence of these vector clocks for purposes of visualization provides the foundation for the relatively low-cost addition of consistent steering.

Finally, the global snapshots are sent to the UI to be visualized. Through the UI, the user may pose queries to the system in order to gather application-specific and system-specific values to aid in overall understanding. The UI also provides the interface through which users may directly steer the distributed computation in real-time. The steering request will be issued directly to the SM for processing. Once

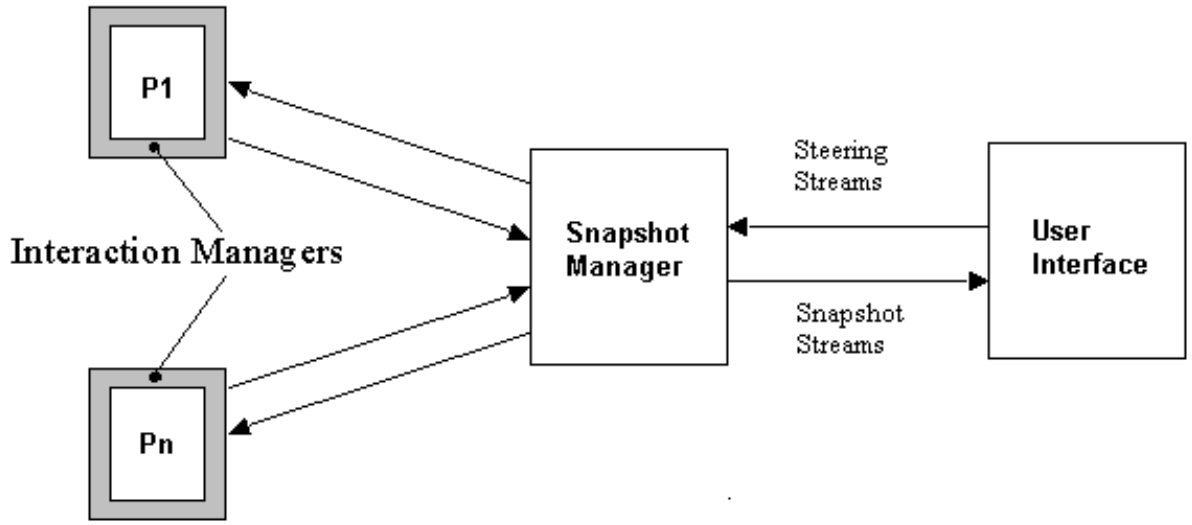


Figure 1 – Pathfinder Architecture
Application processes are "wrapped" in Interaction Manager libraries. The Snapshot Manager merges local snapshots into global snapshots. Contents of the global snapshots are visualized at the User Interface

the SM receives a steering command, it will issue a steering request to the IM at each process involved in the steering action. Each participating IM will report back to the SM the local process time at which the steering event occurred. The SM will use the event times to form a vector timestamp for the steering transaction. Based on the TLP information already present at the SM, together with steering transaction's vector time, the SM can determine if a steering command was applied consistently or not. The algorithm for determining consistency of steering transactions is the focus of this paper.

To illustrate the notion of consistent versus inconsistent steering actions, assume the energy minimization calculation described in the Introduction, and imagine that processors 1 and 2 are heavily loaded, while processor 4 is lightly loaded. The user issues a steering command, directing that some atoms be transferred from processors 1 and 2 to processor 4. Figures 3 and 4 illustrate the times at which the steering events of this steering transaction occurred at each involved process. In figure 3, since all the steering events took place either fully before or fully after each represented transaction, the steering command was applied consistently. However, in figure 4, the steering event for processes P1 and P2 occurred before transaction Pt1, but the steering event for process P4 occurred after transaction Pt1. Thus, an energy calculation performed during Pt1 might exclude the transferred atoms from consideration in the total energy, resulting in an incorrect energy calculation. In other words, because the steering events did not occur fully before or after each transaction, the steering command was applied inconsistently. Section 5.1 will expand upon the notion of *consistent steering*.

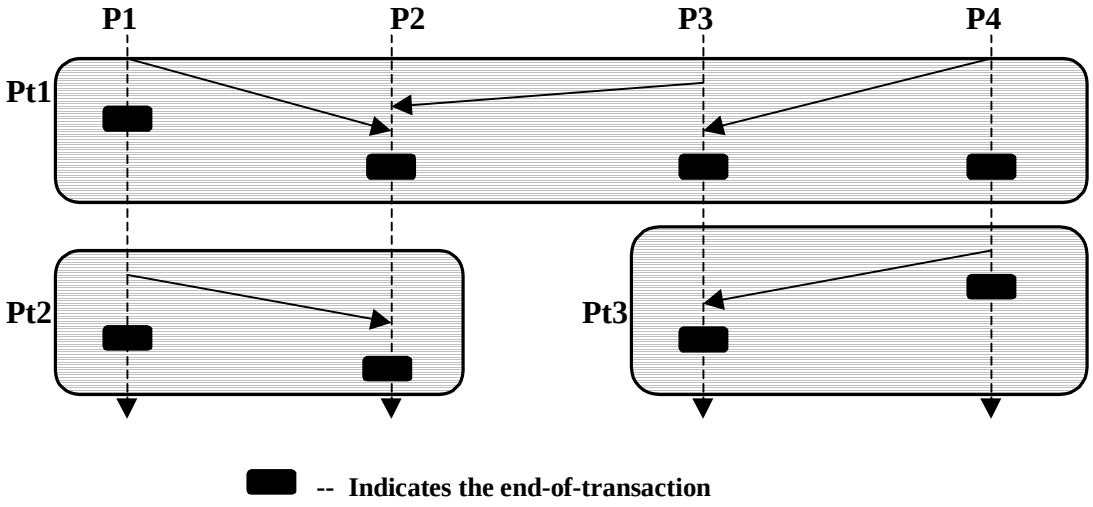


Figure 2 – Based on the end-of-transaction events and the communication pattern, three program transactions, Pt1, Pt2, and Pt3, can be formed.

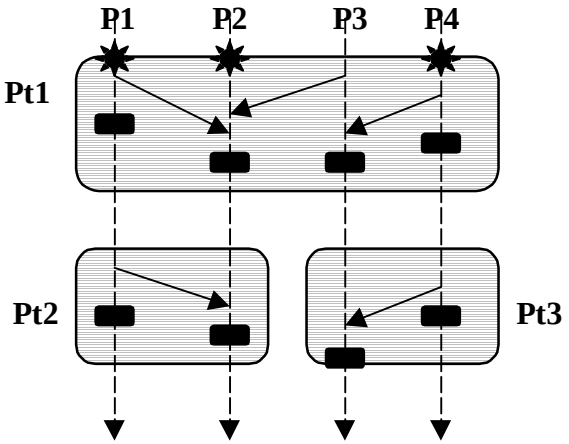


Figure 3 – Consistent Steering Transaction

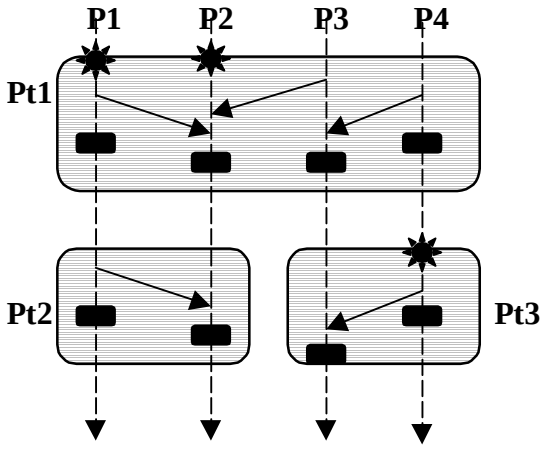


Figure 4 – Inconsistent Steering Transaction

-- Indicates the end-of-transaction
-- Indicates the steering events

4. Model and Definitions

A *distributed computation* consists of a set of dynamic processes that work together to achieve a common goal. In our system, each process exports two sets of attributes: one that reflects the subset of the process state available for monitoring and one that reflects the subset of the process state that is available for steering. The state of the process changes

when a program event occurs or when a user-specified steering event occurs. *Program events* of interest are *sends*, *receives*, and events that mark the end of a process's participation in a *transaction*, known as *end-of-transaction* events. *Steering events* are those actions directly issued by the user to make changes to the state of a computation. The change may be specified to occur at each process in the computation at which a variable resides or may instead affect multiple processes.

Any distributed computation can be decomposed into sets of program events. A *program transaction* is a set of program events collected at one or more processes and representing the same logical point in the program's execution. The events in the set are related through a direct or transitive communication pattern. The time of a program transaction is represented as a vector in which each process involved in a transaction has a timestamp representing its local time when it participated in the transaction. Program transactions are deemed *consistent* if

- the program events within the set belong to one and only one transaction,
- all related send-receive events are in the same transaction,
- for two program transactions Pt1 and Pt2, if the local events at one process in Pt1 occurred before the local events in the same process in Pt2, then, in every other process, all the local events of Pt1 must have occurred before all the local events of Pt2.

Finally, a group of related steering events can be organized into a set, a *steering transaction,* that represents the set of steering events resulting from a single request. These *transactions* are represented by vectors containing the local timestamps of the processes when they participated in the steering actions.

5. Optimistic versus Pessimistic Steering

Interactive steering may be implemented in two ways: as pessimistic steering or as optimistic steering. Pessimistic steering requires that a computation reach quiescence [2, 10] before a steering change is applied. This is a nontrivial process in a distributed, asynchronous environment in which centralized control of the computation is required. This model can result in considerable perturbation of the computation and significant steering lag [8].

In optimistic steering, the system receives a user-initiated set of steering events, grouped into a steering transaction, to make some application-specific modification. The system invokes each steering event in the steering transaction at the respective process without concern for or knowledge of the state of any other process. Therefore, the consistency of the global state of the computation, and specifically, the consistency of the steering transaction, must be checked. If the consistency is verified, the computation continues. If the steering transaction is found to be inconsistent, the computation must roll back to its state prior to the application of the steering change. In our approach, a new steering transaction is calculated; the process then executes forward to the new steering transaction time, applies the steering changes, and continues under normal execution. In this paper, we address the problem of verifying the consistency of a steering transaction, and, if needed, calculating a new consistent steering transaction.

Note that other systems for interactive steering typically do not address the problem of coordinated, distributed changes to general computations(Falcon[5]), perform steering only between iterations of some main loop(Cumulvs[14]), or leave the problem of consistency to be addressed by the programmer on a per-object basis (Magellan[16], Progress[17]). This third approach, however, provides the opportunity to go beyond causal consistency, and to perform other types of

consistency checking as well. Note also that the work described here is part of a larger investigation of optimistic steering**.** Whether optimistic steering is preferable to these other approaches, and if so, under what circumstances, remains to be seen and is an important element of our ongoing work.

5.1 Optimistic Steering

Let m = # processes participating in a distributed computation.
Let u, v be vectors of dimension m, each element a local timestamp of a process.

(1) $u \leq v$ iff $u[k] \leq v[k]$ for $k = 1, \ldots, m$
(2) $u < v$ iff $u \leq v$ and $u \neq v$
(3) $u \parallel v$ iff $\neg (u < v)$ and $\neg (v < u)$
[15].

That is, in cases (1) and (2), when applying the causality relationship defined by Schwarz and Mattern [15], the transaction represented by vector *u happened before* the transaction represented by vector *v* ($u < v$) if each process in *u* has a local timestamp less than or equal to that of the corresponding process in *v,* but the vectors are not identical. While, in case (3), the transaction represented by vector *u* does not *happen before* the transaction represented by vector *v,* nor does the transaction represented by vector *v happen before* the transaction represented by vector *u*. That is, the transactions correlating with vectors *u* and *v* are concurrent, represented by $u \parallel v$.

In our system, a steering transaction may involve a subset of processes, with the result that vector elements representing non-participating processes will be undefined. For example, figure 5 contains an initial vector representing a steering transaction in which process P2 participated at its local time 2 and process P4 participated at its local time 3. However, processes P1 and P3 did not participate, and, therefore, their entries are undefined. In addition, for participating processes, a steering event at time *t* is assumed to have happened prior to a program event at time *t*. For example, in figure 5, the program transaction shows process P2 participated at its local time 2 and process P3 participated at its local time 4. In both the steering transaction and the program transaction, process P2 has a corresponding timestamp of 2. For the purposes of our algorithm, it is assumed that steering transactions are applied immediately prior to program transactions with the same timestamp. Therefore, a steering event is said to *happen before* a program event with the same timestamp. For example, in figures 3 and 4, the steering event and program event representing the send of a message during Pt1 are both recorded with the same timestamp for process P1.

To correctly represent the transitive dependencies between a steering transaction and a program transaction, we must create an updated vector clock for the steering time. The elements representing the non-participating processes in the steering transaction are updated to represent the time of the earliest program transaction after the steering transaction at which those processes could have been affected. Figure 5 shows the update of the steering transaction that indicates process P3 was affected by the steering transaction at local time 4.

We define *consistent steering* such that a steering transaction represented by vector *v* is consistent if and only if it is not *concurrent* with any program transaction represented by vector *u*, denoted $\neg(v \parallel u)$. In section 6, the algorithm presented iterates over each program transaction beginning with the most recent and verifies that it is not concurrent with the steering transaction being checked. If during this process, the steering transaction is found to be concurrent with any program transaction, the algorithm will calculate the earliest, non-concurrent steering transaction that can occur after the present steering transaction.

P1	P2	P3	P4
	2		3

Steering Transaction

P1	P2	P3	P4
	2	4	

Program Transaction

P1	P2	P3	P4
	2	4	3

Updated Steering Transaction

Figure 5 – Demonstrates the updating of a steering transaction to indicate an indirect affects a steering

6. Algorithm for Consistency Verification

6.1 Idea behind Algorithm

In order to determine the consistency of a steering transaction, the system maintains a list of vectors representing a partial ordering of the program transaction history. In fact, the TLP algorithm used at the SM determines this total ordering. Transaction ordering ensures that the total ordering of snapshots is consistent with the partial order over program transactions. The algorithm described takes as inputs this history and a vector representing the time of the steering transaction. If the steering transaction is consistent, the algorithm returns TRUE. If the steering transaction is inconsistent, the algorithm returns a vector representing the earliest consistent time after the given steering transaction at which a steering transaction could occur.

To accomplish consistency detection, the algorithm creates a consistency vector representing a *consistent cut* [13] at which a steering transaction could be applied. The algorithm works backward, generating vector times for consistent cuts based on comparisons between the program transaction being analyzed and the steering transaction. The algorithm completes once the present steering transaction is reached or an inconsistency has been detected.

To determine membership, the algorithms tag each message sent with the local snapshot id. Receiving processes keep track of this information by a vector time and associate it with the current transaction. At the end of transaction, this vector time information will be exchanged between IMs or sent to the SM for the transaction membership assembly and transaction ordering. Since the vector time information is needed only at the end of a transaction and assembly of the transaction information does not block continued execution of the application processes, this should result in relatively low perturbation [18].

6.2 Algorithm

This algorithm, seen in figure 6, requires six data structures and one Boolean variable. First, a *TLP (Transaction Labeling Protocol)* table is used to maintain the chronological history of program transactions. Next, there are four vector times, a Boolean vector, and a Boolean variable*: TV (Transaction Vector), SV (Steering Vector), CV (Consistency Vector), CVTemp, Verified*, and *consistent*, respectively. Figure 7 shows an example of the initialized data structures. The *TV* vector holds the row of the *TLP* table

currently being analyzed. The *SV* vector contains the timestamps representing the steering transaction. The *CV* vector represents the time of a consistent steering transaction. The *CVtemp* vector provides a temporary holder of possible new timestamps for the *CV* vector. The values of *CVtemp* should not be committed to the *CV* vector until all elements of the *SV* vector have been compared against corresponding elements in the *TV* vector. Both the *CV* vector and *CVtemp* vector are initially empty. The Boolean *Verified* vector contains flags signifying that the earliest timestamp for a steering event to occur at each respective process has been verified. If a TRUE flag is present, then no new timestamp for that process should be added to the *CV* vector. *Verified* is initialized with all elements set to FALSE. Finally, the Boolean variable *consistent* indicates whether the values stored in *CVtemp* should be committed to the *CV* vector.

At the beginning of each iteration of the WHILE loop beginning on line 16, *consistent* is set to TRUE. This WHILE loop is used to determine the stopping point for the algorithm. The algorithm terminates once all elements of the *Verified* vector that correspond to elements of the *SV* vector have been set to TRUE. The key point of analysis occurs in the FOR loop starting on line 20. Here, each non-empty element of the *TV* vector is compared with the corresponding non-empty element of the *SV* vector. If the element compared in the *TV* vector holds a timestamp equal to or later than that of the element in the *SV* vector, then that timestamp is entered into the corresponding element in *CVtemp*. If *consistent* remains TRUE through all iterations of the FOR loop, then the values of *CVtemp* are committed to the *CV* vector. However, if any element of the *TV* vector occurred earlier than the corresponding element in the *SV* vector, then *consistent* will be changed to FALSE and the loop will terminate, as seen in lines 33 and 34. All entries in *CVtemp* are then purged and all elements of the *Verified* vector corresponding to elements of *TV* are marked as TRUE. This later action indicates that the present *TV* vector was concurrent with the *SV* vector. As explained, any concurrency implies an inconsistent steering transaction.

One other condition will cause the FOR loop to terminate without completing all iterations. If any element of the *TV* vector corresponding to an element of the *Verified* vector has already been set to TRUE, then all elements of the *Verified* vector corresponding to elements of the *TV* vector will be set to TRUE, and the loop terminates. As above, if any element of the *TV* vector has already been verified, then the earliest time at which a steering event could have occurred for that process has happened. Therefore, no other process having direct or transitive communication with that process could consistently apply a steering action during the program transaction represented by that *TV* vector or any earlier *TV* vector.

Once the condition has been satisfied that all elements of the *Verified* vector corresponding to elements of the *SV* vector have been set to TRUE, the WHILE loop will terminate and a comparison between the *CV* vector and *SV* vector occurs. If the *CV* vector and *SV* vector are found to be identical, the algorithm returns TRUE. The IS system can then purge all checkpoints and stop any message logging. If the *CV* vector is not equal to the *SV* vector, then the algorithm returns the *CV* vector. The IS system can then issue a command for each process to rollback to the checkpoint at the time specified in the *SV* vector, execute forward to the timestamps in the *CV*, then apply the steering command. Again, checkpoints can be purged and message logging may cease.

```
1.    Table TLP /*Table containing transaction history*/
2.    Vector TV /*Vector holding information about present transaction in TLP*/
3.    Vector SV /*Steering Vector containing list of processes involved in steering transaction*/
4.    Vector CV /*Consistent Vector representing when the steering transaction should take place*/
5.    Vector CVtemp /*Temporary vector to hold information until it is verified SV has not made TV
6.        inconsistent*/
7.    Vector Verified /*A Vector of Boolean values set to true when a process listed in SV is at its
8.        earliest logical time to have invoked the steering command*/
9.    Boolean consistent /*Boolean flag to indicate if SV has made TV inconsistent*/
10.
11.   BEGIN
12.
13.   set all elements of Verified to FALSE
14.   set TV equal to last vector of TLP table
15.
16.   WHILE (all processes in SV have not been set to true in Verified)
17.           BEGIN
18.                   consistent set to TRUE
19.
20.                   FOR (compare each corresponding, non-empty cell in TV and SV)
21.                           BEGIN
22.                                   IF(any non-empty cell in TV corresponds to a cell marked TRUE
23.                                   in Verified)
24.                                           THEN mark all cells in Verified corresponding to non-
25.                                            empty cells in TV TRUE
26.                                            set consistent to FALSE
27.                                            break
28.
29.                                   IF(TV is greater than or equal to SV)
30.                                           THEN set corresponding cell of CVtemp to TV
31.                                   ELSE
32.                                           Mark cells in Verified corresponding to non-empty cells in TV to TRUE
33.                                           and mark consistent to FALSE
34.                                           break
35.                           END
36.
37.                   IF(consistent)
38.                           THEN
39.                                   FOR(each non-empty element of CVtemp)
40.                                           set corresponding cells of CV to CVtemp
41.
42.                   IF(all cells of Verified corresponding to all cells SV are marked true)
43.                           break
44.                   ELSE
45.                           set TV equal to previous vector in TLP
46.           END
47.
48.   IF(SV equals CV)
49.           Return Consistent
50.   ELSE
51.           Return CV
52.
53.   END
```

Figure 6 – Consistency Verification Algorithm

6.3 Examples of Algorithm in Use

6.3.1 Inconsistency Detection

	P1	P2	P3	P4
T1	0	0	0	0
T2	1	1	1	
T3			2	1
T4			3	2
T5	2	2		

TLP TABLE

P1	P2	P3	P4
1	1	2	

SV VECTOR

P1	P2	P3	P4
2	2		

TV VECTOR

P1	P2	P3	P4
-	-	-	-

CV VECTOR

P1	P2	P3	P4
F	F	F	F

VERIFIED VECTOR

Figure 7 – Represents the initialization of the primary data structures at the beginning of the algorithm for a hypothetical execution of a distributed computation

Given the initial input above, we trace the updates to the *CV* and *Verified* vector as the algorithm proceeds. The following figures show the updates to the vectors *CV* and *Verified* after each iteration of the while loop.

Iteration 1

All non-empty cells of T5 are greater than the corresponding cells of the *SV*. The *CV* is updated to contain these values. The *Verified* vector is unchanged.

CV Vector

P1	P2	P3	P4
2	2		

Verified Vector

P1	P2	P3	P4
F	F	F	F

Figure 8 – Represents the updated states of the CV vector and Verified vector

Iteration 2

Again, the non-empty cells of T4 are greater than or equal to the corresponding cells of the *SV*. *CV* is updated accordingly.

CV Vector

P1	P2	P3	P4
2	2	3	

Verified Vector

P1	P2	P3	P4
F	F	F	F

Figure 9 – Represents the updated states of the CV and Verified Vector

Iteration 3

All cells of T3 are greater than or equal to their corresponding cells of the *SV*. Again, the *CV* is updated.

CV Vector

P1	P2	P3	P4
2	2	2	

Verified Vector

P1	P2	P3	P4
F	F	F	F

Figure 10 – Represents the updated states of the CV and Verified Vector

Iteration 4

Unlike the previous iterations, all the cells of T2 are not greater than or equal to the corresponding cells in the *SV*. In fact, the value in the cell for P3 is less than that in the *SV*. Therefore, all cells of processes in the *Verified* vector corresponding to cells of processes in T2 are marked TRUE. The *CV* vector remains unchanged.

CV Vector

P1	P2	P3	P4
2	2	2	

Verified Vector

P1	P2	P3	P4
T	T	T	F

Figure 11 – Represents the updated states of the CV and Verified Vector

Iteration 5

Finally, we make a comparison between T1 and *SV* and discover that the cells of T1 contain lesser values than the corresponding cells of the *SV*. We mark the corresponding cells of the *Verified* vector TRUE. Each cell of the *Verified* vector is now marked TRUE.

CV Vector

P1	P2	P3	P4
2	2	2	

Verified Vector

P1	P2	P3	P4
T	T	T	T

Figure 12 – Represents the updated states of the CV and Verified Vector

The *SV* is not equal to the *CV*. Thus, the algorithm returns the *CV*. A rollback to the *SV* is issued and the steering action will be applied at the *CV*, both seen in figure 13.

SV Vector

P1	P2	P3	P4
1	1	2	

CV Vector

P1	P2	P3	P4
2	2	2	

Figure 13 – Represents the original SV vector and the new consistent steering vector time CV

6.3.2 Consistency Verification

	P1	P2	P3	P4
T1	0	0		0
T2	1		0	1
T3			1	2
T4		1	2	
T5	2	2		3

TLP Table

P1	P2	P3	P4
	1	1	2

SV Vector

P1	P2	P3	P4
2	2		3

TV Vector

P1	P2	P3	P4
-	-	-	-

CV Vector

P1	P2	P3	P4
F	F	F	F

Verified Vector

Figure 14 – Represents the initialization of the primary data structures at the beginning of the algorithm for a hypothetical execution of a distributed computation.

Given the initial input above, we trace the updates to the *CV* and *Verified* vector as the algorithm proceeds.

Iteration 1

Since all the non-empty cells of T5 are greater than corresponding cells of the *SV*, we update the cells of the *CV* vector. The *Verified* vector remains unchanged

CV Vector

P1	P2	P3	P4
	2		3

Verified Vector

P1	P2	P3	P4
F	F	F	F

Figure 15 – Represents the updated states of the CV and Verified Vector

Iteration 2

Again, the non-empty cells of T4 are greater than or equal to the corresponding cells of the *SV*. The *CV* vector is updated. *Verified* is unchanged.

CV Vector

P1	P2	P3	P4
	1	2	3

Verified Vector

P1	P2	P3	P4
F	F	F	F

Figure 16 – Represents the updated states of the CV and Verified Vector

Iteration 3

All cells of T3 are greater than or equal to the corresponding cells of the *SV*. The *CV* is updated. *Verified* remains unchanged.

CV Vector

P1	P2	P3	P4
	1	1	2

Verified Vector

P1	P2	P3	P4
F	F	F	F

Figure 17 – Represents the updated states of the CV and Verified Vector

Iteration 4

Unlike the previous iterations, all the cells of T2 are not greater than or equal to the corresponding cells in the *SV*. In fact, the values in the cell for P3 and P4 are less than the corresponding values in the *SV*. Therefore, all cells of processes in the *Verified* vector corresponding to cells of processes in T2 are marked TRUE. The *CV* vector remains unchanged.

CV Vector

P1	P2	P3	P4
	1	1	2

Verified Vector

P1	P2	P3	P4
T	F	T	T

Figure 18 – Represents the updated states of the CV and Verified Vector

Iteration 5

Finally, we make a comparison between T1 and *SV*; the cells of T1 contain lesser values than the corresponding cells of the *SV*. We mark the corresponding cells of the *Verified* vector TRUE. Each cell of the *Verified* vector is now marked TRUE. Since *SV* is equal to *CV*, the algorithm returns TRUE. The IS system is now able to purge all checkpoints and cease all logging.

CV Vector

P1	P2	P3	P4
	1	1	2

Verified Vector

P1	P2	P3	P4
T	T	T	T

Figure 19 – Represents the updated states of the CV and Verified Vector

6.4 Explanation of Algorithm Correctness

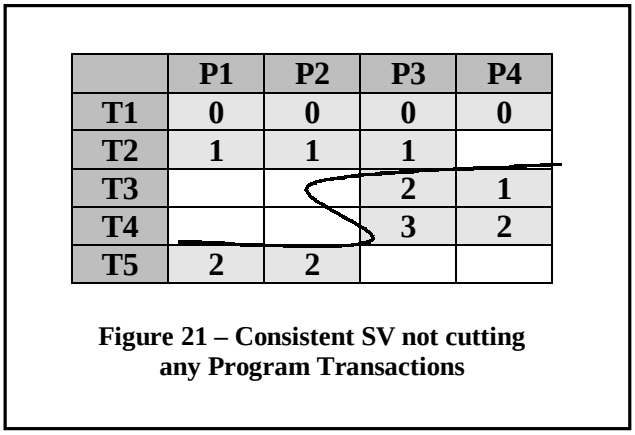
To clarify the point that the original *SV* in section 6.3.1 was not consistent, consider figure 20 containing the original *TLP* table. A bold line represents the points at which the steering transaction was applied. Note that program transaction T2 is cut by the original steering transaction; the steering transaction did not happen fully before or after program transaction T2, but rather was *concurrent* with T2.

	P1	P2	P3	P4
T1	0	0	0	0
T2	1	1	1	
T3			2	1
T4			3	2
T5	2	2		

Figure 20 – Original SV cutting Program Transaction T2

Figure 21 contains the original *TLP* table from section 6.3.1 but illustrates the new steering transaction generated by the *CV* vector. In this figure, it can be seen that no program transaction is cut by the steering transaction, and thus the steering transaction either happened before or after every program transaction but is not concurrent with any of

them. Therefore, this steering transaction is consistent. This same reasoning can be applied to the steering transaction in section 6.3.2 to verify that it too is consistent.



	P1	P2	P3	P4
T1	0	0	0	0
T2	1	1	1	
T3			2	1
T4			3	2
T5	2	2		

Figure 21 – Consistent SV not cutting any Program Transactions

7. Related Work

Causality is fundamental to many problems in distributed computing. For example, determining a consistent global snapshot of a distributed computation [1, 3] requires finding a set of local snapshots such that the causal relation between all events included in the snapshots is respected in the following sense: if e' is contained in the global snapshot formed by the union of local snapshots, and $e \rightarrow e'$ holds, then e must also be included in the global snapshot. Thus, the notion of consistency in distributed systems is basically an issue of correctly reflecting causality. Many important applications of causal consistency are summarized by Schwarz and Mattern in [15].

An important characteristic of distributed systems is that there is no global clock. Consequently, ordering the events in a distributed system can be challenging. Lamport [9] introduced an efficient mechanism called logical clocks for totally ordering the events in a distributed system, but the mechanism is not sufficiently powerful to allow concurrent events to be identified. Mattern [11] and Fidge [4] independently developed vector clocks, which precisely capture the causal ordering between distributed events. The main difference between Mattern and Fidge vector time schemes and ours is the way in which the logical time of a process is measured. Under the Mattern and Fidge schemes, the logical time of a process is measured in "number of past events" at that process. However, under our scheme, the logical time of a process is measured in "number of past local transactions" at that process.

The Z-path and Z-cycle notion, introduced by Netzer and Xu [13], generalizes the notion of a causal path of messages defined by Lamport's happened-before relation [9]. A local checkpoint is useless iff it is involved in a Z-cycle. In the transaction-based computational model, if a checkpoint is taken, all participant processes in a global transaction take the checkpoint at the end of transaction. In this way, no local checkpoint will be involved in a Z-cycle because there is no message in transit at the end of transaction.

8. Conclusion

The EV and IS systems offers a real-time environment through which users may gain understanding and perform on-the-fly program manipulation. However, with the ability to adjust application parameters on-the-fly, comes the necessity to verify that changes are made in a logically consistent manner, at *consistent cuts* [13]. Through the notions of steering transactions, program transactions, and consistent steering, the algorithm presented here provides a means to verify consistency, detect inconsistency, and calculate the earliest consistent cuts. Consistency guarantees that visualizations presented to the users represent a valid state of the computation and that the steering operations are applied in a way that maintains the correctness of the computation.

The currently developed EV environment and the prototyped IS environment permit users to configure the environments to provide the desired balance among consistency, lag, and perturbation of the underlying program execution. Included in this research is the development of a variety of

modular algorithms for the collection of snapshots with varying degrees of consistency, for either selective or comprehensive monitoring, as well as the development of algorithms that permit the consistent application of changes to the program in execution while minimizing the lag and perturbation that result. Also under study are techniques for improving the scalability of these algorithms. As the number of processes participating in steering transactions and the number of steering transactions increases, the message and processing load at the SM will increase significantly and the the SM will become a bottleneck. To address this problem, a hierarchical combination of SMs has been designed, but not yet implemented. Further, as the number of processes participating in steering transactions increases, the probability that a steering transaction will be consistent will likely decrease. Techniques to dynamically cluster related (steered together) processes under the same SM in the SM hierarchy are being considered to address this problem.

References

[1] K.M. Chandy and L. Lamport, "Distributed Snapshots: Determining Global States of Distributed Systems," *ACM Transactions on Computer Systems*, February 1985, 3(1):63-75.

[2] Dijkstra and B.P. Scholten, "Termination Detections for Diffusing Computations," *Information Processing Letters*, 1980, 11(1): 1-4.

[3] J. Fowler and W. Zwaenepoel, "Casual Distributed Breakpoints," in Proceedings, *10th International Conference on Distributed Computing Systems*, Paris, France, May 1990, pp. 134-141.

[4] C.J. Fidge, "Timestamps in Message-Passing Systems that Preserve the Partial Ordering", *Australian Computer Science Communications*, February 1988, pp. 56-66.

[5] W. Gu, G. Eisenhauer, E. Kraemer, K. Schwan, J. Stasko and J. Vetter, "Falcon: On-line Monitoring and Steering of Large-Scale Parallel Programs," in *Proceedings of the Fifth Symposium on the Frontiers of Massively Parallel Computation*, McClean, VA Feb 1995, pp. 422-429.

[6] D. Hart, E. Kraemer, and G.C. Roman, "Interactive Visual Exploration of Distributed Computations," in Proceedings, *11th International Parallel Processing Symposium*, Geneva, Switzerland, April 1997, pp. 121-127.

[7] D. Hart, E. Kraemer, "An Agent-Based Perspective of Distributed Monitoring and Steering," in Proceedings, *2nd Sigmetrics Symposium on Parallel and Distributed Tools*, Welches, Oregon, August 1998, pp. 151.

[8] E. Kraemer, D. Hart, and G.C. Roman, "Balancing Consistency and Lag in Transaction-Based Computational Steering," in Proceedings, *35th Annual Hawaii International Conference on Software Specification and Design*, January 1998, pp. 137-147.

[9] L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System," *Communications of the ACM 21*, 1978, 7(558-565).

[10] Lynch, N., *Distributed Algorithms*, Morgan Kaufmann Publishers, Inc., San Francisco, California, 1996.

[11] F. Mattern, "Virtual Time and Global States of Distributed Systems," *Parallel and Distributed Algorithms*, North-Holland, 1989, pp. 215-226.

[12] F. Mattern, "Efficient Algorithms for Distributed Snapshots and Global Virtual Time Approximation." *Journal of Parallel and Distributed Computing*, 18:423-424, 1993.

[13] Robert H. B. Netzer and J. Xu, "Necessary and Sufficient Conditions for Consistent Global Snapshots," *IEEE Transactions on Parallel and Distributed Systems*, February 1995, 6(2):165-169.

[14] P. M. Papadopoulos, J. A. Kohl, B. D. Semeraro, "CUMULVS: Extending a Generic Steering and Visualization Middleware for Application Fault-Tolerance," *Proceedings of the 31st Hawaii International Conference on System Sciences (HICSS-31)*, Kona, Hawaii, January 1998

[15] Reinhard Schwarz and Friedemann Mattern, "Detecting Causal Relationships in Distributed Computations: In Search of the Holy Grail," *Distributed Computing*, 1994, 7(3): 149-174.

[16] J. Vetter and K. Schwan, "High Performance Computational Steering of Physical Simulations," in *Proceedings of the 11th International Parallel Processing Symposium*, Geneva, Switzerland, April 1997, pp127-132.

[17] J. Vetter and K. Schwan. "Progress: A Toolkit for Interactive Program Steering," *Proceedings of the 1995 International Conference on Parallel Processing*, Urbana, IL, Aug 1995, pp. 139-142.

[18] H. Vuppula, E. Kraemer, and D. Hart, "Algorithms for Collection of Global Snapshots: An Empirical Evaluation," Proceedings in, *ICSA Conference on Parallel and Distributed Computing Systems*, August 2000, pp. 197-204.