Background Execution of Time Warp Programs [1]

Christopher Carothers and Richard M. Fujimoto
College of Computing
Georgia Institute of Technology
Atlanta, GA 30332-0280
chrisc,fujimoto@cc.gatech.edu

[1]This work was supported by U.S. Army Contract DASG60-95-C-0103 funded by the Ballistic Missile Defense Organization, and NSF Grant Number CDA-9501637.

Background Execution of Time Warp Programs

Abstract

A load distribution system is proposed to enable a single Time Warp program to execute in background, spreading over a collection of possibly heterogeneous workstations (including multiprocessor hosts), utilizing whatever otherwise unused CPU cycles are available. The system uses a simple processor allocation policy to dynamically add or delete hosts from the set of processors utilized by the Time Warp program during its execution. A load balancing algorithm is used that allocates logical processes (LPs) to processors, taking into account other computations executing on the host from the system or other user applications. A clustering mechanism is used to group collections of logical processes together, reducing process migration overheads and helping to retain locality of communication for simulations containing large number of LPs.

An initial, prototype implementation of the load distribution system is described that executes on a homogeneous network of Silicon Graphics workstations. Initial experiments indicate this approach shows promise in enabling efficient execution of Time Warp programs "in background" on distributed computing platforms.

Background Execution of Time Warp Programs

1 Introduction

Many large-scale discrete event simulation computations are excessively time consuming, and are a natural candidate for distributed processing. Because of the large amount of computation required, and the availability of large networks of workstations, an attractive paradigm is to allow the distributed simulation to execute "in background," spreading across the available workstations in the network and making use of whatever otherwise unused CPU cycles are available. Because these computations should not degrade performance of interactive users or other higher priority computations, the distributed computation must be able to withdraw from any CPU on short notice if the need arises.

Time Warp is a well known synchronization protocol that detects out-of-order executions of events as they occur, and recovers using a rollback mechanism [10]. Time Warp has demonstrated some success in speeding up a variety of simulation applications, including combat models [17], communication networks [14], wireless networks [4], queuing networks [6], and digital logic circuits [1], among others. We assume that the reader is familiar with the Time Warp mechanism described in [10].

With few exceptions, most research on distributed simulation to date assumes the simulation program is allocated a fixed number of processors when execution begins, and has exclusive access to these processors throughout the lifetime of the simulation. Specifically, interference from other, external, computations is minimal, and no provisions for adding or removing processors during the execution of the simulation are made. In fact, in most experimental studies, one typically goes to great lengths to eliminate any unwanted external interference from other user and system computations in order to obtain performance measurements that are not perturbed by external workloads. While this paradigm makes perfectly good sense from an experimental standpoint, this "dedicated platform" scenario is often not the prevalent paradigm one encounters in practice. In particular, networks of desktop workstations and distributed compute servers consisting of collections of workstation-class CPUs interconnected through high-speed LANs have become prevalent. Despite the continual, reduced cost of computing hardware, shared use of computer resources will continue to be a common computing paradigm in the foreseeable future.

The goal of this study is to examine the "background" execution of Time Warp programs on distributed computing platforms in the presence of other external computations utilizing the same hardware resources, and to develop simple, but effective techniques to enable the Time Warp program to spread over the network and "soak up" whatever CPU cycles are available. Specifically, the background execution scheme must

- dynamically allocate additional CPUs during the execution of the distributed simulation as they become available and migrate portions of the distributed simulation workload onto these machines,
- dynamically release certain CPUs during the simulation as they become loaded with other, external, computations, and off-load the workload to the remaining CPUs used by the distributed simulation, and
- dynamically re-distribute the workload on the existing set of processors as some become more heavily or lightly loaded by changing, internal or externally induced workloads.

The last point, load distribution in the face of external computations, is particularly important for Time Warp because load imbalance can lead to very poor performance. The external workloads competing for the CPU with the Time Warp program may be user or system computations. A Time Warp program that is well balanced when executed on dedicated hardware may become grossly unbalanced when executed on machines with external computations from other users. Logical processes (LPs) that are mapped to heavily utilized processors will advance very slowly through simulated time relative to others executing on lightly loaded processors. This can cause some LPs to advance too far ahead into the simulation future, resulting in very long or frequent rollbacks. Thus, it is essential that any Time Warp simulation running in "background" take into account the other, external, workloads utilizing the same CPUs the Time Warp program is using.

The policies and mechanisms described here are intended to be used in large distributed simulation applications containing many logical processes, e.g., thousands or more. Because of the scale of these simulations, managing individual logical processes is too time consuming. Here, a clustering approach is used where LPs are organized into groups, and a group is the smallest unit that is moved between processors.

The next section describes related work concerning workload management of Time Warp programs. An approach to enable efficient background execution is then described. An implementation of the mechanism is discussed, followed by results from initial measurements of the prototype system.

2 Related Work

Background execution is essentially a load management problem. Traditionally, load balancing involves distributing the simulation workload across a fixed set of processors in order to minimize the elapsed time to execute the program. Here, the traditional load balancing problem must be extended in two ways. First, the load balancing mechanism must take into account dynamically changing external workloads produced by other computations. These external loads cannot be controlled by the load management software. Second, the set of processors that can be utilized by the distributed simulation expands and contracts during the execution of the program in unpredictable ways.

Optimistic synchronization mechanisms introduce new wrinkles to dynamic load management: high processor utilization does not necessarily imply good performance because a processor may be busy executing work that is later undone. Further, there is a close relationship between load management and the efficiency (e.g., amount of rolled back computation) of the synchronization mechanism, as discussed earlier. These factors necessitate development of new load management techniques specific to Time Warp. Thus, systems such as Condor [11] that distribute jobs onto networked workstations as background processes that "soak up" otherwise unused CPU cycles are not sufficient for Time Warp simulations.

Dynamic load management of Time Warp programs has been studied by others. Reiher and Jefferson propose a new metric called *effective processor utilization* which is defined as the fraction of the time during which a processor is executing computations that are eventually committed [15]. Based on this metric, they propose a strategy that migrates processes from processors with high effective utilization to those with low utilization. Reiher and Jefferson also propose splitting a logical process into *phases* to reduce the amount of process state that must be moved when an LP migrates from one processor to another, and present performance measurements of their scheme. Glazer and Tropper propose allocating virtual time-slices to processes, based on their observed rate of advancing the local simulation clock [8]. They present simulation results illustrating this approach yields better performance than the Reiher/Jefferson scheme for certain workloads. To our knowledge, this scheme has not been implemented on an operational Time Warp system. Goldberg describes an interesting approach to the load distribution that replicates bottleneck processes to enable concurrent execution [9]. Time Warp is used to maintain consistency among the replicated copies.

None of these approaches addresses the question of background execution. The approach proposed here does utilizes the ideas of not considering rolled back computation in deriving load balancing metrics, and workload allocation based on the rate of simulated time advance in developing an approach for background execution.

Burdorf and Marti [3] propose an approach to periodically compute the average and standard deviation of all the LP local clocks in the system. If the average local clock among the LPs mapped to a processor is greater than the system-wide average plus one standard deviation, it is concluded that this processor is advancing too rapidly through simulated time, so additional LPs are migrated to that processor to "slow it down." Specifically, the LP with the smallest local clock is moved. In addition, other LPs that have low virtual clocks (specifically, a local clock less than the system-wide average minus one standard deviation) are moved to the processor that has the LP with the largest local clock. Marti and Burdorf observe that this approach will balance the workload in the presence of external computations competing for the same processors.

Schlagenhaft et. al. [16] propose an approach to balance the load of a VLSI circuit application on a distributed Time Warp simulator in the presence of external workloads. They define an inverse measure of the load, called *Virtual Time Progress*, which reflects how fast a simulation process continues in virtual time. This approach has some similarities with ours, however neither their approach nor Burdorf's approach address the question of dynamically changing the set of processors utilized by the simulation.

The central contribution of the work described here is in devising and demonstrating an approach to enable efficient, background execution of Time Warp programs on networked workstations where the simulation is able to dynamically migrate over whatever CPUs are available to execute the simulation. A prototype system and initial performance measurements are described demonstrating the viability of this approach.

3 The Workload Management Policy

The load management policy used here consists of two components:

1. The *processor allocation* policy that defines the set of processors that may be used by the Time Warp program. In general, this *usable set* of processors will change dynamically throughout the execution of the distributed simulation.

2. The *load balancing* policy that migrates LPs between processors in the usable set. This policy must maintain efficient execution, in spite of dynamically changing external workloads in the processors in the usable set. It is assumed the Time Warp system has no control over these workloads, nor control over priority of execution in the operating system of these external computations relative to the Time Warp program.

Dynamic load distribution of individual LPs becomes burdensome for simulations containing large numbers (say, thousands) of LPs. This is because a large number of entities must be considered by the load balancing algorithm, increasing the computation required for load distribution, and load balancing information that must be maintained. Further, because migrating each LP requires a certain amount of overhead, independent of the "size" of the LP, migrating many LPs from one processor to another is less efficient than migrating a group of LPs as a unit. Here, LPs are first grouped (by the application) into *clusters* of LPs, and the cluster forms the atomic unit that can be migrated from one processor to another. In addition to reducing load management and process migration overhead, this approach will keep LPs that frequently communicate together, on the same processor, provided they are grouped within the same cluster.

3.1 Processor Allocation

The hardware platform is assumed to be a collection of possibly heterogeneous processors interconnected by a local area network (LAN). Individual *hosts* of this network may be single processor workstations or multiprocessors. The processor allocation software monitors the computation load assigned to each each processor and derives the *usable set* of processors that may be used by the Time Warp simulation. This usable set is passed to the load balancing policy for assignment of LP clusters to processors.

At present, a simple approach to processor allocation is used. A central process is responsible for monitoring the processors that *may* be used by the distributed simulation. This process executes periodically, at a user defined *scheduling interval* $T_{schedule}$. This process estimates the expected amount of CPU time that would be allocated to a Time Warp simulation if it were to execute on that host, based on the current workload of this host over the last schedule interval. A daemon-like process executes on each processor that is responsible for providing load information on that processor. If a processor is not in the current usable set and the expected CPU allocation (divided by $T_{schedule}$) is above a user defined threshold *AllocThreshold*, then the processor is added to the usable set. On the other hand, if the processor is already included in the usable set, but the expected CPU allocation falls below a second user defined threshold *DeAllocThreshold*, then the processor is removed from the usable set. A straightforward extension of this approach is to allow external computations, e.g., created by the user that owns the CPU, to evict the distributed simulation when that user begins using the machine.

The expected CPU allocation is computed in one of two ways. If the processor is currently in the usable set, then the expected CPU allocation is simply the amount of CPU time the Time Warp program received on that processor during the last scheduling period. On the other hand, if the processor is not in the current usable set, the expected CPU allocation is computed from the processor's load average, obtained by a system call.

3.2 Load Balancing

The load balancing policy is responsible for assigning LP clusters to processors in the usable processor set. Like the processor allocation policy, the current implementation of the load balancing policy uses a central process that executes periodically every $T_{schedule}$ seconds to determine which clusters should be moved to another processor.

The load balancing policy attempts to distribute clusters across processors to equalize the rate of progress of each processor through simulated time, taking into account both the external (non-Time Warp) and internal (Time Warp) workloads assigned to each processor. The central metric that is used to accomplish this is the *processor advance time (PAT)*. The processor advance time indicates the amount of *wall-clock time* required for a processor to advance one unit of simulated time in the absence of rollback. By using wall-clock time rather than CPU time of the Time Warp simulation, the external workload on the processor is automatically taken into account. By only considering committed computation, rolled back computation is not allowed to be treated as additional computation load. The load balancing policy moves clusters from processors with large PAT values to those with lower values with the goal of minimizing the maximum difference between the PAT values in any two processors.

PAT values are easily measured for the current mapping of clusters to processors, as discussed below. In order to assess the effect of redistributing clusters, another mechanism is required to estimate how well (or poorly) the load will be balanced if a hypothetical move of cluster(s) between processors is performed. For this purpose, the *cluster advance time (CAT)* metric is defined. CAT is defined as the amount of computation required to advance a cluster one unit of simulated time, again in the absence of rollback. Unlike PAT, CAT is determined assuming an unloaded processor, so it is independent of the external workload assigned to the processor. In a heterogeneous computing environment, CAT for a single cluster will differ from one processor to another according to the processor's computation speed.

More precisely, we define:

1. $CAT_{c,i}$ is the estimated amount of computation time required by cluster c to advance

one unit of simulation time on host i, measured in seconds. In the current implementation, this value is computed for a "standard" host processor by an estimate of the number of *committed* events processed over the last sample interval (recall load balancing decisions are made periodically) times the average amount of computation time required per event, divided by the net advance in simulated time achieved by the cluster over the sample period. A empirically derived scaling factor is used to convert this number between host architectures.

2. $TWFrac_i$ is defined as the fraction of the total CPU cycles that the Time Warp computation was allocated over the last scheduling interval. At present, this value is computed by dividing the amount of CPU time allocated to the Time Warp program over the last interval by the wall-clock time that elapsed over that time interval.

3. PAT_i is defined as CAT_i / $TWFrac_i$ where CAT_i is defined as the sum of all CAT values of clusters mapped, or hypothesized to be mapped, to processor i.

If cluster c is moved from processor i to processor j, then PAT_i is reduced by the amount $CAT_{c,i}$ / $TWFrac_i$ and PAT_j is increased by $CAT_{c,j}$ / $TWFrac_j$. The new PAT values reflect the expected wall-clock time for each processor to advance one unit of simulation time after the move is made.

In addition to balancing the workload, a second goal of the load balancing policy is to minimize the number of cluster moves required to achieve an acceptable load balance. This is accomplished by prioritizing clusters moves so that clusters with large CAT values are given precedence over cluster with smaller CAT values. Thus, the algorithm will attempt to move one cluster with large computation requirements rather than several clusters with small computation requirements, assuming the move results in a better load distribution.

In addition, load migration is only performed if the maximum difference in PAT values between any pair of processors exceeds a user defined threshold. This avoids performing migrations when the benefit that can be realized by the migration is modest.

The load balancing algorithm attempts to minimize the maximum of $(PAT_i - PAT_j)$ over all i and j. The algorithm is shown in Figure 1. Processors added to the usable processor set are assigned a PAT value of zero, making them prime candidates for accepting new clusters.

```
Function DLMAnalyzeStatistics()
   For Each Processor, PE[i]
      if( PE[i] Is ACTIVE  AND PE[i]'s Load Average > HIGH-THRESHOLD )
         Set PE[i] to INACTIVE;
         UnLoad ALL Clusters From PE[i] to an ACTIVE Processor;
         Set PE[i]'s PAT to INFINITY; /* So SORT Puts Them out of Range */
      if( PE[i] Is INACTIVE AND PE[i]'s Load Average < LOW-THRESHOLD )
         Set PE[i]'s PAT to 0;
   Done = FALSE;
   while( !Done )
      Compute PAT for ALL ACTIVE Processors the CAT for their assigned Clusters;
      Sort Processor ACTIVE Statistics Based on PAT;
      For Each ACTIVE Processor, PE[i]
         Sort PE[i]'s Cluster Statistics Based on the Cluster's CAT;
      if( TryToOffLoad() == FAILED )
         Done = TRUE;
```

Figure 1: Dynamic Load Management Algorithm.

```
Function TryToOffLoad()
   /* NOTE: PE[0..N] is SORTED from lowest to highest PAT values */
   For All Clusters, c, assigned to PE with highest PAT, PE[highest]
      For ALL ACTIVE Processors, PE[i], from 0 to highest-1
         if( Moving Cluster, c, from PE[highest] to PE[i] reduces the
             difference in PAT values between PE[highest] and PE[i] AND
             the current PAT value difference < PAT_VALUE_THRESHOLD )
            Move Cluster c from PE[highest] to PE[i];
            return( SUCCEEDED );
   return( FAILED );
```

Figure 2: Algorithm for *TryToOffLoad*.

Conversely, processors removed from the usable set are assigned a PAT value of infinity. The algorithm repeatedly attempts to move cluster(s) from the processor containing the largest PAT value. Clusters on that processor are scanned in order of decreasing CAT value, in order to try to move large computation load clusters first to try to minimize the number of required migrations, as mentioned earlier. For each cluster, processors are scanned from low PAT values to high in order to locate a destination for the off-loaded workload. If moving the cluster will result in a reduction in the maximum difference between PAT values, the move is accepted, and the procedure is repeated. If no move causes a further reduction in the maximum difference in PAT values, the algorithm terminates and resumes when the next scheduling interval begins.

4 Implementation

An initial implementation of the background load management scheme has been developed. This implementation currently executes on networks of Silicon Graphics uniprocessor workstations. Extensions to multiprocessors and heterogeneous machines are under development. The Time Warp software executing on each processor was derived from a shared-memory multiprocessor-based implementation, and PVM is used for interprocessor communications, so it is anticipated that extension to multiprocessors and heterogeneous environments will be straightforward.

The same period ($T_{schedule}$) is used for both processor allocation and load redistribution decisions, and is a user defined parameter. The current implementation performs load management synchronously, i.e., a barrier is used to stop all processes. Once this happens, processor allocation and load redistribution decisions are made, migrations are performed, and then the simulation is allowed to resume execution.

A well known problem in migrating Time Warp LPs (and thus clusters of LPs) is the fact that each contains a large amount of state. Specifically, each LP maintains a history of state vectors, in case rollback is later required. While phases could be used to address this problem (see [15]), this necessitates implementation of a mechanism for rollbacks to span processor boundaries because a rollback may extend beyond the beginning point of a recently created phase. A simpler, though perhaps more radical, solution is used here. The entire simulation computation is rolled back at each scheduling point to GVT if any load redistribution is to be performed. This makes migration of a cluster of Time Warp LPs no more expensive than migrating non-optimistic computations because there is no need to migrate the history information. This approach also has the side effect of "cleaning up" overly optimistic computation. In this sense, this approach is not unlike the mechanism described in [12] which found such periodic, global rollbacks to be beneficial. Our initial experiments indicate that this mechanism provides a reasonably simple and efficient mechanism for reducing migration overhead.

To enable the shared memory Time Warp kernel to execute in a distributed environment and support dynamic load management several significant changes were made. First, an extra process is created on each workstation to manage all external communications. This process is called the *reflector thread*. Its tasks include the sending and receiving of all GVT, application defined and dynamic load management messages. Application messages or events are marshalled to the Time Warp kernel(s) that are executing on that local workstation via shared memory.

Another change was the piggy-backing of Mattern's GVT algorithm [13] on top of the existing shared memory GVT algorithm [7]. In this arrangement, Mattern's algorithm forces the shared memory algorithm to be executed on each workstation to determine its local virtual time. This information in-conjunction with a lower bound on all transit messages between consistent cuts is used to approximate GVT.

The last significant change was adding support for LP clustering and moving those clusters among the different workstations. To support clustering, a separate calendar queue [2] is used to store the pending set of events for each cluster of LPs. Before processing an event, the Time Warp kernel selects the cluster with the smallest time stamped event. Then that smallest time stamped event is dequeue from it's calendar queue and processed. The process of moving a cluster is accomplished by halting all Time Warp kernels from processing events and forcing all LPs in the system to roll back to GVT. The forced rollback is accomplished by placing a dummy event that has GVT as its time stamp in the message queue of every LP in the system. Once all rollback processing is complete, the moving of clusters between workstations is allowed to commence. Once cluster movement is accomplished the LP routing tables are updated to reflect the changes in LP and cluster location.

The dynamic load manager (DLM) is implemented as a separate stand alone program that currently executes on its own machine. It was designed this way to provide a clean separation between the load management policy and the mechanism needed to support it. Moreover, this design simplifies the implementation by not having to integrate this functionality into the existing Time Warp kernel and has the capacity to work with any Time Warp system that uses PVM for external communications.

Upon receiving an update message from each workstation, the DLM analyzes the load statistics according to the previous mentioned algorithm. If any moves are necessary, the DLM broadcasts the *halt* message to all the workstations. Upon receiving the "halt" message, each workstation rolls back all LPs to GVT. When the *master* Time Warp workstation determines that all *client* Time Warp workstations have completed the force rollback, it sends a *halt complete* message to the DLM. After receiving the *halt complete* message, the DLM issues *move* messages to the appropriate workstations, which they execute. When each move is complete the destination workstation sends a *move acknowledge* message to the DLM. After all *move acknowledge* message are received, the DLM broadcasts the *resume execution* message to all the workstations and event processing commences.

5 Experiments

Experiments were performed to evaluate the effectiveness of the load distribution system in (1) redistributing workload in the presence of external computations on a fixed set of host machines, and (2) being able to effectively redistribute the workload as the usable processor set changes. Controlled experiments were performed with artificial external application processes manually introduced before or during the execution of the Time Warp program.

The Time Warp application used in these experiments is a simulation of a wireless personal communication services (PCS) network [4]. A PCS network [5] provides wireless communication services for nomadic users. The service area of a PCS network is populated with a set of geographically distributed transmitters/receivers called *radio ports*. A set of radio channels are assigned to each radio port, and the users in the *coverage area* (or *cell* for the radio port) can send and receive phone calls by using these radio channels. When a user moves from one cell to another during a phone call a *hand-off* is said to occur. In this case the PCS network attempts to allocate a radio channel in the new cell to allow the phone call connection to continue. If all channels in the new cell are busy, then the phone call is forced to terminate. It is important to engineer the system so that the likelihood of force termination is very low (e.g., less than 1%).

For all experiments the *call-initiated* PCS model is used. This model is organized around two object types: *Cell* and *Call*. The Cell represents the area covered by a cellular receiver/transmitter that has a fixed number of channels allocated to it. The Call represents an active phone conversation. When a Call arrives at a Cell, channel availability is determined. If no channel is available then that Call is counted as being blocked. If a channel is available, it is assigned to that Call. When the Call completes, the occupied channel is released. If the portable to whom the Call is connected is leaving the Cell's area [1], then the allocated channel is released and the Call is handed-off to the destination Cell. The hand-off to the destination Cell is realized as a call arrival event at that neighboring Cell. If a channel at the neighboring Cell is not available, the Call is counted as a *hand-off block*, which is also known as a *forced termination*. If a channel is available, the Call reconnects and continues without interruption. In order to avoid sharing state between LPs, we realized Cells as LPs but Calls/Portables as timestamped messages that travel among the Cell LPs [4].

The PCS simulation was configured with 1024 Cells giving the simulation a total of 1024 LPs. These 1024 LPs are grouped into 64 clusters of 16 LPs each, and mapped to the processors such that the amount of remote communications is minimized. Each processor will initially have 8 clusters. The completion time for a phone call was exponentially distributed with a mean of 3.0. The number of channels per cell is fixed at 10. The number of portables per cell was fixed at 25. Each experiment processes approximately 24 million committed events.

The hardware platform includes 9 Silicon Graphics Indy Workstation interconnected

[1] We assume Portable movement is exponentially distributed.

Simulation	Execution Time (sec)	Efficiency	Avg. RB Length
PCS With DLM	711.89	91.96 %	19.20
PCS Without DLM	967.30	75.18 %	23.77

Table 1: Performance comparison PCS simulation with and without DLM.

through Ethernet. Here, one workstation executes the DLM process and the other eight execute the simulation. Each workstation has a single 150 MHZ, MIPS R4400 processor with 32 MB of RAM.

The following parameter settings were used to configure the DLM: the update interval was set to approximately 60 seconds. The threshold to allocate new processors was set to 1.5, and the threshold to vacate a processor was set at 3.0.

5.1 Static External Workloads

The first set of experiments assume a fixed external computation workload executes on the processors. In these experiments, five external processes are added each to four of the eight simulation processors. The external workload remains fixed throughout the execution of the simulation.

Table 1 compares the execution time, efficiency (fraction of executed events that are eventually committed), average rollback length, and amount of interprocessor communications with and without the load management scheme. In each case we use the *best* static mapping of clusters to PEs that minimize the amount of remote communications. Here it can be seen that the DLM improves the execution time of the simulation by 36%. We also observe higher efficiencies and shorter rollback distances for PCS with DLM than without.

In Figure 3 the number of clusters mapped to each processor is shown over the course of the simulation. It is observed that each processor is initially assigned 8 clusters. However, after the first set of load statistics is collected the DLM issues the movement of 2 cluster each from the four loaded processors (PEs 0, 1 , 2, 3) to the less loaded processors (PEs 4, 5, 6, 7). After the second update, PE 3 looses a cluster to PE 4. After that, no other moves are necessary because the Time Warp system has reached steady state with respect

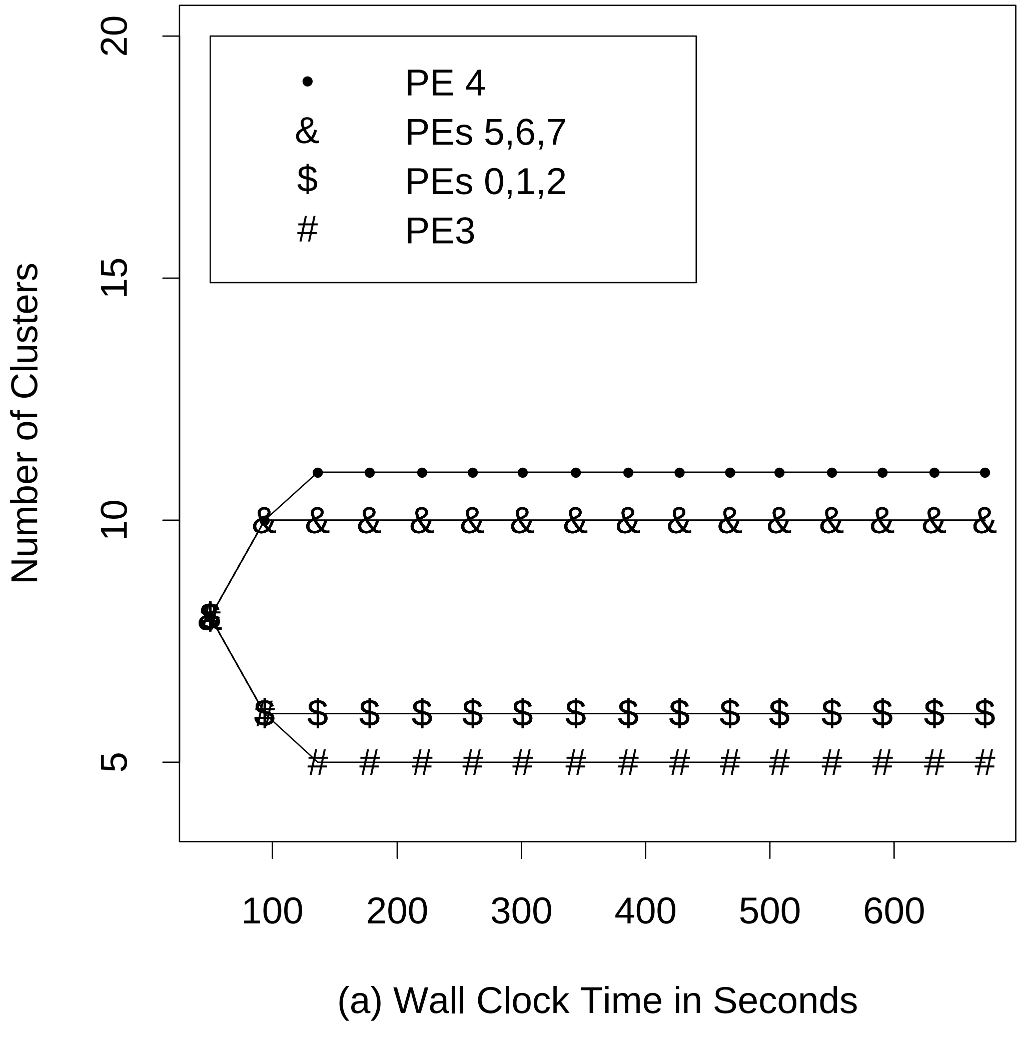

Figure 3: Static Workload: Number of Clusters mapped to each processor/workstation as a function of the wall clock time.

Simulation	Execution Time (sec)	Efficiency	Avg. RB Length
PCS With DLM	1503.44	84.45 %	15.20

Table 2: Performance of Threshold experiments where 4 of the 8 PEs are vacated.

to processor PAT values.

5.2 Changing the Usable Set of Processors

The next set of experiments were designed to evaluate the effectiveness of the algorithm in adapting to a different set of usable processors. Of particular interest is determining if the system can efficiently adapt to removing processors from the usable set due to excessive loads during the Time Warp execution. Experiments to increase the size of the usable set could not be performed in time for this paper submission, due to the contention for the computing facility, but are planned for the final version of this paper.

In these experiments, external jobs are added to four of the eight processor. This external workload is sufficient to cause the Time Warp simulation to vacate the four workstations.

Figure 2 shows execution time, efficiency, and average rollback length for the PCS simulation with dynamic load management when four of the eight processors are heavily loaded. We observe good simulation efficiencies in this experiment.

Figure 4 depicts the DLM making the decision to evacuate four of the eight processors. It is observed that the heavily loaded processors (PEs 1, 3, 5 and 7) have zero cluster assigned to them after the first update and the unloaded processors (PEs 0, 2, 4, 8) each have 16 cluster assigned them. After the initial move of 32 cluster, the Time Warp system reaches steady state with respect to PAT values and no other moves are issued by the DLM.

A trace of the number of rollbacks within each scheduling interval is shown in Figure 5. It is observed prior to migrating the clusters off the heavily loaded processors that the rollback frequency was quite high. However, immediately after the clusters are redistributed off the loaded processors, we see a sharp decline in the number of events that are rolled back. This figure suggests that the that the distributed simulation is able to detect the presence of the excessive external workload, and effectively redistribute its workload over the remaining four

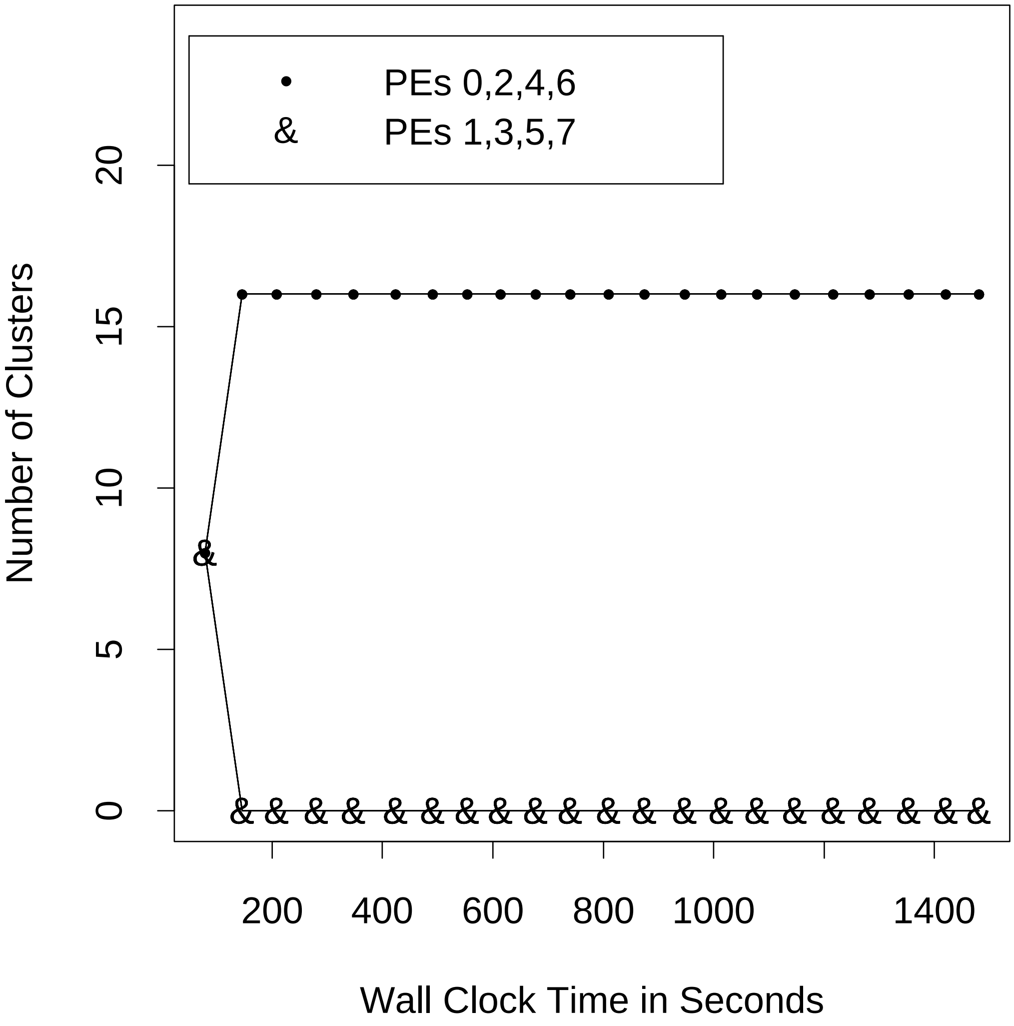

Figure 4: Threshold Workload: Number of clusters mapped to each processor/workstation as a function of the wall clock time.

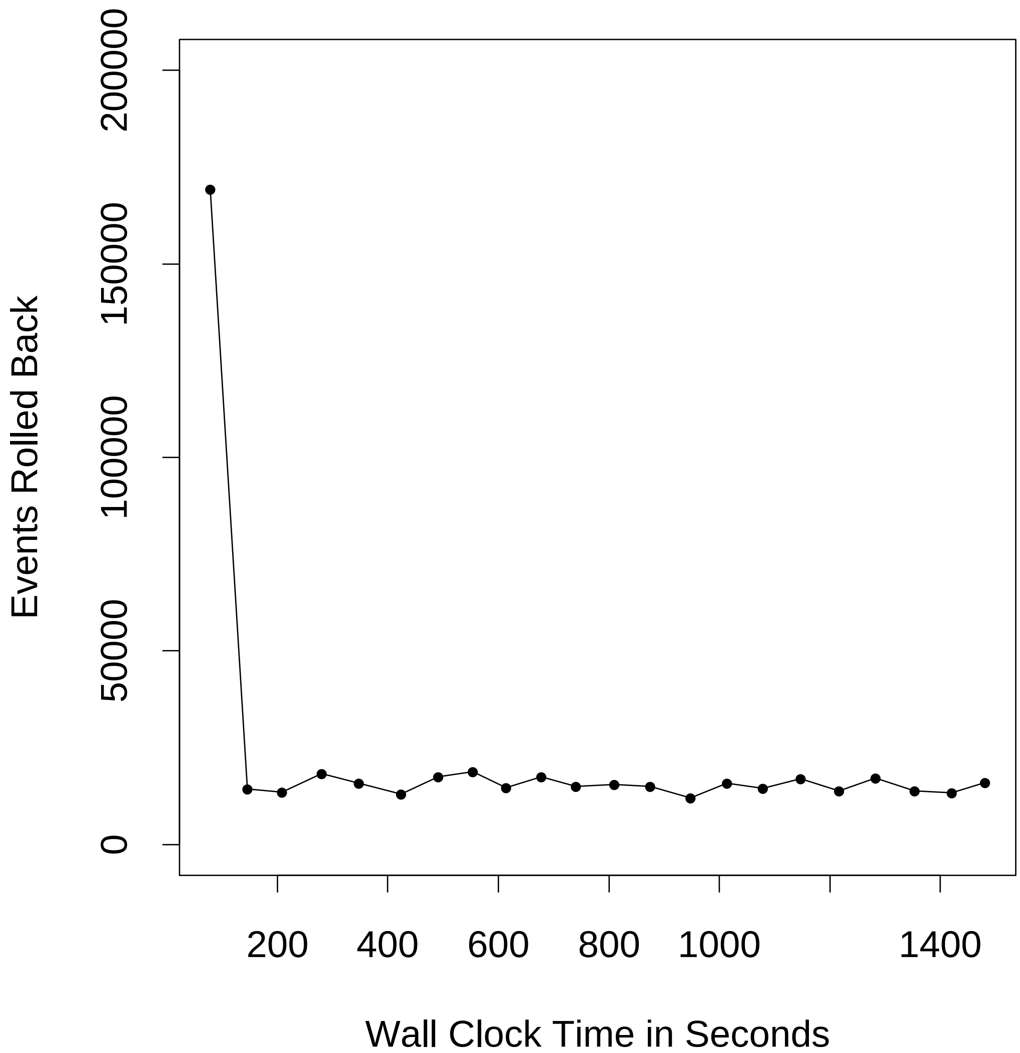

Figure 5: Threshold Workload: Number of events rolled back between DLM update intervals.

processors.

6 Conclusions and Future Work

A load distribution system enabling background execution of Time Warp programs on distributed computing platforms is described. An implementation of this system on a network of workstations was realized. Initial experiments indicate this approach shows promise in enabling efficient execution of Time Warp programs despite dynamically changing, externally induced workloads. Further experimentation using this system is in progress, and additional measurements are expected in the final version of this paper.

Work in refining and optimizing the performance of the load distribution system continues. Possible extensions include distribution of the load distribution software over multiple hosts, and asynchronous execution on the network of workstations. Another question is determination of appropriate parameter settings to maximize performance as well refining the PAT metric to include cluster-cluster affinity (to minimize interprocessor traffic) and the processor's available memory resources.

Work is proceeding to extend the current implementation of the load distribution system to networks of heterogeneous workstations, including multiprocessor hosts. The distributed simulator already executes on Sun workstations, and multiprocessor Sparc machines, and an implementation on multiprocessor SGI machines is planned. The cluster migration mechanism is built using PVM, migration of clusters among heterogeneous host machines is nearly completed.

References

[1] J. Briner, Jr. Fast parallel simulation of digital systems. In *Advances in Parallel and Distributed Simulation*, volume 23, pages 71–77. SCS Simulation Series, January 1991.

[2] R. Brown. Calendar queues: A fast 0(1) priority queue implementation for the simulation event set problem. *Communications of the ACM*, 31(10):1220–1227, October 1988.

[3] C. Burdorf and J. Marti. Load Balancing Strategies for Time Warp on Multi-User Workstations. *The Computer Journal*, 36(2):168–176, 1993.

[4] Reference Deleted.

[5] Cox, D. C. Personal Communications – A Viewpoint. *IEEE Communications Magazine*, 128(11), 1990.

[6] Reference Deleted.

[7] Reference Deleted.

[8] D. W. Glazer and C. Tropper. On process migration and load balancing in Time Warp. *IEEE Transactions on Parallel and Distributed Systems*, 3(4):318–327, March 1993.

[9] A. Goldberg. Virtual time synchronization of replicated processes. In 6^{th} *Workshop on Parallel and Distributed Simulation*, volume 24, pages 107–116. SCS Simulation Series, Jan. 1992.

[10] D. R. Jefferson. Virtual time. *ACM Transactions on Programming Languages and Systems*, 7(3):404–425, July 1985.

[11] M. Litzkow and M. Livny. Experience with the condor distributed batch system. In *IEEE Workshop on Experimental Distributed Systems*, October 1990.

[12] V. K. Madisetti, D. A. Hardaker, and R. M. Fujimoto. The mimdix operating system for parallel simulation and supercomputing. *Journal of Parallel and Distributed Computing*, 18(4):473–483, August 1993.

[13] F. Mattern. Efficient distributed snapshots and global virtual time algorithms for non-fifo systems. *Journal of Parallel and Distributed Computing*, 18(4):423–434, August 1993.

[14] M. Presley, M. Ebling, F. Wieland, and D. R. Jefferson. Benchmarking the Time Warp Operating System with a computer network simulation. In *Proceedings of the*

SCS Multiconference on Distributed Simulation, volume 21, pages 8–13. SCS Simulation Series, March 1989.

[15] P. L. Reiher and D. Jefferson. Dynamic load management in the Time Warp Operating System. *Transactions of the Society for Computer Simulation*, 7(2):91–120, June 1990.

[16] R. SchlagenHaft, M. Ruhwandl, C. Sporrer, and H. Bauer. Dynamic Load Balancing of a Multi-Cluster Simulation on a Network of Workstations. In 9^{th} *Workshop on Parallel and Distributed Simulation*, volume 24, pages 175–180. IEEE, July 1995.

[17] F. Wieland, L. Hawley, A. Feinberg, M. DiLorento, L. Blume, P. Reiher, B. Beckman, P. Hontalas, S. Bellenot, and D. R. Jefferson. Distributed combat simulation and Time Warp: The model and its performance. In *Proceedings of the SCS Multiconference on Distributed Simulation*, volume 21, pages 14–20. SCS Simulation Series, March 1989.