The Dual Timestamping Methodology for Visualizing Distributed Applications

Brad Topol*
John T. Stasko*
Vaidy Sunderam†

May 1995

GIT-CC-95/21

Abstract

This article motivates and describes the dual timestamping methodology, a novel monitoring technique whose goal is to provide first class support for the visualization and animation of distributed and parallel applications. Central to this methodology is the use of both a primary and secondary timestamp in trace events. The primary timestamp is a logical timestamp that provides information about the concurrency of events. This information is useful for generating visualizations that depict the events as occurring in parallel. The secondary timestamp provides a normalized, causality preserving, real-time clock for use in performance visualization. The dual timestamping methodology is the basis for PVaniM, a collection of general purpose and application-specific visualizations of PVM applications. The implementation of PVaniM relies solely on macros and postprocessors. Because system modifications were not required, the PVaniM implementation strategies are general and easily adaptable to other distributed computing system domains.

*Author’s address: College of Computing, Georgia Institute of Technology, Atlanta, Georgia 30332-0280.
†Author’s address: Department of Math and Computer Science, Emory University, Atlanta, GA, 30322
1 Introduction

Visualization is emerging as a useful tool for the debugging and performance tuning of distributed and parallel applications. Visualization is necessarily dependent on some type of monitoring support such as event tracing. Unfortunately, developers of visualizations are rarely supplied with monitoring support that satisfies their needs. Typically, monitoring data lacks adequate timestamping, information regarding concurrency, and support for application-specific visualizations.

In this article, we motivate and describe a novel monitoring technique that can be adapted to a variety of distributed and parallel visualization methodologies. We begin with a brief overview of the relationship between monitoring and visualization, and discuss sampling and tracing, the two monitoring techniques on which visualizations rely. We also describe visualization techniques that are popular and commonly used. This article then introduces our approach towards distributed system event tracing. Central to our approach is the dual timestamping methodology that provides both primary and secondary timestamps in trace events. The primary timestamp is a logical timestamp that provides information on which events are concurrent and hence can be animated (visualized) in parallel. The secondary timestamp provides normalized causality preserving “wall clock” timestamps for use in performance visualization.

The dual timestamping methodology is the cornerstone of PVaniM, a framework we have developed to support the creation of general purpose and application-specific visualizations of distributed programs. PVaniM is an enhancement package for the popular PVM[Sun90] distributed computing environment. Section 3 describes PVaniM and its implementation which did not require any modifications to PVM; rather, it relies solely on postprocessors and macros. We feel these techniques are general and easily adaptable to other domains. Finally, Section 4 summarizes this work and discusses future directions.

2 Monitoring and Visualizing Distributed Applications

Monitoring and visualization depend deeply on each other; without proper monitoring, no information exists to “drive” a visualization. Without visualization, understanding the data derived from monitoring can be tedious and complex. In this section, we provide a brief background of both topics.

2.1 Monitoring Collection Techniques

Monitors utilize two fundamental techniques for information collection: tracing and sampling[OSS93]. In tracing, all occurrences of events are stored for a certain interval of time. Typically, this interval is for the duration of the distributed application. In sampling, occurrences of an event are collected asynchronously, typically only at the request of the monitor.

Tracing utilizes sensors which are small pieces of code that are embedded within the program and that perform the desired recording of information. Although complex techniques for developing sensors exist[OSS93], simple techniques are surprisingly useful. For example, many distributed systems[Sun90, BFS94] supply library routines for communication, synchronization and spawning of tasks. These integral events are traced by providing macro wrappers that first perform the tracing operation and then call the desired routines. Interesting events not related to any library routine may be traced by providing the user
with a function similar to `printf()` that allows events with custom application-specific data to be recorded.

Sampling may be performed by sensors or in some cases by probes, which reside in the monitor and directly access the address space of the application [OSS93]. Sampling is useful when one may only need cumulative statistics such as the number of sends and receives by a node at various stages of the execution of an application. Utilizing probes (when possible) can minimize the perturbation to the application that would be incurred had sensors been utilized.

### 2.1.1 Postmortem and Online Approaches to Monitoring

Another important consideration in monitoring is whether the information gathered is utilized online, i.e., while the application is executing, or in a postmortem fashion. For example, if tracing is utilized for postmortem analysis, extensive buffering of the recorded data is possible since analysis is deferred until after the application has completed. If tracing is used for online analysis, the events must be processed as soon as possible and extensive buffering would not be applicable.

Currently, our research has focused primarily on support for heterogeneous network computing environments such as PVM. This is due to their popularity, availability, and ability to support high-performance scientific applications in a variety of disciplines. In these environments, one must be careful because the monitoring also requires network bandwidth; this can inhibit the effectiveness of the distributed computing environment. For example, Xab [BS93], an online tracing tool for PVM, has trouble peacefully coinciding with applications it monitors. Our ongoing research with online tracing systems has found their perturbation to be excessive and hence their tracing is unreliable. Instead, we have relied on postmortem tracing systems and buffering techniques to achieve minimal perturbation and reliable results, while permitting the detailed information gathering achievable with tracing.

In general, we have refrained from utilizing sampling because tracing provides more detailed information, and hence provides more support for visualization. Consequently, this article solely focuses on tracing. Ongoing research suggests, however, that online sampling is a meritorious approach in various situations. An overview of online monitoring can be found in [Sch95].

### 2.2 Desired Trace Support

As previously mentioned, visualization is dependent on some type of monitoring that records the interesting aspects of a distributed system. These records subsequently are interpreted by the visualization tool to produce the graphical presentation. Typically, the annotations contain a timestamp, an event identifier, and event specific information. Because concurrent systems are built without a focus on subsequent graphical presentations, software visualizers are given very little support with regards to trace event profiling. However, this need not be the case if proper tracing is provided. The following are the tracing attributes we consider acceptable for the development of visualizations of applications in distributed computing environments:

- **Concurrency Information**—Modern visualization toolkits often support the notion of concurrent animations [SK93]; events in the distributed environment that are concurrent can be visualized as such. However, if the tracing does not provide information
as to which events are (or can be) concurrent, producing an accurate animation can be difficult. One possible approach, which we have previously used [TSS95], is the addition of a Lamport logical timestamp that provides information as to which events are concurrent in a distributed system. By animating all events with the same Lamport timestamp in parallel, a feasible concurrent visualization is achieved.

- **Sensible Performance Information Timestamping**—The "wall clock", or real-time timestamping provided by the distributed computing environment should not be naive. It is well known that the various clocks in a distributed system cannot be perfectly synchronized and will suffer from clock drift. Effort must be made to synchronize clocks as much as possible and to guarantee anomalies such as tachyons [BS93] do not occur. Tachyons result, for instance, when a message receive event is assigned a timestamp that is earlier than the timestamp of the send event associated with the message. Offloading this problem to the visualization system is simply unacceptable.

- **Support for Custom User Trace Events**—In many cases, application-specific visualizations are more desirable than general purpose visualization environments. An application-specific visualization typically can provide more information about the domain and fundamental operations of a program. These visualizations require custom trace events that are not well supported by generic trace formats such as PICL [G+90]. For this facet of visualization, it is necessary that custom user trace events be supported by the tracing package.

- **Minimal Perturbation**—Because many visualizations are used for performance tuning and the location of bottlenecks, distributed tracing support should minimize perturbation as much as possible to ensure an accurate visualization. A simple approach to minimizing perturbation used by systems such as PGPVM[TSA94] is the use of extensive buffered tracing. Some systems, such as AIMS [YSS94], also use explicit perturbation compensators that account for delays introduced by monitoring. Other approaches to minimizing or accounting for perturbation can be found in [MRW92, WAG+92, HM91].

These four attributes have driven the program tracing techniques that we have developed for the PVM distributed computing environment. In the next section, we provide a brief overview of the various techniques used to visualize distributed applications.

### 2.3 Visualization Techniques

Although visualization is still in its infancy, some general distributed computation visualization techniques are emerging. These include statistical displays, communication views, animations, and application-specific visualizations [KS93].

#### 2.3.1 Statistical Displays

Many performance visualization systems such as ParaGraph [HE91] rely heavily on statistical displays for the presentation of performance data. Commonly used statistical displays include bar charts, Kviat diagrams and utilization Gantt charts. These displays provide insight into the performance of a parallel application, and they rely heavily on real-time timestamps due to their performance oriented nature.
2.3.2 Communication Views

Communication views are utilized to represent the message passing between nodes in a distributed system. Typically, the topology of the processes and the interconnection network that is displayed matches the user's mental model of the topology of the distributed system. The ParaGraph system [HE91] provides a substantial set of topology specific communication views.

One popular communication view is the Feynman view, also known as a Lamport or space-time view. In it, process numbers are listed along the Y-axis. Time, whether it be real-time, or logical, is denoted on the X-axis. Messages are represented as lines drawn between the sending and receiving process. The X-coordinates of a line are determined by the send time and the receive time, and the Y-coordinates are determined by the processes' location along the Y-axis. A snapshot of ParaGraph's Feynman view is depicted in Figure 1. When the Feynman view utilizes real-time as its X-axis, the view provides both utilization information and general communication patterns. When the view utilizes Lamport logical time on its X-axis, the view not only displays a communication pattern, but also enforces a consistent ordering on a computation. Consistency is an important feature for testing and debugging, and is not achievable with a global real-time timestamp [Fid94].

2.3.3 Animations

Sophisticated graphical toolkits support the ability to simultaneously animate events that are concurrent. This approach can convey critical information to the viewer, information that simply cannot be achieved with a serialized view of a parallel application. An example of a concurrent animation is the Conch message passing view [TSS95], a snapshot of which is displayed in Figure 2. In this view, processes are laid out around the outside circle. When a process sends a message, a circle representing the message moves into the center of the circle in the general vicinity of the process that will receive the message. When the message is received, the message moves from its intermediate position in the ring to
the receiving process. When this display is used, broadcasts and general message patterns are represented in a clear manner to the user. These communication patterns produce distinguishable patterns that can be tracked and evaluated.

![Conch Message Passing](image)

Figure 2: Snapshot of Conch’s Message Passing view.

2.3.4 Application-Specific Visualizations

Application-specific visualizations are program depictions that are developed specifically for a particular application. For example, an application-specific visualization of a graph program might show the vertices, edges, and geometry of the graph in the manner we are most familiar with. Application-specific visualizations can be very informative when general purpose visualizations fail to provide the user with necessary information. A thorough discussion of application-specific views can be found in [SK93].

3 The Dual Timestamping Methodology

As previously touched upon, our dual timestamping methodology utilizes a primary and secondary timestamp in its event traces. Both timestamps contribute to provide knowledge regarding events in the distributed computing system. This section provides an overview of both timestamps. It then describes PVaniM, the software realization of the dual timestamping methodology, the views it provides, and details of its implementation.

3.1 Primary Timestamp—Lamport timestamp

In our methodology, all events are first ordered by a Lamport timestamp[Lam78] which is consistent with causality. Events that have the same Lamport timestamp can be thought of as concurrent events. By animating the events with the same logical timestamps in parallel, we are guaranteed that the visualization produced is representative of a plausible ordering of events. Note that other feasible orderings may exist also. For example, the Animation
Choreographer[KS94] is a tool that truly characterizes causality by allowing the user to choose a feasible ordering of events, and the corresponding concurrent visualization will be representative of the selected ordering.

3.2 Secondary Timestamp—Synchronized wall clock time

Although the Lamport primary timestamp is useful for program understanding, a synchronized wall clock timestamp is more suitable when utilizing visualization for performance tuning. Systems such as ParaGraph rely on this type of timestamp to provide its vast library of performance tuning views. As a secondary timestamp, our system provides a hybrid of physical and logical clocks as described in [Lam78] that is normalized at the beginning of application execution. This approach is also used by the PGPVM[TSA94] package. The hybrid clocking method is novel as it synchronizes clocks without setting them backwards and therefore without the risk of introducing tachyons into the system. The automatic removal of tachyons is important to the production of application-specific performance visualizations, especially for those who are uncomfortable with factors such as normalizing distributed clocks and compensating for clock drift. Towards the top of Figure 3 is a Gantt chart visualization that utilizes the secondary timestamp provided by our methodology.

The main contribution of our timestamping methodology is its support for program visualizers. Performance visualization is well supported as our performance tracing techniques are similar to PICL tracing that is used by the ParaGraph system. Further, parallel animations are well supported due to the concurrency and potential causality information provided by the logical timestamp. This tracing provides visualization support that cannot be matched by systems that rely solely on a wall clock timestamp or a logical timestamp.

3.3 Software Realization

In this section, we discuss the concrete manifestation of the concepts and principles described in the previous sections. Our realization of these techniques is the PVaniM visualization system developed for the PVM distributed computing environment. PVaniM provides animated program visualizations of the execution of PVM applications. In this section, we present the primary PVaniM views, and their dependence upon our timestamping techniques. We then present PVaniM's implementation of its tracing support, including a thorough description of its techniques for providing dual timestamps.

3.3.1 PVaniM Views

Figure 3 is a snapshot of views from the PVaniM view library. The views were built using the POLKA animation toolkit[SK93]. Near the top of Figure 3 is the Gantt view. This is PVaniM's adaptation of the ParaGraph Gantt view. It uses color to represent the various states of a processing node: a green rectangle represents computation; a yellow rectangle represents message sending; and red is drawn when a process is idle waiting to receive a message. Ours is a somewhat simplified version of the ParaGraph view because it automatically provides a time scaling such that the entire execution fits in the display window. This view relies on the secondary real-time timestamp, because it represents how much real time processes were computing, communicating, or idle. The view provides a global overview of process utilization and is a substantial aid for performance tuning.

Below the Gantt view in Figure 3 is the PVaniM Causality view. This view is an adaptation of the Lamport view provided by the Conch distributed computing environment’s
Figure 3: Snapshot of PVaniM graphical views.
visualization subsystem [TSS95]. In the PVaniM version of this communication view, the Y-axis is labeled with process identifiers and the X-axis is labeled with Lamport clock values. When a message is sent, a circle appears at the appropriate logical time coordinate. Varying circle radii are used to denote message size. When a message is delivered, an arrow "grows" from the coordinate of where the message was sent to the correct Lamport delivery time on the receiver’s timeline. Simultaneously, the circle representing the message moves along this path and then disappears. Arrows may be interactively queried by users which causes pertinent data about the message such as who sent the message, who received it, its type and length to be displayed in the PVaniM Information Box. The Causality view is highly dependent on the primary Lamport logical timestamp. First, it utilizes the concurrency information in the timestamp to determine which arrows should simultaneously "grow" (for message receipts) and which circles should simultaneously appear (for message sends). Second, the Lamport timestamp value is used as the unit of length on the X-coordinate axis. The result is a view that illustrates a consistent ordering of PVM communication events thus providing a view that is useful for debugging and verification of a distributed application.

Below the Causality view in Figure 3 is the PVaniM Message Passing view. This view is also an adaptation from views provided by the Conch system. In this view, all the processes are laid out around the outside circle. Messages are represented as circles that smoothly move into the center of the ring of processes when sent, and end up in the vicinity of the process that the message is intended to be received by. When a process receives a message, the message moves from its position in the central area of the ring towards the process, and disappears after arriving at the process. This view is an example of a concurrent animation. Concurrent communication is represented as the circles moving simultaneously to their respected destinations, whether it be towards the center of the circle (i.e. send events) or toward a process at the edge of the circle (i.e. receive events). This view is especially useful for detecting communication patterns, undelivered messages, and large messages. The view is heavily dependent on the primary Lamport timestamp to provide a concurrent animation of communication in the distributed system.

3.3.2 Developing Application-Specific Visualizations

PVaniM’s dual timestamping traces coupled with the POLKA toolkit provide strong scaffolding for those who wish to create their own application-specific visualizations. Our justification for this is twofold. First, the POLKA toolkit provides rich support for concurrent animations. POLKA allows the user to first schedule graphical events to happen simultaneously at a particular frame time. The frames are generated (i.e. the animation actually executes) using an animation primitive. The user has total control over when new frames are generated. POLKA has been successfully used by many programmers with little or no graphics experience. Second, a PVaniM trace is sorted by its primary Lamport logical timestamp which implies that concurrent trace events (i.e. those with the same Lamport timestamp) are adjacent to each other in the trace file. As long as the application-specific visualization uses POLKA’s primitives to concurrently animate trace events with the same Lamport logical timestamp value, the user is guaranteed that a feasible concurrent animation is being produced. Pseudo code for reading a trace file, “programming” and “animating” concurrent events is provided in Figure 4. These factors, coupled with PVaniM’s support for user trace events (discussed thoroughly in the next section) provide a palatable environment for developing application-specific visualizations, even by those with minimal
understanding of the notion of concurrent events. PVaniM and POLKA are currently being used to develop application-specific views for a High Performance Fortran system and for developing views for parallel branch and bound algorithms. In the next section, a detailed overview of the current implementation of PVaniM is provided.

3.3.3 Implementation

PVaniM's dual timestamping methodology is implemented utilizing macros and postprocessors. An architectural overview of its techniques is shown in Figure 5. No modifications to PVM were required for its implementation. For complex “production” systems such as PVM that are seemingly continuously updated and revised, this approach is simply more practical than modifying native system source code. For other distributed communication environments, system integration may be a feasible approach. The techniques discussed in this section for performing dual timestamping can be integrated into a system using an approach similar to that used by the Conch distributed computing environment [TSS95], as all postprocessing techniques are derivatives of online distributed algorithms introduced in [Lam78]. This section provides a thorough description of our macros and postprocessing techniques. Since this approach is achievable without source code modifications, it should be readily portable to other distributed computing environments, both experimental and commercial.

Tracing Instrumentation  The PVaniM tracing library uses macro wrappers to add its tracing to the PVM communication primitives for applications written in the C or C++ programming languages. This technique is well established; it has been used in previous online tracing systems such as Xab and previous postmortem tracing systems such as PG-PVM. The user adds a header file that redefines communications primitives to be PVaniM stubs. These stubs perform the tracing and then call the appropriate communication primitive. Our tracing produces two trace events for communication events, one before the PVM primitive and one after. This allows us to determine the duration of the communication primitive. We have adapted this approach from PICL [G+90]. This allows us to produce trace events that contain dual timestamps as well as pertinent communication specific data. Figure 6 shows example lines from a PVaniM trace file.

Timestamp Creation  Crucial to tracing is, of course, the creation of timestamps. Our desire is to have both a causality preserving Lamport timestamp as well as a hybrid physical and logical timestamp to serve as a “wall clock” timestamp. Lamport provides algorithms for both of these that require the “piggybacking” of clock information onto messages that are sent in the distributed system [Lam78]. Integrating these types of algorithms into systems (and avoiding the need for postprocessing) clearly has several advantages [TSS95]. However, the contribution of the nonintegrated approach described in this article is that its techniques apply to distributed systems in general. They may even be used by systems in which communication subsystem modification is not a viable option.

The initial phase of the timestamping method utilized by the system is an optimistic one. At the beginning of an applications’ execution, it invokes an initialization routine provided by our system. In this routine, gettimeofday() is used to produce a reference timestamp. All future timestamps are produced as the difference of the current time and the reference timestamp. In this initial phase, the Lamport timestamp is simply a monotonically increasing integer that is incremented before each trace event. This approach is clearly
while(/*trace file is not empty*/) {

    readEvent(&curtraceinfo);  // Read a trace event from file, 
    // store pertinent parts in curtraceinfo

    if (curtraceinfo.lam_time == curl_time) {  // If this event has same Lamport 
        // timestamp as the current 
        // Lamport value, this is a 
        // concurrent event

        len = programEvent(curtraceinfo, animruntime);  // Program the event to happen 
        // at the current working 
        // frame(animruntime) This will 
        // program the necessary animation 
        // routines

        maxlen = MAX(len, maxlen);  // Store the maximum number of 
                             // frames that have been created 
                             // so far.

    } else {
        go(animruntime, maxlen);  // No more concurrent events 
                             // exist at this Lamport time, 
                             // we now animate the events 
                             // we have previously programmed, 
                             // and then program this event for 
                             // the next round of animation 
                             // Call proper animate routines 
                             // to animate the views for 
                             // maxlen number of frames.

        curl_time = curtraceinfo.lam_time;  // Update current Lamport time

        animruntime += maxlen;  // Update current frame number

        maxlen=0;

        len = programEvent(curtraceinfo, animruntime);  // Program event, storing the 

        maxlen = MAX(len, maxlen);  // number of frames of animation 
                             // this event will require.
    }
}

Figure 4: Pseudo code for a PVaniM application-specific visualization.
Application Process

PVM Macros utilized to perform monitoring.

Buffered trace files produced.

Automatic collation of trace files at end of execution.

Tracefile postprocessing to produce timestamps.

PVaniM views may now be utilized.

Figure 5: Overview of PVaniM architecture.

Figure 6: Sample lines from a PVaniM trace file. The first column contains the event type. Column two contains the logical timestamp. The third and fourth columns compose the hybrid real-time timestamp. The fifth column contains the process identifier, and the remaining columns represent event-specific information.
optimistic: because there is no “piggybacking” on messages of clock information, it would be rare and fortuitous that either of these timestamps actually be causality preserving. A rigorous postprocessor is provided that adjusts timestamps to simultaneously emulate the use of both Lamport’s logical clocking and his hybrid clocking algorithm.

Clock Synchronization Postprocessing Our postprocessing techniques are based on those developed for the PGPVM system [TSA94] and similar to Beguelin’s work on post-processing trace events [BS93]. The PVaniM postprocessor differentiates itself by providing support for a pure logical timestamp in addition to a hybrid logical and physical timestamp. PVaniM also supports and corrects timestamps associated with custom user trace events. The PVaniM postprocessor creates an acyclic directed graph and utilizes a topological sorting variant to adjust both timestamps. Consecutive events on the same process will have a directed edge in the graph, and edges are inserted whenever a process sends a message to another process. The Lamport timestamp of an event is adjusted to be greater than the maximum timestamp of any event that has a directed edge towards this event. This emulates Lamport’s algorithm for pure logical clocks.

The adjusting of the hybrid timestamp is slightly more complicated. If an event is a non-receive event, the hybrid timestamp is only adjusted to guarantee that its timestamp is greater than the previous event on the same process. This is only necessary if adjacent events on the same processor received the same timestamp. The adjustment requires the addition of a “clock tick” and is necessary to satisfy Lamport’s requirement that events on the same process cannot have equal timestamps. If an event is a receive event, the algorithm also consults the timestamp of the “send event” which is paired with the receive event currently being adjusted. The timestamp of the receive event is adjusted to be the maximum of its current value (with one clock tick adjustment, if necessary) and the timestamp of the send event plus a minimum message delay time (referred to as $\mu_m$ in Lamport’s vernacular [Lam78]). Conceptually, we take communication time into consideration when adjusting the clock, if necessary. This procedure for adjusting clocks continues until the entire graph has been traversed. After this process is complete, the Unix sort utility is used to sort trace events first by Lamport clock value and then by wall clock timestamp value.

Lamport proves that the above algorithm performs true clock synchronization if a sufficient number of “heartbeat” messages are sent by the processes. In our implementation, the application’s communication serves as the “heartbeat” messages. Therefore, the quality of the synchronization is dependent upon the amount of communication inherent in the distributed application. Although in many of our applications enough communication is present to provide substantial clock synchronization, this may not always be the case. In applications with minimal communication, these techniques only guarantee that timestamps have been normalized and are causality preserving. As our goals are pragmatic ones, the minimal guarantees provided by the hybrid real-time timestamp appear satisfactory. Nonetheless, future work will investigate if more rigorous clock synchronization can be implemented without system modifications and without causing undue perturbation to the distributed application.

To summarize, we have implemented an online logical clock and a hybrid logical and physical clock adjustment algorithm, both of which are described in [Lam78] by emulating them with a postprocessor. This approach is useful when modifications to a distributed system are not a feasible option. This algorithm, coupled with buffered postmortem tracing, has yielded very satisfying results. In addition, our current postprocessor is portable to
anyone who subscribes to these timestamping and tracing methodologies.

**Application-Specific Trace Support**  In the current version of PVaniM, the user may add application-specific trace events, trace information that cannot be discerned from the generic PVaniM tracing. PVaniM provides a `pvanim.print` routine that allows the user to write a character string with a maximum length of 512 bytes to the trace file. This technique is adapted from application-specific visualization support provided by the Conch system [TSS95]. The `pvanim.print` routine prepends this string with an event identifier, a process identifier and the dual timestamps. This information is required for all types of trace events used by PVaniM. PVaniM first reads the prepended information and uses the event identifier to determine what type of data follows the “standard” fields. If the event is a user trace event, the postprocessor stores all data it finds after the standard fields as a string, until it reaches an end of string character. With this approach, the PVaniM postprocessor is able to adjust timestamps of both standard trace events and custom user trace events. These features are currently being utilized when more esoteric data from an application needs to be visualized.

A substantial amount of work has been done in the area of application-specific monitoring [Sno88, KS91, OSS93]. Typically, special languages are provided to allow the user to specify the application-specific aspects of a program that need to be monitored. This is then used to produce application-specific sensors and probes. Unfortunately, these monitors typically do not provide the timestamping support that can be crucial to the development of visualizations and animations. For example, the Issos monitor [OSS93] does not address the need for synchronizing wall clock timestamps, nor does it provide support for logical timestamps. Although these may be innocuous issues when monitoring tightly coupled multiprocessors, they are in our opinion crucial when monitoring loosely coupled distributed systems. Clearly our methods for providing application-specific tracing are simplistic; they are nonetheless easy to use and provide appropriate timestamping for our domain.

### 4 Discussion

In this paper, we have described the implementation and usefulness of a novel timestamping methodology which relies on both a primary Lamport logical timestamp for concurrency and (potential) causality information and a secondary normalized causality preserving physical and logical hybrid timestamp for performance information. We have used this dual timestamping technique as the foundation for PVaniM, a performance and program visualization system for the PVM distributed computing environment. PVaniM supports a variety of visualization and animation techniques, as well as support for application-specific visualizations. We describe the implementation of PVaniM which required no modifications to PVM. Since these dual timestamping techniques do not require system communication modifications, they are easily adaptable to other system domains. Future research includes the further revision of PVaniM’s general purpose view package, as well as utilizing its facilities for developing novel application-specific views for various PVM applications.

### Acknowledgments

This research was supported in part by the U. S. Department of Energy grant DE-FG05-91ER25105, the National Science Foundation grants NSF ASC-9214149 and CCR-9121607,
and the Office of Naval Research grant N00014-93-1-0278.

References


