Efficient Execution of Time Warp Programs on Heterogeneous, NOW Platforms

Christopher D. Carothers and Richard M. Fujimoto, *Member*, *IEEE Computer Society*

Abstract—Time Warp is an optimistic protocol for synchronizing parallel discrete event simulations. To achieve performance in a multiuser network of workstation (NOW) environment, Time Warp must continue to operate efficiently in the presence of external workloads caused by other users, processor heterogeneity, and irregular internal workloads caused by the simulation model. However, these performance problems can cause a Time Warp program to become grossly unbalanced, resulting in slower execution. The key observation asserted in this article is that each of these performance problems, while different in source, has a similar manifestation. For a Time Warp program to be balanced, the amount of wall clock time necessary to advance an LP one unit of simulation time should be about the same for all LPs. Using this observation, we devise a single algorithm that mitigates these performance problems and enables the "background" execution of Time Warp programs on heterogeneous distributed computing platforms in the presence of external as well as irregular internal workloads.

Index Terms—Discrete event simulation, distributed simulation, network of workstations (NOW), time warp, dynamic load balancing.

1 Introduction

Time Warp is an optimistic synchronization mechanism develop by Jefferson and Sowizral [28] used in the parallelization of discrete event simulation. The distributed simulator consists of a collection of *logical processes* or LPs, each modeling a distinct component of the system being modeled, e.g., a server in a queuing network. LPs communicate by exchanging timestamped event messages, e.g., denoting the arrival of a new job at that server.

The Time Warp mechanism uses a detection-and-recovery protocol to synchronize the computation. Any time an LP determines that it has processed events out of timestamp order, it "rolls back" those events, and reexecutes them. For a detailed discussion of Time Warp, as well as other parallel simulation protocols, we refer the reader to [19].

With few exceptions, most research on Time Warp to date assumes the simulation program has 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. These extreme measures are taken because 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 simulated future, resulting in very long or frequent rollbacks. While good from an experimental standpoint, this "dedicated platform" paradigm is often not the prevalent paradigm one encounters in practice. In particular, networks of desktop workstations (NOWs) 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.

Moreover, this multiuser computing environment is typically composed of heterogeneous workstations that contain different processors. These processors may have quite different performance characteristics that, if not taken into consideration, can lead to poor Time Warp performance.

In addition to performance-robbing external workloads and processor heterogeneity, the application model itself can be a source of perturbation. In many simulation models, the amount of CPU time required to process an event varies among logical processes (LPs), and the event population may differ across LPs, both of which can degrade performance.

The key observation asserted in this article is that each of these performance problems, while different in source, has a similar manifestation. For a Time Warp program to be balanced, the amount of wall clock time (i.e., elapsed real-time) necessary to advance an LP one unit of simulation time should be about the same for all LPs. Using this

- *C.D. Carothers is with the Department of Computer Science, Rensselaer Polytechnic Institute, 110 8th St., Troy, NY 12180-3590. E-mail: chrisc@cs.rpi.edu.*
- *R.M. Fujimoto is with the College of Computing, Georgia Institute of Technology, Atlanta, GA 30332-0280. E-mail: fujimoto@cc.gatech.edu.*

Manuscript received 18 Nov. 1997; revised 12 Aug. 1998; accepted 3 Apr. 1999.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number 105944.

observation, we devise a single algorithm that mitigates these performance problems and enables the "background" execution of Time Warp programs on heterogeneous distributed computing platforms in the presence of external as well as irregular internal workloads.

To demonstrate the effectiveness of our *Background Execution (BGE)* algorithm, we construct an experimental testbed. In particular, we develop a model of a wireless (PCS) communications systems that can be executed atop our Georgia Tech Time Warp (GTW) system [22]. What is special about this application is that it is an instance of a class of applications that are *self-initiated*[34]. Here LPs typically schedule most of their events to themselves, which leads to relatively few remote messages making this class of applications well suited for the NOW platforms, and which are known to have high remote communication overheads.

Using this simulation model in addition to a synthetic benchmark application, we demonstrate our Background Execution (BGE) algorithm is able to: 1) 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, 2) 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 3) dynamically redistribute the workload on the existing set of processors as some become more heavily or lightly loaded by changing externally or internally induced workloads.

The remainder of this study is organized as follows: Section 2 characterizes the different kinds of workload imbalances to which Time Warp programs can be subjected. Section 3 presents related work. Section 4 presents our BGE algorithm that is suitable for balancing the load in the presence of irregular internal workloads and external workloads. We then describe the implementation of our BGE algorithm in Section 5. The benchmark applications used in this experimental study are discussed in Section 6. Results from an experimental performance study are then presented where we compare the performance of our GTW system with and without our BGE algorithm in Sections 7, 8, 9, and 10. Section 11 summarizes our results and presents future research directions.

2 Characterization of Time Warp Workload Imbalance

For Time Warp programs, there are three sources of internal workload imbalance: 1) *event population*, 2) *event granularity*, and 3) *communications*. With irregular event population workloads, the number of events processed over a period of simulated time may differ among LPs. Consequently, some LPs induce a much greater "load" on the processor than others because they have to process more events to reach the same point in simulated time. This can have a detrimental impact on Time Warp performance, resulting in the underloaded processors becoming "overly optimistic" and being rolled back by the overloaded processors. Applications that suffer from this behavior include digital logic circuit simulations [8], [44] National Airspace System models [51], [53] and PCS models where the portable population differs among calling areas.

A special case of an unbalanced event population workload is an unbalanced *LP population* where the number of LPs per processor differs significantly. We make LP population a special case because the driving force behind the simulation is the event population. That is to say, if we add LPs to a simulation where all LPs are the same without adding any additional events, the forward execution costs will remain approximately the same.

Load imbalances caused by irregular event granularity workloads occur when the amount of wall clock time to process an event varies greatly among LPs. We assume that state saving overheads are part of event processing costs. Thus, the event population for each LP could be the same, but the amount of time a processor takes to process the events varies. As with event population workloads, these kinds of irregular workloads also cause the overloaded processors to constantly roll back the underloaded ones, leading to poor Time Warp performance. Applications that exhibit this kind of internal workload include network simulations, such as SS7 [54], ATM [26], and battle manager simulations, such as TISES [48]. PCS models, such as those presented in [12], can also be configured to generate event granularity workload imbalances.

The last cause of internal workload imbalance is communication. Here, an LP or group of LPs change their communication pattern so that new off processor communications are introduced that did not exist previously. The consequence of this is that some set of processors are now using more CPU cycles to send and receive remote messages, which can be quite costly in a NOW environment. The effect on Time Warp performance is that the communication-laden processors act as if they are loaded and slow their rate of advance through simulated time. Meanwhile, the other processors advancing at a much faster rate will be rolled back, thus reducing system performance. These communications workloads typically happen in applications where the "action" is subject to radical changes, such as the Eagle combat model [38]. Due to implementation-specific limitations, our BGE algorithm is currently unable to mitigate this type of workload imbalance. We do however suggest some solutions in Section 11.

In addition to irregular internal workloads, Time Warp programs executing in a multiuser NOW environment can be subjected to two sources of external workload imbalance. The first source is a *user-induced* external workload, which occurs when a user or group of users executes a program either locally or remotely on the same pool of computing resources that is currently being used by the Time Warp program. These external workloads effectively "steal" CPU cycles from the Time Warp program, assuming the operating system is "fair" in its allocation of CPU resources, causing some processors to become "overloaded" with work. This can seriously degrade the performance of the Time Warp program. Here LPs mapped to the "overloaded" processors will require more wall clock time to advance through simulated time, allowing "underloaded" processors to advance at a much faster rate. These faster

processors become "overly optimistic" and are consistently rolled back as the slow processors send them straggler events that arrive in an LPs past, resulting in a "thrashing" rollback behavior and degraded Time Warp performance.

Included in user-induced workloads are *system-induced* workloads. These are workloads generated by the network operating system, such as automated backup and software distribution programs, such as `depot`.

The last source of external workload imbalance is *processor heterogeneity*, where processor performance characteristics vary among workstations in a NOW. From a Time Warp programs point of view, workstations containing older, slower processors appear as though they are overburdened with work when compared with more advanced workstations. As with user-induced workloads, heterogeneous processors create a situation where LPs mapped to slow processors progress at a rate less than that of LPs mapped to faster processors, resulting in long rollbacks because the faster processors become "overly optimistic," greatly reducing the performance gains of the distributed simulation.

With the exception of the communication workload imbalance, we will demonstrate how our BGE algorithm is able to detect and mitigate each of these workload imbalances.

3 Related Work

Background execution is essentially a load management problem. Traditionally, load balancing or load sharing involves distributing the workload across a fixed set of processors in order to minimize the elapsed time to execute the program. Many load balancing techniques designed to support distributed systems (e.g., NOWs) have been proposed and implemented, such as [52]. However, when executing Time Warp programs on a NOW, the traditional load balancing problem must be extended in three ways which necessitates the need for a new algorithm. 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. Finally, the set of processors may have different performance characteristics that must be taken into consideration.

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 load management techniques specific to Time Warp. Thus, process/object migration systems such as Accent [55], Amoeba [33], Charlotte [1], Condor [29], Locus [50], MOSIX [3], MpPVM [13], RHODOS [56], Sprite [35], and V-System [49] that distribute jobs onto networked workstations as independent processes (i.e., not parallel program processes) to "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 [37]. Based on this metric, they propose a strategy that migrates processes from processors with highly 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. Glazer and Tropper propose allocating virtual time slices to processes, based on their observed rate of advancing the local simulation clock [24]. 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 load distribution that replicates bottleneck processes to enable concurrent execution [25]. Time Warp is used to maintain consistency among the replicated copies.

More recently, Wilson and Nicol [53] devise a method for automated load balancing in the SPEEDES parallel simulation environment [46]. The scheme they propose collects computation data, which consist of the number of events processed by each LP. These data are saved in a file and used during subsequent runs to statically partition the simulation application onto the set of available processors. Using this approach, the performance of the current run is only improved based on data collected from previous runs of the simulation.

Additionally, Avril and Tropper [2] present a scheme for dynamically load balancing Time Warp programs with irregular, internal workloads. Here *processor load* is defined to be the number of events which were processed by the LPs assigned to that processor, including events rolled back and reprocessed. Using this scheme, they improve Time Warp's throughput by 40 to 100 percent for VLSI models from the ISCS'89 benchmarks, where throughput is defined to be the number of non-rolled-back events per unit of time. A fundamental difference between this approach and ours is the explicit exclusion of virtual time in the load balancing metric. The consequence of this decision is that it implicitly assumes that all events span the same amount of virtual time. Moreover, by using event counts in the processor load metric, it assumes that all events have the same computational requirements. While these two assumptions are true for VLSI simulation models, it is not true of all simulation models, such as TISES [48]. Consequently, these assumptions limit the utility of their approach.

None of these approaches address the question of balancing the load in the presence of external workloads and processor heterogeneity. The approach proposed here 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 [10] 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. A drawback with this approach is that it depends on virtual time differences among LPs to detect load balances. *Message-initiated* applications [34] can exhibit behaviors which allow underloaded processors to advance their LPs only to be rolled back later due to late arriving messages caused by slow processors. Thus, when the load distribution algorithm takes a snapshot of where LPs are with respect to virtual time, it could be that all LPs are at the same point, despite the presence of a load imbalance. Moreover, throttling techniques, such as RiskFree TWOS [4], SRADS [17], Adaptive Flow Control [36], Elastic Time Algorithm [45] and Breathing Time Warp [47] limit the advance of processors such that LPs are not allowed to become "overly optimistic" and may all at the same point in virtual time, despite an obvious workload imbalance. Consequently, under certain application workloads or when this approach is combined with other risk-limiting, throttling mechanisms, load imbalances may go undetected.

Schlagenhaft et al. [41] 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 it nor Burdorf's approach address the question of dynamically changing the set of processors utilized by the simulation, processor heterogeneity, or dynamically changing internal workloads.

Work in dynamic load balancing for conservative parallel simulations has been done, as well. Most recently, Boukerche and Das [7] devise a novel load balancing scheme for an optimized version of the Chandy-Misra null message algorithm [14]. Their approach introduces the notion of CPU-queue length as a measure of workload on each processor. This workload measure is determined for both real messages and null messages on each processors and combined using a weighted average function to compute the overall workload on a processor. Using this approach, they reduce synchronization overheads in the Chandy-Misra algorithm by 30 to 40 percent when compared to the use of a static load balancing algorithm. While good results are achieved with this approach for conservative simulation protocols, it is not appropriate for optimistic protocols, because an unknown amount of work that is currently pending on a processor *could be* later undone. Optimistic protocols, such as Time Warp, require a measure of workload that only considers the amount of *committed* computation.

4 The BGE 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 burdens 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 single 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 overheads, this approach will keep LPs that frequently communicate together, on the same processor, provided they are grouped within the same cluster. We assume the modeler has enough knowledge of the application to do a good job of "clustering" the LPs together. Once an LP is assigned to a cluster, it remains so for the lifetime of the simulation.

In our BGE algorithm, a central process is responsible for monitoring the processors that are to be used by the distributed simulation. This process executes periodically at a user-defined *scheduling interval*, denoted by $T_{schedule}$ and 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. The load balancing policy is responsible for assigning LP clusters to processors. Our 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 or processing element (PE).

4.1 Processor Allocation

Initially, all processors are in the usable set. However, as shown in Fig. 1, should a processor lose all of its clusters as a result of normal load balancing, the BGE algorithm

```
AnalyzeStatistics()
  for all INACTIVE processors, PE[i]
    if( PE[i].Load < PE[i].DeAllocLoad / 2 )
      PE[i].Status = ACTIVE;
      PE[i].PAT = 0;
    end if;
  end for;
  done = FALSE;
  while( !done )
    compute PAT for all ACTIVE processors by
      summing the CAT values for each
      processor s assigned clusters;
    sort processor statistics based on PAT;
    for all ACTIVE processors, PE[i]
      sort PE[i] s cluster statistics based
        on the cluster s communications
        affinity to PE[high];
    end for;
    if( TryToOffLoad() == FAILED )
      done = TRUE;
    end if;
    if( PE[high].NumClusters == 0 )
      PE[high].DeAllocLoad = PE[high].Load;
      PE[high].Status = INACTIVE;
    end if;
  end while;
end AnalyzeStatistics;
```

Fig. 1. Background Execution Algorithm.

records the current load and removes that processor from the usable set. We call this load the $DeAllocLoad$. This processor remains unusable until the current load over a $T_{schedule}$ interval falls below $DeAllocLoad/2$.

When a processor is added to the usable set, its status is set to ACTIVE and its *Processor Advance Time (PAT)* value is set to zero, making it a prime candidate for accepting new clusters. Conversely, processors removed from the usable set are marked as being INACTIVE. PAT is the metric used to trigger load migrations and will be discussed in greater detail in the next section.

4.2 Load Balancing Policy

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 the internal (Time Warp) and external workloads assigned to each processor, as well as differences in processor speed (processor heterogeneity). 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. Since we are only interested in obtaining a measure of "useful" work, the BGE algorithm only considers committed or "useful" computation, and rolled back computation is not allowed to be treated as additional computation load. Our reason for excluding "nonuseful" work in our metric is because these computations are application dependent and not always caused by workload imbalances (internal or external). In fact, even if the load is balanced, Time Warp systems can still experience a cascade of rollbacks caused by the application [30]. These rollbacks create an unstable situation where rollbacks become geometrically longer and ultimately result in significantly longer execution times than the sequential simulation. Consequently, if these "nonuseful" computations are included in the metric they might fool the system into making a wrong load balancing decision.

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 pair of processors. PAT values are easily measured for the current mapping of clusters to processors. However, 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.

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.
2. $TWFrac_i$ is the fraction of total CPU cycles that a Time Warp program on processor i was allocated over the last $T_{schedule}$ interval. This measure is used to account for a processor's user-induced, external workload.
3. PAT_i is defined as the sum of all $CAT_{c,i}/TWFrac_i$ values of clusters mapped, or hypothesized to be mapped, to processor i.

Operationally, $CAT_{c,i}$ is calculated as follows:

$$CAT_{c,i} = CPU_{c,i}/\Delta_{GVT} \tag{1}$$

where $CPU_{c,i}$ is the amount CPU time used to process *committed* events by cluster c on processor i over the last $T_{schedule}$ interval, and Δ_{GVT} is the change in GVT over the last $T_{schedule}$ interval. By dividing that result by Δ_{GVT}, we obtain the amount of computation time required to advance cluster c one unit of simulation time. By combining the above definitions, we obtain the following:

$$PAT_i = \sum_{c=0}^{C_i-1} CAT_{c,i}/TWFrac_i \tag{2}$$

$$= \sum_{c=0}^{C_i-1} CPU_{c,i}/(TWFrac_i\Delta_{GVT}) \tag{3}$$

$$= CPU_i/(TWFrac_i\Delta_{GVT}), \tag{4}$$

where $CPU_i = \sum_{c=0}^{C_i-1} CPU_{c,i}$.

We derive (4) by substituting $CAT_{c,i}$ in (1). By performing dimensional analysis on (4), we observe that PAT_i

```
TryToOffLoad()
  highest = NumActiveProcessors-1;
  for all clusters, c, assigned to PE[highest]
    for all ACTIVE processors, PE[i], 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]
          && the current PAT value difference >
          ΘPAT_max)
          MoveCluster( highest, c, i );
          return( SUCCEEDED );
      end if;
    end for;
  end for;
  return( FAILED );
end TryToOffLoad;
```

Fig. 2. TryToOffLoad function. $PE[0..N-1]$ is sorted from lowest to highest PAT value. $PE[NumActiveProcessors-1]$ is the active processor with the highest PAT value. Processors $PE[NumActiveProcessor..N-1]$ are inactive and have a PAT value set of infinity.

does in fact represent the amount of wall clock time needed to advance processor i one unit of simulation time. We observe that based on its definition,

$$TWFrac_i = TotalCPU_i / T_{schedule},$$

where $TotalCPU_i$ is the amount of total amount of user CPU time given to processor i over the last $T_{schedule}$ interval. Note that $TotalCPU_i$ is equal to CPU_i plus the amount of time spent doing "nonuseful" work. Substituting these equations into (4) yields:

$$PAT_i = (CPU_i T_{schedule}) / (TotalCPU_i \Delta_{GVT}) \quad (5)$$

$$= (T_{schedule}) / \Delta_{GVT} \quad (6)$$

which is wall clock time, represented by $T_{schedule}$ per unit of simulation time, represented by Δ_{GVT}. We allow $TotalCPU_i$ to cancel CPU_i, since they both are a measure of CPU cycles consumed.

If cluster c is moved from processor i to processor j, then PAT_i is reduced by the amount $CAT_{c,i}$ and PAT_j is increased by $CAT_{c,j}$. 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.

The load balancing algorithm attempts to minimize the maximum of $(PAT_i - PAT_j)$ over all i and j, shown in Fig. 1. The algorithm repeatedly attempts to move cluster(s) from the processor containing the largest PAT value. Clusters on the donating processor are scanned in order of highest communication affinity to lowest affinity to the receiving (low PAT) processor (see Fig. 2). Cluster-processor communication affinity is determined based on message counts over the last $T_{schedule}$ interval. This is done to lessen the potential for new remote communications to be introduced into the distributed simulation computation. For each cluster, processors are scanned from low PAT values to high in order to locate a destination for the

```
MoveCluster( srcpe, c, destpe )
  copy PE[srcpe] cluster statistics For c to
    PE[destpe];
  CPU_destpe,c = CPU_destpe,c(σ_srcpe/σ_destpe);
  add this move to movelist;
end MoveCluster;
```

Fig. 3. MoveCluster function.

offloaded workload. If moving the cluster will result in a reduction in the difference between PAT values, the move is accepted, and the procedure is repeated. If subsequent moves fail to reduce the difference in PAT values, the algorithm terminates and resumes when the next $T_{schedule}$ interval begins.

Because statistics are collected for each processor individually regarding cluster CPU utilization and processor load, no modifications are required of the initial PAT value comparison in order to account for processor heterogeneity. However, we observe that once a cluster c migrates from some processor i to another processor j, we must modify the CPU time consumed by cluster c on processor i to reflect the difference in the relative speeds of processors i and j. To accomplish this we introduce a new parameter, σ_i, which denotes the relative speed of processor i, and is set by the user. This parameter is used in the MoveCluster function, shown in Fig. 3. Here, the migrating cluster's CPU time, $CPU_{destpe,c}$ is increased or decreased by the ratio of the source processor's σ to the destination processor's σ. Consequently, if the source processor is twice as fast as the destination processor, then the migrating cluster's CPU time will be doubled. Likewise, if the destination processor is twice as fast as the source processor, then the migrating cluster's CPU time will be reduced by half.

Load migration is only performed if the maximum difference in PAT values between any pair of processors exceeds ΘPAT_{max}, where Θ is a user-defined percentage between zero and one. This avoids performing migrations when the benefit that can be realized by the migration is modest.

5 Implementation

Our BGE algorithm is implemented as part of the Georgia Tech Time Warp (GTW) system, which is a parallel discrete event simulation executive based on Jefferson's Time Warp mechanism [27]. Currently, it runs on shared-memory machines as well as distributed memory platforms. The initial implementation was developed on a BBN Butterfly, GP-1000 [18]. From there, it has been rehosted to the KSR, the SGI Power Challenge, and the Sun Solaris multiprocessor platforms. A detailed description of all of GTW's optimizations can be found in [16].

To enable the shared-memory GTW kernel to execute in a distributed environment and support dynamic load management, several significant changes were made. First, a *reflector-thread* is created on each workstation to manage all external communications. 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. To mitigate any OS overheads, the reflector-thread is user-level and periodically polled by the GTW kernel. PVM [23] is used for remote or off-processor communications.

Another change was the piggy-backing of Mattern's GVT algorithm [32] on top of the existing shared-memory GVT algorithm [21]. 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 to the shared memory GTW executive was adding support for moving LP clusters among the different workstations. 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 [37]), this requires implementing 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 to rollback the entire simulation computation to GVT if any load redistribution is to be performed. This makes migration of Time Warp LPs no more expensive than migrating nonoptimistic computations because there is no need to migrate the history information. This approach also has the side effect of "cleaning up" overly optimistic computations. In this sense, this approach is not unlike the mechanism described in [31], which found such periodic, global rollbacks to be beneficial. Our experiments indicate that this mechanism provides a reasonably simple and efficient mechanism for reducing migration overhead.

We define $T_{schedule}$ as the interval of time used for determining load redistribution decisions, and is a user-defined parameter given in seconds. This implementation performs load management synchronously, i.e., a barrier is used to stop all processes once the workload policy program has determined that LP cluster migrations are necessary. After the distributed simulation is halted, workload policy migrations are performed, and then the simulation is allowed to resume execution.

To calculate $CAT_{c,i}$, $CPU_{c,i}$, as shown in (1), must be determined for all clusters in the system. To obtain this value, we employ the use of monotonically increasing hardware timers[1] and measure the computation time used to process each event. These timers where chosen because microsecond resolution was needed to accurately measure these low granularity computations (i.e., computations that only require tens of microseconds to execute). Other Unix timers, such as `gettimeofday` and `getrusage` only provide millisecond resolution on platforms such as the SGI, which is insufficient for our needs.

Using the fast hardware timers, $CPU_{c,i}$ is a running sum over the $T_{schedule}$ interval that includes the time to 1) enqueue each event into the pending set of events, 2) dequeue each event from a cluster's calendar queue, 3) state saving overheads prior to event processing, and 4) event processing time. Recall that $CPU_{c,i}$ only includes timing information from committed events during the $T_{schedule}$ interval.

Next, to calculate $TWFrac$, which represents the allocated CPU time as a percentage of elapsed wall clock time, denoted by $T_{schedule}$, the `getrusage` system call is used. Since a typically $T_{schedule}$ interval ranges between 5 and 50 seconds, the `getrusage` system call provides the required timer resolution in this case. This system call returns information describing the resources utilized by the current process, or all its terminated child processes, including such statistics as CPU time spent in user space, CPU time spent in the operating system, page faults, and swaps. By dividing the user CPU time over the last $T_{schedule}$ interval by $T_{schedule}$, $TWFrac$ is obtained.

The BGE manager (BM) 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 GTW executive.

Currently, this version of GTW executes on networks of Silicon Graphics and Sun Solaris uniprocessor workstations and multiprocessor servers.

6 Benchmark Applications

For the experiments presented in this study, we used the following two benchmark applications.

6.1 PHold Synthetic Workload

PHold is a simulation using a synthetic workload model [20]. The simulation consists of a fixed message population that moves among the LPs making up the simulation. The processing of a message consists of computing for a certain amount of time and then sending one new message to another LP with a certain timestamp increment. The distribution of the computation time per event, the timestamp increment, and the LP to which the message will be forwarded are parameters of the synthetic workload.

6.2 PCS

A PCS network [15] 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

1. On the Sun/Solaris machines, the `gethrtime` system call is used and provides microsecond timing resolution. On the SGI/IRIX machines, the hardware timer is memory mapped into GTW's address space and provides half-microsecond resolution.

TABLE 1
GTW Performance Comparison Using Balanced *Null* Event Granularity PHold with and without BGE Monitoring

$ExecTime_{GTW}$	$ExecTime_{GTW-BGE}$	% Difference
1118	1207 ($T_{schedule} = 10$)	+7.96 %
	1188 ($T_{schedule} = 20$)	+6.26 %
	1187 ($T_{schedule} = 30$)	+6.17 %
	1206 ($T_{schedule} = 40$)	+7.87 %
	1184 ($T_{schedule} = 50$)	+5.90 %
	1155 ($T_{schedule} = 60$)	+3.31 %

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 percent). For a detailed explanation of the PCS model, we refer the reader to [11].

7 Internal Workload Experiments

In this section, the results from our experimental study are presented. For all experimental data presented in this section, we use eight 167 MHZ Sun Sparc Ultra-1 workstations running version 2.5 of the Solaris operating system.

7.1 PHold Configuration

We configure the PHold program to have a one of two computation granularities: *null* and *one millisecond*. In the *null* case, event processing is made as small as possible. It consists of scheduling a single event into the future at a time $t + 1.0$, where t is the timestamp of the event currently being processed. In the *one millisecond* case, a one millisecond delay loop is added to the processing overheads of the *null* event. Like the *null* case, a single event is scheduled into the future at a time $t + 1.0$.

Each LPs initial message population is 25. Each of these events is assigned a timestamp that is exponentially distributed between 0 and 1. The number of LPs is fixed at 2,048, making the total message population 51,200. These LPs are grouped into 128 clusters, 16 LPs each. These 128 clusters are evenly distributed onto the eight processors, giving each processor 16 clusters or 256 LPs. The maximum number of clusters a processor can support is 32.

PHold is additionally configured to be self-initiated. When an event is processed where the source LP is different from the destination LP, the destination LP will schedule the next d generations of the event to itself. By d generations, we mean that the child of the event, and the child's child, and so on up to d times will be scheduled for the same LP. After d generations of the event have been produced, the destination LP is randomly picked. For the experiments discussed here, d is a number initially generated for each event based on a uniform distribution between 0 and 2,000.

For the static workload experiments, a one millisecond delay loop was added to the event computation whenever any LP mapped to cluster 0 processes an event. All other LPs process null events. Cluster 0 was mapped to PE 0. This type of workload is an unbalanced *event granularity* workload.

For the time-varying workload experiments, a one millisecond delay loop was added to the event computation whenever any LP mapped to cluster zero processed an event for the first-third of the simulation, then to the second-third of the simulation this delay loop was shifted to LPs in cluster 31 on PE 1. Finally, in the last third of the simulation, the delay loop is shifted to LPs in cluster 63 on PE 3.

7.2 PCS Configuration

PCS is arranged with 2048 cells. Each cell is configured with 25 to 75 portables per cell, yielding an initial message-population from 51,200 to 153,600 respectively. The number of channels per cell is 10. Call holding times are exponentially distributed with a mean of three minutes. Call mobility rate is exponentially distributed with a mean of 1 every 75 minutes. Call interarrival times per portable are exponentially distributed with a mean of 10 minutes. LPs are divided into 128 clusters, 16 LPs to a cluster, and 16 clusters to a processor.

In PCS, an irregular *event population* internal workload is introduced by giving each Cell (LP) mapped to cluster 0, an initial portable-population (i.e., event population) of 1,000.

7.3 Rules for Setting Θ and $T_{schedule}$

Recall, Θ determines how sensitive the BGE algorithm is to workload imbalances and $T_{schedule}$ In determining the best values for Θ and $T_{schedule}$, several factors were considered. First, we were concerned that the monitoring of GTW might significantly degrade GTW performance for small values of $T_{schedule}$, which determines how frequently GTW performance data are collected. For each cluster the following information is collected: 1) CPU time, 2) number of events processed, 3) number of events rolled back, 4) number of events events aborted,[2] and 5) a cluster communications matrix. The cluster communications matrix for a cluster denotes the destination cluster for every message sent by this cluster over the last $T_{schedule}$ interval.

To quantify this perturbation, we compared the execution times of *null* event granularity PHold under balanced internal workload conditions with and without BGE monitoring. Θ was set to 1.0 to eliminate all migrations. The results of this comparison are shown in Table 1. We observed that for event small values of $T_{schedule}$ (i.e., 10 and 20 seconds), the perturbation was less than 8 percent. Consequently, we believe that monitoring overheads are not a significant factor in determining the value for $T_{schedule}$ or Θ.

The next factor to be considered is migration costs, which includes the amount of time required to halt and roll the simulation back to GVT, known as *halt time*, as well as the time to effect the prescribed LP cluster migrations, called *move time*. We observed move times are affected by the amount of data contained within a cluster as well as how many clusters are moved. Halt times are affected by the degree to which GTW is "out-of-balance" since the more "out-of-balance" GTW is the further some processors must be rolled back during the halt phase.

2. As a flow-control mechanism, GTW will "abort" an event if, during the processing of that event, no memory is available to schedule a future event.

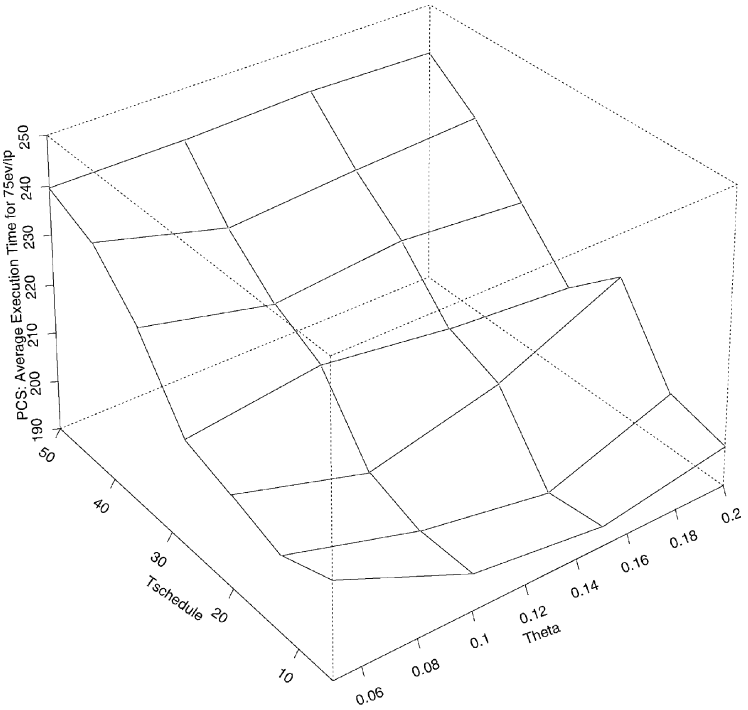

Fig. 4. *Perspective Plot of PCS, 75 events per LP:* Execution Time as a function of Θ and $T_{schedule}$. The minimum along the Θ axis is 0.05 and for the $T_{schedule}$ axis is 5 seconds.

Moreover, both Θ and $T_{schedule}$ affect total migration costs. As $T_{schedule}$ is increased, the opportunity for migration decision points decrease, which decreases the rate at which clusters could be moved. Likewise, if Θ is increased, the maximum difference in observed PAT values must be greater to effect any cluster migrations, thus reducing the likelihood that any cluster migration will happen, particularly for small to medium sized workload imbalances. As a result, one may have a tendency to set both Θ and $T_{schedule}$ to large values. However, we shall see that other factors such as reaction time and workload sensitivity suggest otherwise.

To better understand how migration costs, reaction time and workload sensitivity interact for particular set of Θ and $T_{schedule}$ values, we conducted experiments where we induced an internal workload on PCS with 75 events per LP and varied Θ and $T_{schedule}$ across a wide range of values to determine the set of values that minimizes execution time. The results of this experiment are shown in Fig. 4. The execution times are the average over three runs.

We observe that as Θ is increased from 0.05 to 0.15 and $T_{schedule}$ is increased from 5 to 15 seconds, the execution time drops. Please note that the origin of the graph in Fig. 4 is $\Theta = 0.05$ and $T_{schedule} = 5$ seconds. The reason for this behavior is because when both values are set low, the migration costs overshadow any benefits gained due to an increase in reaction time (small $T_{schedule}$) or sensitivity (small Θ). It is at this point the system is in the "valley" of low execution times and is where the system should operate to achieve the highest possible performance. However, as Θ and $T_{schedule}$ are increased beyond those values, the execution time increases and continues to do so. This behavior is attributed to a slow reaction time, which is caused by increasing $T_{schedule}$, and lower algorithm sensitivity, which is caused by increasing Θ.

Based on our experience with the BGE algorithm, we make the following guidelines for setting Θ and $T_{schedule}$ such that the "valley" of low execution times is discovered in as few runs as possible.

- Initially, configure $\Theta = 0.05$, and $T_{schedule} = 5$.
  By setting Θ and $T_{schedule}$ low, we avoid having to worry about the BGE algorithm not detecting an unbalanced workload or having a slow reaction time.
- If it is observed that the BGE algorithm is consistently initiating cluster migrations every $T_{schedule}$ epoch, particularly when the workload is balanced, increase Θ by 0.05 and $T_{schedule}$ by 5 seconds.
- If the total migration costs are greater than 10 percent of total execution time, then increase Θ by 0.05 and $T_{schedule}$ by 5 seconds. This will reduce the migration costs without having to be concerned about a slow algorithm reaction time or the BGE algorithm not detecting an unbalanced workload.

7.4 Performance Results

In this section we compare the execution time of GTW with and without our BGE algorithm with the introduction of a internal (static and time-varying) workloads. In all cases, the results presented are the average of three runs.

We define *Speedup* as the sequential execution time divided by GTW (parallel) execution time. For all the results presented here, a fast sequential simulator is used, which contains an optimized Calendar Queue [9]. The Calendar Queue has a $O(1)$ enqueue and dequeue time for the simulation models tested here. Consequently, our speedup

TABLE 2
Performance of BGE Algorithm in the Presence of Irregular Internal Workloads for GTW with BGE, $\Theta = 0.15$, and $T_{schedule} = 10$

Application	Speedup Improvement	Speedup
PHold, Static Workload	29%	1.8
PHold, Time-Varying Workload	35%	1.8
PCS, Static Workload	127%	2.2

results are very conservative compared to using a $O(log(n))$ data structure, such as the Skew Heap [43] or Splay Tree [42]. Our experiments indicate, that the sequential execution time for *null* event granularity PHold is 2.6 times faster when using the Calendar Queue than with the Skew Heap. Similar results where reiterated in a recent, comprehensive study of priority queue data structures by Ronngren and Ayani [39].

Next, we define *SpeedUp Improvement* as

$$(Speedup_{GTW-BGE} - Speedup_{GTW})/Speedup_{GTW}$$

where $Speedup_{GTW-BGE}$ is the speedup obtained with the BGE algorithm and $Speedup_{GTW}$ is the speedup obtained without the BGE.

As shown in Table 2, we observe good improvements in speedup, ranging from about 30 percent for PHold to almost 130 percent for PCS. However, the overall speedup may seem low. Closer examination of these internal workload reveals that the execution time obtained by GTW with BGE is within 5 percent of optimal for PHold and 50 percent of optimal for PCS.

To determine the optimal execution time for *PHold*, we use a technique called *Critical Path Analysis (CPA)*, developed by Berry and Jefferson [5]. CPA provides a means of analyzing the eventdependencies in a parallel simulation, such that the minimum execution time assuming an infinite number processors can be determined. The *critical path* of a simulation is the longest path, measured in real time, through the event dependency graph.

In constructing a dependency graph for the PHold model (static workload case) with the one millisecond event granularity for all LPs mapped to cluster 0, we note that there are very few interactions between LPs due to the self-initiating nature of this particular model. This allows us to simplify the critical path analysis by assuming that no dependencies exist between LPs (this assumption actually reduces the optimal execution time). Now, because of clustering, all LPs mapped to the same cluster will process events sequentially. Consequently, it is easy to see that the events processed by an LP assigned to cluster 0 form the critical path. This is due to their one millisecond event granularity, compared to the eight microsecond event granularity for events processed in other clusters.

The execution time of the critical path can be calculated as follows. First, there are 16 LPs with 25 initial events each assigned to cluster 0. Each of these events has an initial time stamp of about zero. Upon processing, an event is scheduled for the same LP 1 unit of simulated time into the future. Consequently, to advance 1,000 units of simulated time, the LPs of cluster 0 must process $16 * 25 * 1,000 = 400,000$ events. Each of these events require one millisecond of processing time, yielding an optimal execution time of 400 seconds. The execution time reported by GTW with BGE is 418 seconds, which is only 4.5 percent greater than the optimal.

The critical path in the time varying internal workload, PHold model is similar to the static internal workload. Here, the critical path starts with cluster 0 then migrates to cluster 31 and ends with cluster 63. Because the simulation end time was twice as long, the critical path is twice as long, yielding an optimal execution time of 800 seconds. For GTW with BGE, the best execution time is 840 seconds, which is only 5 percent greater than optimal.

Using Amdahl's Law, we observe that the PCS model performs within 50 percent of the optimal speedup for an infinite number processors with zero overhead for synchronization. Amdahl's Law states that compute time can be divided into the parallel portion and serial portion, and no matter how high the degree of parallelism in the former, the speedup will be asymptotically limited by the serial portion. To use this Law we first find the percentage of the PCS application that must be done sequentially. To do that, we observe that cluster 0 has about 40 times more work to perform per unit of time than the other clusters (computed by 1,000 event per LP divided by 25 events per LP). Consequently, cluster 0 comprises $(40/(40 + 127)) = 23.95\%$ of the entire simulation computation, and this computation must be performed serially, since all LPs in this cluster are mapped to the same processor. We also note that cluster 0 is the critical path, if we assume no dependencies. Using Amdahl's Law, the absolute best speedup is then $1/0.23.95 = 4.175$. Given that we are operating in a NOW environment with high communications overheads (i.e., no special purpose communications hardware is employed), a speedup of 2.2 does appear in line with what can be expected.

8 External Workload Experiments

In this section, the results from our experimental study are presented. Here, we use eight 167-MHZ Sun Sparc Ultra-1 workstations running version 2.5 of the Solaris operating system. A workload manager program ensures the external workloads are consistently induced. All results presented are the average over three trials.

8.1 PHold

Both the *null* and *1ms* event granularity PHold models are configured with 2,048 LPs and 25 initial messages per LP, yielding a total message population of 51,200. These LPs are grouped into 128 clusters, 16 LPs each. These 128 cluster are evenly distributed onto the eight processors, giving each processor 16 clusters, or 256 LPs. The above PHold configuration parameters are used in all experiments presented here.

8.2 PCS

We configure the *call initiated* PCS model with 2,048 cells and 25 portables per cell. The number of channels per cell is 10. Call holding times are exponentially distributed with a mean of three minutes. Call mobility rate is exponentially

TABLE 3
Performance of BGE Algorithm in the Presence of External Workloads

Application	Speedup Improvement	BGE Speedup (4-task case)
Null PHold, Static Workload	30% (1-task) to 170% (4-task)	1.8
PCS, Static Workload	40% (1-task) to 180% (4-task)	2.3
1ms PHold, Static Workload	50% (1-task) to 260% (4-task)	6.3
Null PHold, Time-Varying Workload	28% (4-task, 250 sec.)	1.7
PCS, Time-Varying Workload	20% (4-task, 250 sec.)	2.1
1ms PHold, Time-Varying Workload	24% (4-task, 250 sec.)	5.7

distributed with a mean of 1 every 75 minutes. Call inter-arrival times per portable are exponentially distributed with a mean of 10 minutes. LPs are divided into 128 clusters, 16 LPs to a cluster, and 16 clusters to a processor.

8.3 Performance Results

In this section, we compare the execution time of GTW with and without our BGE algorithm in the presence of external (static and time-varying) workloads. In all cases, the results presented are the average of three runs. A summary of the performance results is shown in Table 3.

For the static workload experiments, we induce three, increasingly larger, static workloads on PE 1 and run each application (*null* PHold, PCS, and *1ms* PHold). In the first case, a single external task is induced on PE 1. We then induce two tasks on PE 1, and in the third case, four external task are induced on PE 1. Each external task is CPU bound and performs no external I/O.

For the time-varying workload experiments, we induce two, increasingly larger static workloads on PE 1, with a varying *on* and *off-period* and run each application. In the first case, a two task external workload is induced on PE 1. We then induce four tasks on PE 1. In each case, we vary the *on* and *off periods*, but keep them equal. That is to say that the *on period* is equal to the *off period*. Consequently, all of the time-varying workloads have a 50 percent duty cycle. Here, we vary the *on/off period* among the following set of values: 25, 50, 150, and 250 seconds.

First, we present the comparison results for *null* event granularity PHold. We observe that GTW with BGE is consistently faster than GTW without it, ranging from 30 percent faster in the one task case to 170 percent faster in the four task case. It was determined that relatively small values of $\Theta =$ and $T_{schedule}$ yield the fastest execution times.

Despite the significant improvements in speedup, GTW with BGE is still only able to obtain a speedup of about 2 on eight processors. However, given the low event granularity of the PHold application combined with the high overheads for sending and receiving messages, these results appear in line with what can be expected.

Next, we compare the performance of GTW with and without the BGE algorithm using the PCS model. We observed a similar pattern to that previously shown for the *null* event granularity PHold model, where GTW with BGE for all static workloads yields shorter execution times. However, for PCS, the fastest execution results when $T_{schedule} = 30$ as opposed to $T_{schedule} = 20$ for *null* event granularity PHold. The reason this increases in $T_{schedule}$ is because the move costs for PCS are higher because of a larger message size. For PCS, the message data contained in an event is 72 bytes, compared to only 8 bytes for PHold.

In this last series of static external workload experiments, we compare GTW with and without the BGE algorithm running *1ms* event granularity PHold. Similar to the previous two series of experiments, we again observe that GTW with the BGE algorithm completes *1ms* event granularity PHold with significantly shorter execution times than plain GTW, yielding a peak performance improvement of 260 percent in the four task case.

Now, unlike the previous applications, when running 1ms PHold, PE 1 was deallocated from the usable set of processors, as shown in Fig. 5. The primary difference between these applications is event granularity. Consequently, these findings suggest that event granularity plays an important role in determining when a processor should be deemed unusable.

Last, we observe some anomalous cluster allocations occurring during the deallocation of PE 1. It appears another processor during the same epoch has 11 of its 16 clusters redistributed. This behavior is attributed to inaccuracies in cluster CPU utilization times. We will revisit this phenomenon in the next section and provide a detailed explanation for its occurrence.

For the time-varying workloads, it was determined that $\Theta = 0.15$ and a $T_{schedule} = 50$ did the best overall at consistently detecting and migrating the load imbalance at the appropriate points. However, these settings appear to conflict with the best settings for internal and static external workloads, particularly for $T_{schedule}$. For these workloads, the fastest execution time results when $T_{schedule}$ was set to a much lower value. We attribute this phenomenon to an increase in inaccurate cluster CPU utilizations being reported. These inaccuracies will be quantified later in this section.

For *Null* PHold in the presence of the two task, time-varying workload, we observe that in each case, GTW with BGE is faster than GTW without BGE, ranging from 10 percent in the 150 second case to almost 30 percent faster in the 250 second case. For the four task workload, we observed similar speedup improvements.

Now, the observed improvement in speedup may seem low (only about 30 percent in the four task, 250 second case), however, when the amount of lost computation power due to the time-varying workload is considered, this improvement is in line with what can be expected. For these experiments, the four task time-varying workload has a

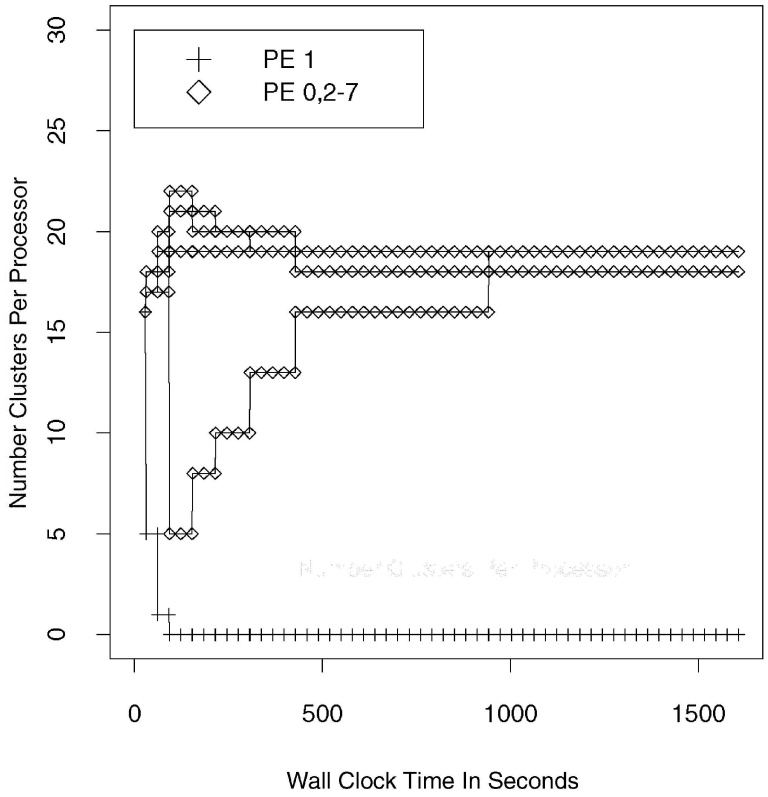

Fig. 5. *1ms Event Granularity PHold:* Cluster Allocation over Time with Four Static, External Tasks on PE 1.

50 percent duty cycle, thus giving GTW a full eight PEs half the time and 7.2 PEs when the workload is active. 7.2 PEs is calculated by the fact that GTW on PE 1 is given 20 percent of PE 1s available CPU cycles (i.e., $1/5$ since there are four external tasks plus GTW all vying for PE 1), plus the other seven machines. Thus, on average, GTW is given $(7.2 + 8)/2 = 7.6$ processors, which means that GTW is losing 0.4 processors, or only about 5 percent of the available computing power.

Next, we present the time-varying results using PCS. Here, for both the two and four task, time-varying workloads, the observed performance is somewhat different from that of *null* event granularity PHold. The primary difference is in the 50 and 150 second cases. In both cases, we observe little or no reduction in execution time. We attribute this phenomenon to a combination of thrashing migrations every $T_{schedule}$ epoch and higher migration overheads for the PCS. Because of these higher migration costs in the PCS model, only a 20 percent increase in speedup is obtained for GTW with the BGE algorithm, shown in Table 3.

Last, we present the results using *1ms* event granularity PHold. Here we observe that GTW with BGE consistently yields shorter execution times than GTW without BGE both the two task and four task, time-varying external workloads. Even the 50 second case with four tasks yields lower execution times, which was not the case for *null* event granularity PHold. We attribute these findings to the low migration overheads of the PHold model, combined with the one millisecond event granularity. Despite the somewhat thrashing cluster migrations made by the BGE algorithm in the 50-second case, these large event granularities appear to mask them, enabling GTW with BGE to process events at slightly faster event rate than GTW without BGE.

We observe a reasonable increase in speedup given the 5 percent loss of total computing power due to the four task, time-varying external workload. This speedup is attributed to the reduction in number of aborted events which is a consequence of the BGE algorithm migrating clusters off PE 1. In fact, PE 1 is *deallocated* from the usable set during the workload's *on-period*, and then reallocated when workload sleeps, as shown in Fig. 6.

However, what is surprising about these cluster allocations, is that there appears to be an additional processor that has lost its clusters at the same time PE 1 is being deallocated. This phenomena was also observed in the four task, static external workload case for *1ms* event granularity PHold. It appears as if there exists an additional time-varying workload being induced on one of the other processors. Further examination of the BGE's trace files reveals that additional external workloads did not exist. So if this phenomena is not the result of an additional external workload, then what is the cause? Upon closer examination of a the BGE algorithm decisions, the single cluster mapped to PE 1 contains an over-inflated CAT value due to inaccuracies being reported in CPU utilization for that cluster. When this cluster is migrated off PE 1 as a consequence of normal BGE processing, the receiving processor now appears to be overloaded with work. Consequently, the BGE algorithm begins to migrate clusters off that processor, as well.

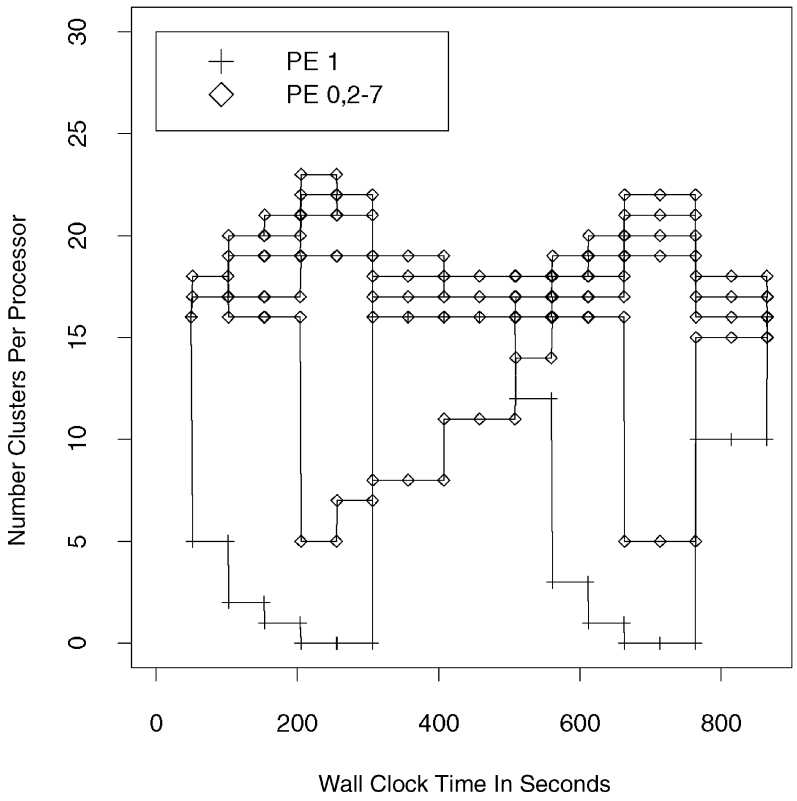

Fig. 6. *1ms Event Granularity PHold:* cluster allocation over time with four time-varying (250 seconds case), external tasks on PE 1. For GTW with BGE, $\Theta = 0.15$, and $T_{schedule} = 50$.

Now, in many of the previous experiments, we have attributed particular anomalous behavior to inaccuracies being reported in cluster CPU utilization. To quantify these inaccuracies, GTW was instrumented to record the number of events and event processing times for the purpose of creating a histogram. We then reran *1ms* event granularity PHold with four task, time-varying external workload on PHold. The results from this experiment were very surprising.

The histogram, shown in Fig. 7, confirms that PE 1 is recording numerous perturbations of event processing times. There were over 500 events recorded with event processing on the order of 160 milliseconds. The actual event processing time should have been only one millisecond. In another case, over 300 events were recorded with event processing times in excess of 300 milliseconds. These collective perturbations make PE 1 appear as though it is consuming 75 percent more CPU time than it is in reality. We attribute these timer perturbations to timed event computations being charged for cycles they do not consume, because the hardware clock continues to run while the GTW process has been interrupted by the operating system to allow the other external tasks time to execute.

9 Heterogeneous Experiments

The results from our heterogeneous platform study are discussed in this section. For all experimental data presented here, we employed the use of three different kinds of machines: 1) 167 MHZ Sun Sparc Ultra-1 workstation running version 2.5 of the Solaris operating system, 2) 200 MHZ SGI Indy workstation running version 6.2 of the IRIX operating system, and 3) 40 MHZ Sun Sparc IPX running version 2.5 of the Solaris operating system.

9.1 Determining σ

In order to determine the appropriate value for σ, we compared the sequential execution of the PCS simulation among the various platforms used in these experiments. We determined that the Sun Ultra is about two times faster than the SGI Indy workstation and is about 22 times faster than Sun IPX workstation, thus yielding a σ value of 22.0 for the Ultra workstations, 11.0 for the Indy workstation and 1.0 for the Sun IPX workstation.

9.2 Performance Results

In this section, we present the results from the comparison of GTW with and without the BGE algorithm in the presence of heterogeneous processors. Here, we create two different heterogeneous configurations. The first consists of seven Sun Ultra workstations and one SGI Indy. This configuration is referred to as *7-fast and 1-slow*. The second configuration consists of six Sun Ultra workstations, one SGI Indy, and one Sun IPX. This configuration is dubbed *6-fast and 2-slow*. For these experiments, the PCS model is used.

For both configurations, the optimal value pair set of Θ and $T_{schedule}$ was determined to be 0.15 and 30 seconds,

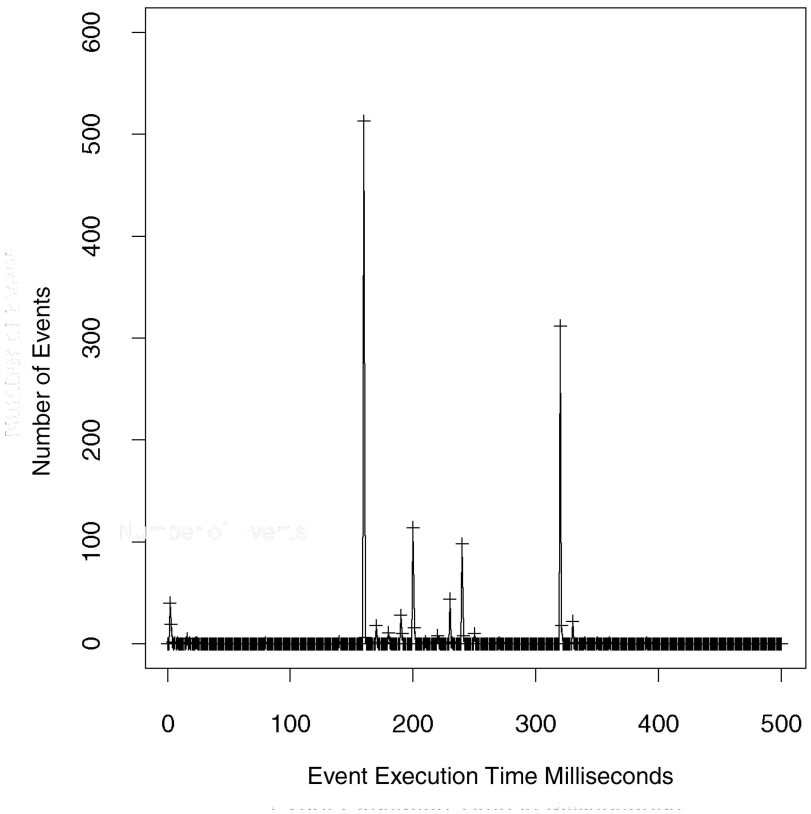

Fig. 7. *1ms Event Granularity PHold:* histogram of event processing times for loaded PE 1, which is four time-varying (250 seconds case), external tasks.

respectively. This is the same value set pair that was deemed best for PCS in the static, external workload experiments. These results underscore the observation that processor heterogeneity and static external workloads are in fact duals of each other. That is to say, from the point of view of a Time Warp program, a processor with half the computing power behaves as if it has twice the static external workload as the other processors.

In this first series of heterogeneous experiments, we present the results for the 7-fast and 1-slow configuration. Here, we observe an almost 70 percent improvement in speedup, which mirror static external workload results.

The cluster allocations for this configuration, shown in Fig. 8, reveal a picture that is very similar to that found for the static external workloads. Here, we observe the slow processor, denoted by PE 7, having about half its workload removed and redistributed among the other processors. This is what one would expect to happen given that PE 7 is half as fast as the other processors. Consequently, it should be allocated half the number of clusters.

Next, we present the results for the 6-fast and 2-slow configuration. The purpose of this analysis is to see how significant the performance degradation is when we use computing technology that has a large difference in processing capabilities. As previously indicated, in this configuration the Sun Ultra is about 22 times faster than the Sun IPX workstation. When we compare the performance for the 6-fast and 2-slow configuration with the 7-fast and 1-slow configuration, we observed a 25 percent decrease in speedup. Now, because we are reducing the total amount of computing power by 12 percent, this accounts for about half the observed performance degradation. The other half is accounted for by the 700 percent increase in aborted events in the 6-fast and 2-slow configuration. The reason for these aborted events is because Sun IPX with one cluster (see cluster allocations in Fig. 9, PE 7) is still not progressing fast enough. Recall that the Sun IPX is 22 times slower than the Sun Ultra, however one cluster represents about $1/16$ of the workload that is assigned to Sun Ultras. Consequently, the Sun IPX is still overloaded with work, despite having only one cluster, but is not sufficiently overloaded to warrant deallocation by the BGE algorithm. Because the Sun IPX is overloaded, the rate of GVT's advance is slowed, which causes the Sun Ultras to become overly optimistic and abort events. This suggests that LP clusters may need to be divided into smaller units of computation, to allow the slower IPX processor to share a smaller portion of the workload.

10 Combination-Workload Experiments

For all experimental data presented here, the PCS simulation model is used. The configuration remains unchanged from that previously used in other experiments. The one exception is that for LPs mapped to cluster 0, which in turn is mapped to PE 0, 1,000 initial events are assigned. All other LPs are assigned 25 initial events. This was done to recreate the static internal workload used for PCS experi-

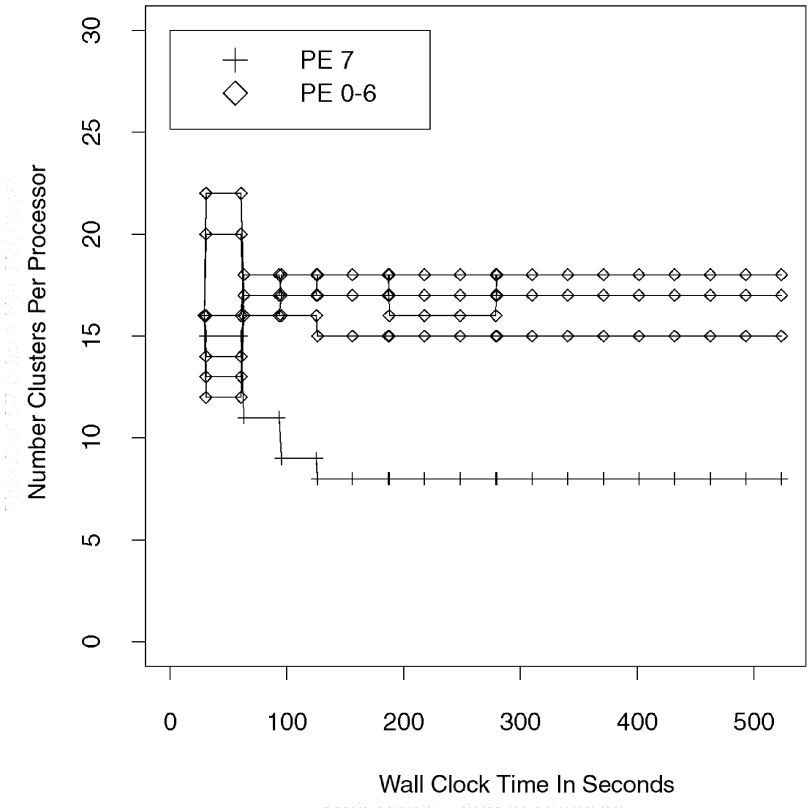

Fig. 8. *PCS:* cluster allocation over time on 7-fast processor and 1-slow processor (best case). For GTW with BGE, $\Theta = 0.15$, and $T_{schedule} = 30$.

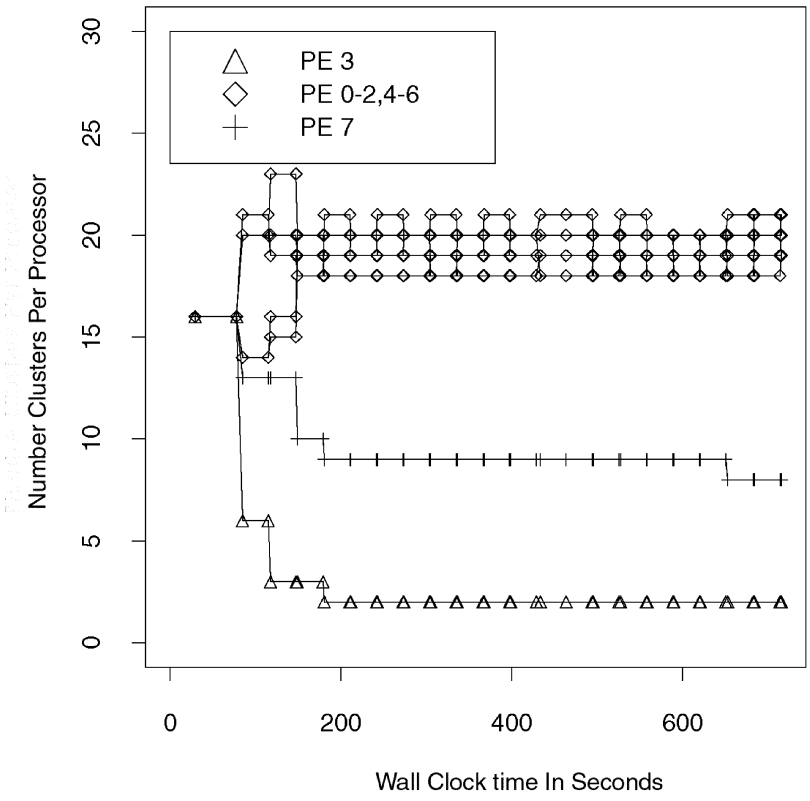

Fig. 9. *PCS:* cluster allocation over time on six fast processors and two slow processors. For GTW with BGE, $\Theta = 0.15$, and $T_{schedule} = 30$.

ments presented in Section 7. The experiments are performed on eight 167 MHZ Sun Ultra-1 workstations, where a four task, time-varying workload is induced on PE 1. The *on period* for this workload is 150 seconds, however, the *off period* is only 50 seconds. Thus, this workload has a 75 percent duty cycle.

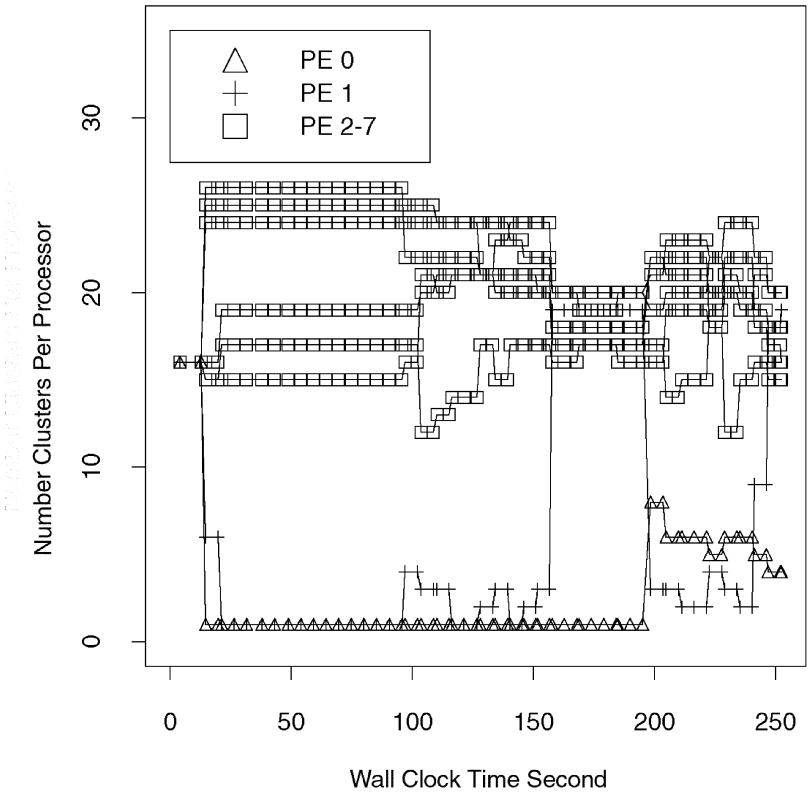

Fig. 10. *PCS:* cluster allocation over time with combination of four time-varying (150/50 second case), external tasks on PE 1 and static internal workload on PE 0. For GTW with BGE, $\Theta = 0.05$, and $T_{schedule} = 5$.

We observed that GTW with the BGE algorithm improves speedup by 100 percent over GTW without BGE. As seen in previous experiments, the BGE algorithm migrates the appropriate number of clusters off the overloaded processors and redistributing them among the underloaded processors, as shown in Fig. 10. It is observed that both PE 0s and 1s cluster allocation is reduced to that of a single cluster. However, at 150 seconds into the execution of the simulation, PE 1 is reloaded with work. This phenomenon is in response to the time-varying external workload on PE 1 going to sleep for 50 seconds.

11 Conclusions and Future Work

For Time Warp programs executing on a NOW environment, there are internal and external workload sources that must be taken into consideration if efficient execution is to be maintained. The principal contribution of this work is devising a single algorithm that is able to mitigate both kinds of irregular workloads. The observation driving this algorithm is that in order for a Time Warp program to be balanced, the amount of wall clock time necessary to advance an LP one unit of simulation time should be about the same for all LPs in the system. In particular, we have demonstrated using a PCS simulation model as well as a synthetic application that our Background Execution Algorithm (BGE) is able to:

- 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 redistribute the workload on the existing set of processors as some become more heavily or lightly loaded by changing externally or internally induced workloads.

During the course of this experimental study, two implementation-specific limitations were discovered. The first concerns the inability of our algorithm to detect internal workload imbalances caused by a sharp change in the simulations communication pattern. This limitation is a result of current commercial Unix operating systems, such as Sun Solaris and SGI IRIX, not providing applications a mechanism for determining how much time the operating system spends processing socket operations on the behalf of the application. One solution to this problem is to use a high-performance message-passing system, such as Myrinet [6], or Rosu et al.'s fast ATM firmware [40]. In these systems, the operating system is avoided, placing all message-passing overheads in GTW's address space and thus allowing the communication overheads to be easily obtained by direct measurement. Here, each cluster's CAT value would then be the sum of the amount of wall clock

time spent processing committed events plus the time spent sending and receiving committed off-processor messages.

The second implementation-specific limitation concerns the accuracy of event processing times. Because the timers used are free running, monotonically increasing, hardware clocks, timed event computations may include the time for operating system related tasks or time slices of other external workloads, should the operating system de-schedule the current running process during the timing of an event.

The obvious solution to these limitations is to modify the operating system such that better performance analysis support is provided. However, this solution lacks portability across many platforms. To ensure application portability, techniques need to be developed that make use of the current operating system interfaces and enable the accurate approximation of system resource utilization. One possible solution we plan to investigate in the future is using context switch statics provided by the `getrusage` system call combined with operating system time slicing statistics to develop an error potential measurement. This error potential will then be subtracted from the cluster CPU utilization statistics to increase the accuracy of the timing measurements, thus making the BGE algorithm more accurate.

While the focus of this work has centered on efficient execution of Time Warp on NOW platforms, we believe these results are applicable in other synchronization protocols. The key observation made by this work is based on a fundamental truth that not only applies to Time Warp programs, but to *any* data-parallel, distributed simulation synchronization protocol. For example, in a conservative synchronization scheme, if the amount of wall clock time required to advance an LP one unit of simulation time differs among the various processors, then some form a load imbalance exists.

Finally, when viewed from a higher level, Time Warp load management techniques, such as the one presented here, are stability assurance mechanisms (SAMs), which monitor the Time Warp system and make changes in the behavior of the system to maintain efficient execution. Other SAMs include flow-control, and adaptive memory buffer management techniques. An open question is how do these various SAMs interoperate? At the very least these mechanisms should be designed such that they do not interfere with one another or feed back on each other in a manner that degrades performance. However, an even more interesting question than the issue of interoperability is the existence of a unifying SAM that encompasses all types of SAMs for Time Warp systems. What about for all distributed simulation protocols in general? To answer these questions, further investigation is required.

ACKNOWLEDGMENTS

This work was supported by the U.S. Army, Contract DASG60-95-C-0103, and funded by the Ballistic Missile Defense Organization, and the U.S. National Science Foundation, Grant Number CDA-9501637.

REFERENCES

[1] Y. Artsy and R. Finkel, "Designing a Process Migration Facility: The Charlotte Experience," *Computer,* vol. 22, no. 9, pp. 47–53, Sept. 1989

[2] H. Avril and C. Tropper, "The Dynamic Load Balancing of Clustered Time Warp for Logic Simulation," *Proc. 10th Workshop on Parallel and Distributed Simulation (PADS '96),* pp. 20–27, May 1996.

[3] A. Barak, S. Guday, and R.G. Wheeler, *The MOSIX Distributed Operating System: Load Balancing for Unix.* Springer Verlag, 1993.

[4] S. Bellenot, "Performance of a Risk-Free Time Warp Operating System," *Proc. Seventh Workshop on Parallel and Distributed Simulation (PADS '96),* pp. 155–158, vol. 23, May 1993.

[5] O. Berry and D. Jefferson, "Critical Path Analysis of Distributed Simulation," *Proc. 1985 SCS Multiconference on Distributed Simulation,* pp. 57–60, Jan. 1985.

[6] N.J. Boden, D. Cohen, R. Felderman, A.E Kulawik, C.L. Setiz, J.N. Seizovic, and W.-K. Su, "Myrinet: A Gigabit per Second Local Area Network," *IEEE Micro,* vol. 15, no. 1, pp. 29–36, Feb. 1995.

[7] A. Boukerche and S.K. Das, "Dynamic Load Balancing Strategies for Conservative Parallel Simulations," *Proc. 11th Workshop on Parallel and Distributed Simulation,* pp. 20–28, June 1997.

[8] J. Briner, "Fast Parallel Simulation of Digital Systems," *Advances in Parallel and Distributed Simulation,* vol. 23, pp. 71–77, Jan. 1991.

[9] R. Brown, "Calendar Queues: A Fast 0(1) Priority Queue Implementation for the Simulation Event Set Problem," *Comm. ACM,* vol. 31, no. 10, pp. 1,220–1,227, Oct. 1988.

[10] C. Burdorf and J. Marti, "Load Balancing Strategies for Time Warp on Multiuser Workstations" *The Computer J.,* vol. 36, no. 2, pp. 168–176, 1993.

[11] C.D. Carothers, R.M. Fujimoto, and Y.-B. Lin, "A Case Study in Simulating PCS Networks Using Time Warp," *Proc. Ninth Workshop on Parallel and Distributed Simulation,* pp. 87–94, 1995.

[12] C.D. Carothers, Y.-B. Lin, and R.M. Fujimoto, "A Redial Model for Personal Communications Services Networks," *Proc. 1995 Vehicular Technology Conf. (VTC '95),* pp. 135–139, July 1995.

[13] K. Chanchio and X.-H. Sun, "Efficient Process Migration for Parallel Processing on Nondedicated Networks of Workstations," Technical Report 96-74, ICASE, Dec. 1996.

[14] K.M. Chandy and J. Misra, "Distributed Simulation: A Case Study in Design and Verification of Distributed Programs," *IEEE Trans. Software Eng.,* vol. 5, no. 5, pp. 440–452, Sept. 1979.

[15] D.C. Cox, "Personal Communications–A Viewpoint," *IEEE Comm. Magazine,* vol. 128, no. 11, 1990.

[16] S. Das, R. Fujimoto, K. Panesar, D. Allison, and M. Hybinette, "GTW: A Time Warp System for Shared Memory Multiprocessors," *1994 Winter Simulation Conf. Proc.,* pp. 1,332–1,339, Dec. 1994.

[17] P.M. Dickens and P.F. Reynolds, Jr., "SRADS with Local Rollback," *Proc. SCS Multiconference on Distributed Simulation,* pp. 161–164, vol. 22, Jan. 1990.

[18] R.M. Fujimoto, "Time Warp on a Shared Memory Multiprocessor," *Proc. 1989 Int'l Conf. Parallel Processing,* vol. 3, pp. 242–249, Aug. 1989.

[19] R.M. Fujimoto, "Parallel Discrete Event Simulation," *Comm. ACM,* vol. 33, no. 10, pp. 30–53, Oct. 1990.

[20] R.M. Fujimoto, "Performance of Time Warp under Synthetic Workloads," *Proc. SCS Multiconference on Distributed Simulation,* vol. 22, pp. 23–28, Jan. 1990.

[21] R.M. Fujimoto and M. Hybinette, "Computing Global Virtual Time on Shared-Memory Multiprocessors," technical report, College of Computing, Georgia Inst. Technology, Aug. 1994.

[22] R.M. Fujimoto, S.R. Das, K.S. Panesar, M. Hybinette, and C. Carothers, "Georgia Tech. Time Warp Programmer's Manual for Distributed Network of Workstations," Technical Report GIT-CC-97-18, College of Computing, Georgia Inst. Technology, July 1997.

[23] A. Geist, A. Beguelin, J. Dongarra, W. Jian, R. Manchek, and V. Sunderam, "Pvm 3 User's Guide and Reference Manual," Technical Report TM-12187, Oak Ridge Nat'l Laboratory, May 1993.

[24] D.W. Glazer and C. Tropper, "On Process Migration and Load Balancing in Time Warp," *IEEE Trans. Parallel and Distributed Systems,* vol. 3, no. 4, pp. 318–327, Mar. 1993.

[25] A. Goldberg, "Virtual Time Synchronization of Rplicated Processes," *Proc. Sixth Workshop on Parallel and Distributed Simulation,* vol. 24, pp. 107–116, Jan. 1992.
[26] F. Hao, K. Wilson, R.M. Fujimoto, and E. Zegura, "Logical Process Size in Parallel ATM Simulations," *1996 Winter Simulation Conf. Proc.,* pp. 645–652, Dec. 1996.
[27] D.R. Jefferson, "Virtual Time," *ACM Trans. Programming Languages and Systems,* vol. 7, no. 3, pp. 404–425, July 1985.
[28] D.R. Jefferson and H. Sowizral, "Fast Concurrent Simulation Using the Time Warp Mechanism, Part I: Local Control," Technical Report N-1906-AF, RAND Corporation, Dec. 1982.
[29] M. Litzkow and M. Livny, "Experience with the Condor Distributed Batch System," *Proc. IEEE Workshop on Experimental Distributed Systems,* Oct. 1990.
[30] B.D. Lubachevsky, A. Shwartz, and A. Weiss, "An Analysis of Rollback-Based Simulation," *ACM Trans. Modeling and Computer Simulation,* vol. 1, no. 2, pp. 154–193, Apr. 1991.
[31] V.K. Madisetti, D.A. Hardaker, and R.M. Fujimoto, "The Mimdix Operating System forParallel Simulation and Supercomputing," *J. Parallel and Distributed Computing,* vol. 18, no. 4, pp. 473–483, Aug. 1993.
[32] F. Mattern, "Efficient Distributed Snapshots and Global Virtual Time Algorithms for Non-Fifo Systems," *J. Parallel and Distributed Computing,* vol. 18, no. 4, pp. 423–434, Aug. 1993.
[33] S.J. Mullender, G. van Rossum, R. van Renesse, and H. van Staveren, "Amoeba–A Distributed Operating System for the 1990s, *Computer,* vol. 23, no. 5, pp. 44–53, May 1990.
[34] D.M. Nicol, Performance Bounds on Parallel Self-Initiating Discrete Event Simulations, *ACM Trans. Modeling and Computer Simulation,* vol. 1, no. 1, pp. 24–50, Jan. 1991.
[35] J.K. Osterhout, A.R. Cherenson, F. Douglis, M.N. Nelson, and B.B. Welch, "The Sprite Network Operating System," *Computer,* vol. 21, no. 2, pp. 23–36, Feb. 1988.
[36] K. Panesar and R.M. Fujimoto, "Adaptive Flow Control in Time Warp," *Proc. 11th Workshop on Parallel and Distributed Simulation,* June 1997.
[37] P.L. Reiher and D. Jefferson, "Dynamic Load Management in the Time Warp Operating System," *Trans. Society for Computer Simulation,* vol. 7, no. 2, pp. 91–120, June 1990.
[38] D.O. Rich and R.E. Michelsen, "An Assessment of the Modsim/TWOS Parallel Simulation Environment," *1991 Winter Simulation Conf. Proc.,* pp. 509–518, Dec. 1991.
[39] R. Ronngren and R. Ayani, "Parallel and Sequential Priority Queue Algorithms," *ACM Trans. Modeling and Simulation,* vol. 7, no. 2, pp. 157–209, Apr. 1997.
[40] MC. Rosu, K. Schwan, and R.M. Fujimoto, "Supporting Parallel Applications on Clusters of Workstations: The Intelligent Net–work Interface Approach," *Proc. 1997 IEEE High Performance Distributed Computing,* Aug. 1997.
[41] R.S. Haft, M. Ruhwandl, C. Sporrer, and H. Bauer, "Dynamic Load Balancing of a Multicluster Simulation on a Network of Workstations," *Proc. Ninth Workshop on Parallel and Distributed Simulation,* vol. 24, pp. 175–180, July 1995.
[42] D.D. Sleator and R.E. Tarjan, "Self-Adjusting Binary SearchTrees," *J. ACM,* vol. 32, no. 3, pp. 652–686, July 1985.
[43] D.D. Sleator and R.E. Tarjan, "Self-Adjusting Binary Heaps," *SIAM J. Computing,* vol. 15, no. 1, pp. 52–69, Feb. 1986.
[44] L. Soule and A. Gupta, "An Evaluation of the Chand-Misra-Byrant Algorithm for Digital Logic Simulation," *ACM Trans. Modeling and Computer Simulation,* vol. 1, no. 4, Oct. 1991.
[45] S. Srinivasan and P.F. Reynolds, Jr., "NPSI Adaptive Synchronization Algorithms for PDES," *Proc. 1995 Winter Simulation Conf. (WSC '95),* pp. 658-665, Dec. 1995.
[46] J. Steinman, "Speedes: A Multiple-Synchronization Environment for Parallel Discrete Event Simulation," *Int'l J. in Computer Simulation,* vol. 2, no. 3, pp. 251–286, Mar. 1992.
[47] J. Steinman, "Breathing Time Warp," *Proc. Seventh Workshop on Parallel and Distributed Simulation,* vol. 23, pp. 109–118, May 1993.
[48] THAAD Project Office, "Software Product Specification for the THAAD Integrate System Effectiveness Simulation (TISES),"user documentation for TISES software, Sept. 1995.
[49] M.M. Theimer, A. Lantz, and D.R. Cheriton, "Preemptable Remote Execution Facilities for the V-System," *Proc. 10th ACM Symp. Operating System Principles,* Dec. 1985.
[50] G. Thiel, "Locus Operating System: A Transparent System," *Computer Comm.,* vol. 14, no. 6, pp. 336–346, 1991.
[51] F. Wieland, E. Blair, and T. Zukas, "Parallel Discrete Event Simulation (PDES): A Case Study in Design, Development, and Performance Using Speedes," *Proc. Ninth Workshop on Parallel and Distributed Simulation,* pp. 103–110, June 1995.
[52] M.H. Willebeek-LeMair and A.P. Reeves, "Strategies for Dynamic Load Balancing on Highly Parallel Computers," *IEEE Trans. Parallel and Distributed Systems,* vol. 4. no. 9, pp. 979–993, Sept. 1993.
[53] L.F. Wilson and D.M. Nicol, "Experiments in Automated Load Balancing," *Proc. 10th Workshop on Parallel and Distributed Simulation,* pp. 4–11, May 1996.
[54] Z. Xiao and F. Gomes, "Benchmarking SMTW with an SS7 Performance Model Simulation," unpublished project report for CPSC 601.24, Fall 1993.
[55] E.R. Zayas, "Attacking the Process Migration Bottleneck," *Proc. 11th ACM Symp. Operating System Principles,* pp. 13–24, 1987.
[56] W. Zhu and A. Goscinski, "Process Migration in RHODOS," Technical Report CS90/9, Dept. Computer Science, Univ. College, Univ. of New South Wales, Mar. 1990.

Christopher D. Carothers is an assistant professor in the Computer Science Department at Rensselaer Polytechnic Institute. He received the BS, MS, and PhD degrees from the Georgia Institute of Technology in 1991, 1996, and 1997, respectively. Prior to joining Rensselaer, he was a research scientist at the Georgia Institute of Technology. As a PhD student, he interned twice with Bellcore, where he worked on wireless network models. In the summer of 1996, he interned at the MITRE Corporation, where he was part of the DoD High Level Architecture development team. His research interests include parallel and distributed systems, simulation, wireless networks, and computer architecture.

Richard M. Fujimoto is a professor in the College of Computing at the Georgia Institute of Technology. He received the BS degrees from the University of Illinois (Urbana) in 1977 and 1978 in computer science and computer engineering and the MS and PhD degrees in computer science and electrical engineering from the University of California (Berkeley) in 1980 and 1983. He has been an active researcher in the parallel and distributed simulation community since 1985, and has published over 70 technical papers in this field. He has given several tutorials on parallel and distributed simulation at leading conferences, and has coauthored a book on parallel processing. He served as chair of the technical working group responsible for defining the time management services for the DoD High Level Architecture (HLA). Fujimoto is an area editor for *ACM Transactions on Modeling and Computer Simulation*. He also served as chair of the steering committee for the Workshop on Parallel and Distributed Simulation (PADS) from 1990 to 1998. Dr. Fujimoto is a member of the IEEE Computer Society