Proceedings of the 2004 Winter Simulation Conference
R .G. Ingalls, M. D. Rossetti, J. S. Smith, and B. A. Peters, eds.

CONTROLLING OVER-OPTIMISM IN TIME-WARP VIA CPU-BASED FLOW CONTROL

Vinay Sachdev
Maria Hybinette
Eileen Kraemer

Computer Science Department
415 Boyd Graduate Studies Research Center
The University of Georgia
Athens, GA 30602, U.S.A.

ABSTRACT

In standard optimistic parallel event simulation, no restriction exists on the maximum lag in simulation time between the fastest and slowest logical processes (LPs). Over-optimistic applications exhibit a large lag, which encourages rollback and may degrade performance. We investigate an approach for controlling over-optimism that classifies LPs as FAST, MEDIUM, or SLOW and migrates FAST and/or SLOW processes. FAST LPs are aggregated, forcing them to compete for CPU cycles. SLOW LPs are dispersed, to limit their competition for CPU cycles. The approach was implemented on distributed Georgia Tech Time Warp(GTW)(Das et al. 1994) and experiments performed using the synthetic application *P-Hold*(Fujimoto 1990). For over-optimistic test cases, our approach was found to perform 1.25 to 2.75 times better than the standard approach in terms of useful work and to exhibit execution times shorter than or equal to the standard computation.

1 INTRODUCTION

Parallel simulation based on the optimistic Time Warp protocol(Jefferson and Sowizral 1985) is widely used in large-scale simulations such as those of air-traffic control and the World Wide Web. A primary concern of such simulations is good performance. However, these simulations are subject to the problem of *over-optimism*, in which some logical processes (LPs) progress far beyond others. To facilitate discussion of the nature of inefficiencies that result from over-optimism, we first present some essential background on optimistic discrete event simulation using Time Warp.

A discrete event simulation consists of a collection of logical processes (LPs), which may execute on different processors (PEs). The simulation is driven by the exchange of timestamped message by the LPs. Consistency in the processing of messages requires that all events be processed in timestamp order. Two main synchronization protocols exist: *conservative* and *optimistic.* The conservative protocol enforces consistency by avoiding the possibility of ever receiving an event in the past. In contrast, the optimistic protocol permits the receipt of an event from the past but responds by "rolling back" events that were optimistically processed too early.

Each LP in an optimistic simulation maintains a current *logical clock (local virtual time, or LVT);* whenever an LP receives a message with a timestamp earlier than LVT, it rolls back its execution to the time-stamp before that of the arrived message. Such an out-of-order message is called a *straggler message*. When rollback is necessary an LP reverts to the appropriate previous state and "un-schedules" any messages sent prior to the rollback. To support this, LPs maintain a history of state information and keep a record (an *anti-message)* for each message sent. In the case of rollback, the LP sends anti-messages, which annihilate the original messages sent. Anti-messages may cause additional rollbacks, called secondary rollbacks.

The earliest timestamp of any unprocessed or partially processed message in the system defines *Global Virtual Time (GVT)*. A message is guaranteed not to rollback if its timestamp is earlier than GVT. Thus, memory for events with timestamps *before* GVT, along with their corresponding state and anti-messages, can be reclaimed in a process known as *fossil collection.*

Over-optimism results in poor memory utilization because it creates a wide gap between GVT and the most recent timestamp in the system. State information, event histories, and anti-messages must be stored for all computation with a simulation time later than GVT. If over-optimism is not controlled, the memory requirement may grow to the point that memory becomes exhausted.

Another inefficiency caused by over-optimistic behavior is that rollbacks may be quite long when slower LPs send a message to faster (over-optimistic) LPs. Previously

processed events are rolled back and anti-messages are sent to cancel the events scheduled as a result of the events currently being rolled back. This nullifies all the useful computation done by both the over-optimistic LPs and the LPs to which the anti-messages were sent. Further, useful CPU cycles are wasted in undoing the work already done.

Finally, long and frequent rollbacks result in a large number of anti-messages being sent. These anti-messages utilize bandwidth that could otherwise have been used for useful communication. Thus, over-optimistic behavior of LPs can cause inefficiencies including **poor memory utilization**, **excessive rollbacks** and **communication overheads.**

Simulations that are susceptible to over-optimism include those operating in **heterogeneous environments**, those subject to **external workloads**, and those for which **application-specific characteristics** of the simulation promote over-optimistic behavior.

Networks of workstations (NOWs), an important platform for large scale simulations, are typically heterogeneous, with computing and memory resources varying among machines. Logical processes running on fast processors may progress faster in simulated time than logical processes running on slower processors.

A large scale simulation may run on a system that is shared among many users. Here, logical processes may compete with other applications for shared resources, causing some logical processes to run on more heavily loaded processors, while others run on less loaded processors. Logical processes running on heavily loaded processors make less progress in simulation time compared to logical processes on less loaded processors.

The manner in which a particular application is implemented can also influence over-optimistic behavior. Applications that exhibit *self-instantiation* and *uneven granularity of load* per LP may demonstrate over-optimistic behavior. *Self-instantiation* means that an LP schedules events to itself rather than to a remote LP. *Degree of self-instantiation* refers to the number of messages an LP sends to itself before sending a remote message. Applications that consist mainly of LPs with a high degree of self-instantiation communicate with other LPs infrequently. Because of this infrequent communication, whenever an LP that is far behind sends a message to an LP that is far ahead, it causes long rollbacks due to out-of-order messages. For example, the implementation of a Personal Communication Systems (PCS), described in (Carothers and Fujimoto 1994), which includes LPs having a high degree of self-instantiation, has been shown to exhibit over-optimistic behavior.

Another characteristic that may cause over-optimistic behavior is uneven granularity of load per LP in an application. This happens when some LPs incorporate more work and take more time to process their event set than the other LPs. Again, when the LPs that are far behind in the simulation communicate with LPs that are far ahead, long rollbacks and anti-messages result. Implementations of Asynchrous Transfer Networks (ATMs) are prone to such an uneven granularity of load per LP (Hao, et al. 1996).

A goal of our approach is to use the CPU as a flow control mechanism for over-optimistic execution. We present a process migration scheme that controls over-optimistic processes by isolating their impact on other processes, while promoting the progress of slower, less optimistic processes. We evaluate our approach using a synthetic benchmark application called *P-Hold*.

2 BACKGROUND

Prior work addressing the problem of over-optimism falls into three broad categories: protocols using limited optimism, memory management protocols and adaptive techniques.

2.1 Optimism Limiting Protocols

Blocking is a commonly used technique for reducing the amount of rollback(Reiher et al. 1989) that limits the progress of over-optimistic LPs through use of a time window of size W. LPs are prevented from progressing beyond GVT + W, and are blocked until LPs that are far behind catch up. Window size may be determined statically or dynamically.

The aggressive no-risk protocol(Dickens and Reynolds 1990) avoids sending a message until it is guaranteed that the message will not cause rollbacks. Messages sent by an LP are stored in its PE's buffer and not sent until GVT advances beyond the send timestamp of the message, assuring that the messages will not be rolled back later.

The *look ahead information* approach (Lubachevsky et al. 1989), also may be used to decide whether it is safe to process a given message. A hybrid conservative and optimistic protocol is employed that begins with the conservative protocol to determine which events are safe to process and later adds optimistic synchronization features to "unsafe" events.

Another approach introduces *additional rollbacks* at stochastically selected intervals (Madisetti et al. 1983). These additional rollbacks prevent overly optimistic execution of LPs that could be rolled back to GVT if a rollback decision were determined for that LP. Probability vectors are used to determine if the LP should be rolled back.

The *breathing time protocol*(Steinman 1983) limits the number of events a particular LP can process beyond GVT and involves determination of the minimum time stamp among events that will be produced in the future.

2.2 Memory Management Protocols

Two protocols used to limit memory utilization in an over-optimistic simulation are *artificial rollback* (Jefferson 1990) and *cancel back* (Lin and Preiss 1991). These protocols are used when the system runs out of memory and fos-

sil collection attempts cannot reclaim the memory needed for the simulation to progress. These schemes roll back some of the logical processes and utilize the freed memory to continue. *Artificial rollback* works by identifying the most over-optimistic LPs (those furthest ahead in simulation time) and then rolling them back. *Cancel back,* on the other hand, achieves the same effect by sending back certain messages to the sender LP, rolling back the sender.

2.3 Adaptive Techniques

S. Das and R. Fujimoto proposed an adaptive technique combining memory management and limited optimism synchronization protocols(Das and Fujimoto 1997). The amount of memory allocated to a Time Warp simulation automatically limits the amount of optimistic execution, i.e., the degree to which processes may advance ahead of other processes. This protocol seeks to provide sufficient memory for Time Warp to execute efficiently, but not so much memory that overly optimistic execution can occur. The protocol attempts to simultaneously address rollback thrashing and memory management issues. The approach is adaptive; it monitors the execution of the Time Warp program and automatically adjusts the memory provided to the parallel simulator. An adaptive protocol was necessary because the the synchronization and memory management protocol parameters depend on characteristics of the application such as symmetry and homogeneity among the simulation processes and memory required to execute the program using Time Warp.

A load distribution system for background execution of Time Warp(Carothers and Fujimoto 2000) is designed to use free cycles of a collection of heterogeneous machines to run a Time Warp simulation. The load management policy involves both processor allocation and load balancing. The processor allocation policy dynamically determines the set of processors to be used for a Time Warp simulation. LPs are grouped into clusters by the application to reduce migration overhead. Clusters, rather than individual LPs are migrated, with the goal of equalizing the progress of all the processors, taking into consideration the external and internal workload, processor speeds, etc. The metrics for classifying the processors and individual clusters are PAT (Processor Advance Time) and CAT (Cluster Allocation Time), respectively.

A load balancing technique for the Time Warp distributed system for object-oriented simulation(Burdorf and Marti 1993) distributes objects across nodes and provides optimistic concurrency control. The scheme consists of static and dynamic load balancing monitors. The static monitor determines pre-assignment of objects to processors. The dynamic load balancing module monitors load imbalance and initiates migration of objects, using knowledge of simulation time (LVT) to reduce rollback. That is, it minimizes the distance between the simulation time of the farthest ahead object and the furthest behind object.

Another interesting approach that applies load balancing and optimism limiting protocol (Jones and Das 1998) combines the throttling of over-optimistic processes and scheduling (or load balancing) to control over-optimistic behavior. Throttling is implemented by a *moving time window* protocol. In the scheduling component LPs are remapped to processors so the N slowest LPs in simulation time are mapped to different processors, where N is the number of processors on which the simulation is run.

2.4 Comparison

S. Das's adaptive memory management technique that uses memory as a flow control for controlling over-optimism is similar to our approach, which uses CPU as a flow control mechanism. Unlike Carothers and Fujimoto's approach, which uses available CPU cycles to load balance logical processes, we use the CPU itself to control over-optimism. By aggregating over-optimistic LPs to one CPU, we force these LPs to compete with each other for available CPU cycles. This slows down their progress while isolating their impact on other LPs. In addition, we spread out the less optimistic LPs, limiting their competiton for CPU cycles.

S. Das and K. Jones approach of using throttling with scheduling is similar to our approach in terms of making the less optimistic LPs progress to catch up with LPs that are ahead in simulated future. But for LPs that are far ahead in simulation time, Das and Jones use blocking, which wastes CPU cycles, unlike our approach which isolates these LPs on one PE to slow their progress. Too much throttling is harmful as too few events are admitted for processing. Another important difference is that their approach was implemented on a simulated distributed system, whereas our implementation is deployed on a real distributed system on real processors.

3 OUR APPROACH

The idea behind our approach is to aggregate fast LPs onto one processor so that they must compete for processor cycles, slowing their progress. Slow logical processes are dispersed to different processors to limit their competition for CPU cycles. Migration costs are minimized by reducing the number of LPs moved. This is done by selecting a *fast repository* (CPU on which the fast LPs will be aggregated) that already has the most fast LPs mapped to it. The cost of moves are justified in our approach in that we move only LPs that are either too fast and waste work, or too slow and likely require more resources.

Our approach to controlling over-optimism involves the following steps:

(i) Ranking of LPs: LPs are ranked according to how far ahead in simulation time they are compared to the rest

of the simulation. We use a metric called *GVTLag*, the difference between an LP's current simulation time and GVT (GVTLag = LVT- GVT). LPs are ranked according to their GVTLag value.

***(ii) Classification of LPs**:* LPs are then classified as one of *FAST*, *MEDIUM* and *SLOW*. The k * N LPs that have the highest value of GVTLag are selected to be labeled as FAST, where k is a scaling factor and N is the number of processors on which the GTW kernel is currently running. The factor k is an experimental parameter and depends on the migration cost in the system. A selected number of the slowest LPs are classified as SLOW; the remaining LPs are classified as MEDIUM. In our experiments, the number of SLOW LPs is fixed at N-1.

***(iii) Identification of the FAST-REPOSITORY**:* The FAST-REPOSITORY is the CPU on which the over-optimistic (FAST) LPs will be aggregated. The CPU containing the greatest number of LPs labeled as FAST is selected as the FAST-REPOSITORY. This reduces the number of LPs that will be moved in subsequent steps.

(iv) Isolation of FAST LPs: All FAST LPs not currently mapped to the FAST-REPOSITORY are migrated to it. This is done to isolate the effects of the over-optimistic LPs from the rest of the LPs in the simulation and to force these optimistic LPs to compete for CPU cycles.

(v) *Spreading of SLOW LPs:* SLOW LPs are redistributed to the processors other than the FAST-REPOSITORY, one per processor. The goal of this step is to limit the contention of the least optimistic LPs for CPU cycles, in the hopes of allowing them to "catch up" with the rest of the simulation.

3.1 Implementation

Our load balancing algorithm is implemented on the distributed Georgia Tech Time Warp system (GTW) (Das et al. 1994), which is a parallel and distributed discrete event simulation executive based on Jefferson's Time Warp (Jefferson 1985). GTW runs on both shared memory and distributed memory machines. Details of GTW can be found in (Fujimoto et al. 1997).

Distributed GTW employs an additional thread on each machine(Carothers and Fujimoto 2000) that handles all the external communication with other machines. The Parallel Virtual Machine (PVM) communication library is used for remote communication. We implemented our process migration algorithm in a separate thread called MonitorOPT on top of distributed GTW.

The software architecture of distributed GTW, including our thread, is shown in Figure1. GTW provides the APIs for the simulation application to exchange information with GTW regarding the number of LPs, number of processors, event handlers for each LP and other information. Once GTW has the application-specific information it sets up various data structures to carry out the execution of the simulation. Distributed GTW consist of two libraries: the *kernel library* and the *kernel communication library.* The kernel library consists of the core functionalities of GTW, including the state saving mechanism, scheduler of events, mechanism for computing GVT and communication thread. The kernel communication library consists of various methods that invoke PVM calls. The kernel library calls the method in the communication library in order to execute PVM functionality.

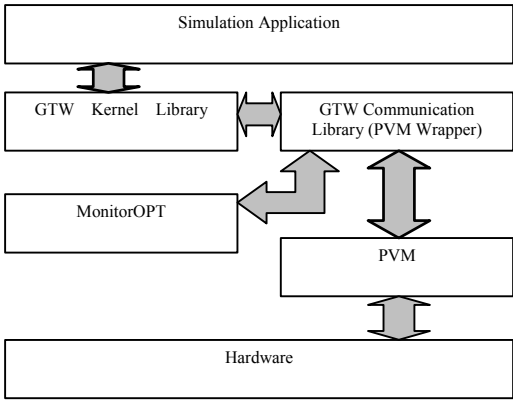

Figure 1: A Software Architecture for Distributed GTW

We implemented an optimism controlling module consisting of a central monitoring process called "*MonitorOPT*" that is heart of our algorithm. The process runs on a dedicated machine. MonitorOPT executes periodically to collect statistics from other processing elements (PEs). The specific period is an experimental parameter and could be varied. MonitorOPT computes the moves of the LPs based on the collected statistics of the whole system and uses this for controlling over-optimism of the LPs. The MonitorOPT process responsible for migration decisions communicates with the kernel communication library in order to exchange messages with the communication thread of the kernel library.

Once the new mapping of LPs to PEs is computed, a move-list is generated, containing information about which LPs to move, and their source and destination PEs. If MonitorOPT determines that some moves are to be made, a *HALT message* is sent to each processor. This is done to stop each PE's computation and roll back all LPs to GVT. This helps to synchronize all PEs and to perform load redistribution effectively, reducing the cost of LP migration as the history information of the LPs will not be moved from source processor to the destination.

After all the processors roll back their computations to GVT, they send an acknowledgment message, *HALT-ACK,* to the MonitorOPT process. The MonitorOPT process then sends a *MOVE message* for each move in the move-list to the source processor from which an LP will be moved. The source PE transfers all the information regarding the

LP to the destination PE, and then sends a *MOVE-LP* message to the destination PE. The destination PE updates all the necessary information and then sends a *MOVE-LP-ACK* message to the source PE, after which the source PE sends a *MOVE-ACK* to the MonitorOPT.

Once the MOVE messages have been sent for each move in the move-list, the MonitorOPT process computes an updated mapping of all LPs to PEs. It then sends a *MAPPING message* to all the processors to reflect this new mapping. Once all PEs update their local mapping structures for the mapping message they send a *MAPPING-ACK message* to MonitorOPTLB. Next, a *RESUME message* is sent so that the normal computation can resume on all processors. The load balancing interval starts with this message and at the interval expiry new computations and data collection can begin. This process continues until the end of the simulation.

4 EXPERIMENTS AND DISCUSSION

Experiments were performed to compare the performance of our approach with that of the standard Time Warp simulation without process migration. The experiments presented in this study were performed in a heterogeneous environment consisting of 8 machines: one SGI Origin 2000 with sixteen 195 MHZ MIPS R 10,000 processors running IRIX version 6.5 and seven 167-MHZ Sun Sparc Ultra -1 workstations running version 2.5 of the Solaris operating system. In all experiments the GTW kernel was executed on a total of 8 processors with one processor from each of the machines involved in the experiment. MonitorOPT was executed on the SGI machine. The benchmark application used to perform these experiments is P-Hold.

4.1 The Benchmark Application: P-Hold

P-Hold is a benchmark application using a synthetic workload model(Fujimoto 1990). The benchmark uses a fixed message population. Processing of each message takes a finite amount of time, after which a new message is sent to another LP with a specified time stamp increment. The initial messages have a timestamp that is exponentially distributed between 0 and 1. As the messages are forwarded their new timestamp increments are fixed at 1. The total number of LPs for these experiments was fixed at 256. These LPs were initially evenly distributed on 8 processors, with 32 LPs per processor. The experiments used a fixed message population of 6,400.

In order to effect an imbalance, P-Hold was instrumented with two distinct synthetic workloads: null and one millisecond. In the null case, event processing is made as small as possible; in the one millisecond case event processing includes a 1 msec delay loop.

To make the application more imbalanced, P-Hold was also configured as self-instantiated, and the degree of self-instantiation of an event was varied. The experiments used two degrees of self-instantiation: 50 and 200. When an event is processed in which the source LP differs from the destination LP, the destination LP schedules the next *d* generations of the event to itself. After *d* generations of the event have been produced, a new destination LP is randomly selected.

For the first set of experiments, two classes of LP/event restrictions are generated:

Class [200: 1 msec]: This includes events with degree of self-instantiation 200 and LPs that take a 1 millisecond delay (event granularity of 1 msec) to execute every event.

Class [50: null]: This include events with degree of self-instantiation 50 and LPs that execute events without any delay (null event granularity).

The percentage of LPs in Class [200:1msec] was varied from 0 to 100 percent in increments of 20 percent. The remaining LPs in the simulation operated under Class[50 : null]. By varying the percentage of LPs executing in either of the two classes, we created different levels of optimism and imbalance. LPs under Class [200: 1msec] progress slowly as their event granularity is larger than event granularity of Class [50: null]. Additionally, whenever the LPs in Class [200:1msec] send remote messages to LPs in Class [50:null], it may roll back the computation of the Class [50:null] LPs.

A second set of experiments was performed for comparison. While keeping all other operating parameters the same, we swapped the delay of event execution on the two classes of LPs. LPs under these experiments were made to execute under the two classes of operating restrictions.

Class [50: 1msec]: Events with degree of self-instantiation 50 and LPs execute event with 1 millisecond delay (1 msec event granularity).

Class [200: null]: Events with degree of self-instantiation 200 and LPs execute events without any delay (null event granularity).

An empirically determined sample period of 50 seconds was used. Results represent an average of four trials.

4.2 Experiments under Set 1

In Figure 2 below, we show the performance of three variations of our migration algorithm: (1) with fast repository and slow LP spreading, (2) without fast repository but with slow LP spreading, (3) with fast repository but without slow LP spreading. We measure performance as the ratio of the useful work with process migration to the useful work without process migration. Useful work is the ratio of events that were committed (not rolled back) to all events processed. The three variations are described below:

Both Fast Repository and Slow LP spreading*:* The number of FAST LPs is N * k, where k = 5 and N = 8. The numbers of LPs classified as SLOW is fixed to 7 (N-1, where N is the number of processors in the system).

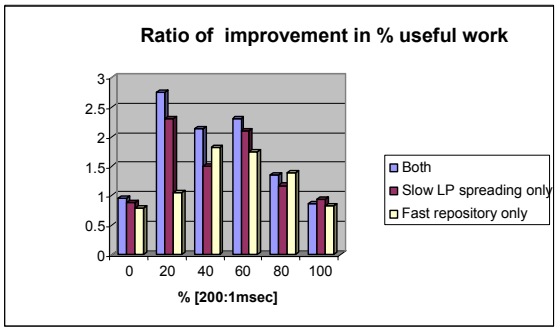

Figure 2: Ratio of Percentage Improvement in Useful Work as Compared to without Process Migration, (Set 1)

***Slow LP spreading only*:** The number of FAST LPs is 0; accomplished by setting k =0. We classify 8 (N=8) LPs as SLOW and the remaining LPs as MEDIUM.

***Fast Repository only*:** 40 LPs (k=5) are classified as FAST. The remaining LPs are classified as MEDIUM.

From Figure 2 we see that in cases where over-optimism exists the simulation performs better with process migration ("Both") than without process migration in terms of percentage of useful work, by a factor of 1.25 to 2.75 over useful work without process migration. The best performance is at data point 20%, where "Both" (Fast repository and slow spreading) performs approximately 2.75 times better than the simulation without process migration. None of the variations of process migration perform better than the simulation without process migration at data points 0 and 100. At these points the simulation is balanced; all LPs execute under one class of event granularity, so it is likely that it is not over-optimistic. At data point 0, all the LPs execute under Class [50 : null], whereas at data point 100 all LPs operate under Class [200:1 msec]. The LPs all have the same event granularity and progress at the same pace, causing less rollback. The ratio of useful work of less than one in these cases in which over-optimism is absent is a result of the overhead of process migration.

When the application is unbalanced, process migration(Both) demonstrates maximum benefits performing by a factor of 1.25 to 2.75 times better than the results without process migration. When the application is overly optimistic, with no migration, the simulation spends much of its time rolling back its computation rather than progressing forward and hence results in less useful work. On the other hand, due to process migration, the over-optimistic LPs are isolated on a separate processor slowing their progress down and less optimistic LPs are allowed to make progress by redistributing them on different processors. This helps in reducing the differences among the progress of the LPs and the number of rollbacks.

The two variations of process migration also perform better than no process migration when the application is unbalanced (data points 20% – 80%). In some cases the fast repository (which limits fast LPs) produces the benefit, while in other cases the spreading of slow LPs (which promotes progress by the slow LPs) produces the benefit. It depends upon the application state which type of LPs dominates the simulation. To complement each of these two approaches, we combined the two variations in our implementation of the process migration algorithm. In this combined approach, we control both the less optimistic and over-optimistic LPs, consistently providing better performance in cases where over-optimism exists.

Figure 3 shows the execution time performance for the above experiment, comparing the simulation run with process migration and its various variations against the simulation without process migration. It was observed that process migration did not provide any benefit in terms of execution time when the optimism is balanced(data point 0 and 100 when all LPs have same event granularity).The most benefit in terms of execution time is obtained at 20% and 40% of Class [200:1msec]. At these two points there is also a better percentage of useful work, as shown in Figure 2. For data points 60 and 80 it is observed that process migration takes almost equal execution time compared to without process migration. The percentage of useful work performed by the process migration run for data point 60 is approximately 2 times better than without process migration and for data point 80 is 1.15 times better than without process migration. That is, although we do not benefit in terms of execution time at these two data points, we do benefit in terms of useful work. This is the result of migration overhead. If this overhead consists largely of communication costs, then it may be possible for external workloads to benefit from this reduced work, as the CPU cycles not used in performing wasted work would be available. However, we have yet to characterize the computation/communication trade-offs of process migration, and plan to do so in future work. For point 100, without process migration(None) performs better for both the percentage of useful work in Figure 2 and the execution time in Figure 3. This is due to the application being balanced at this instant as all LPs belong to one class of self- instantiation and event granularity and thus progress at the same pace. The overheads of process migration exceed the benefits obtained when the application is balanced and least optimistic.

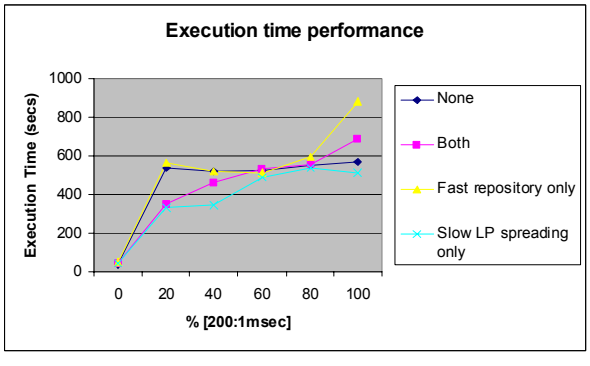

Figure 3: Execution Time Performance (Set 1)

Figure 4 shows the results of varying the value of k, used to determine the number of LPs to be classified as FAST. This experiment is performed for the data point 40 of the previous experiment (60% [200:1], 40% [50:0]) It is observed that as the number of fast LPs is increased, the percentage of useful work also improves. However, if k is large migration cost will dominate.

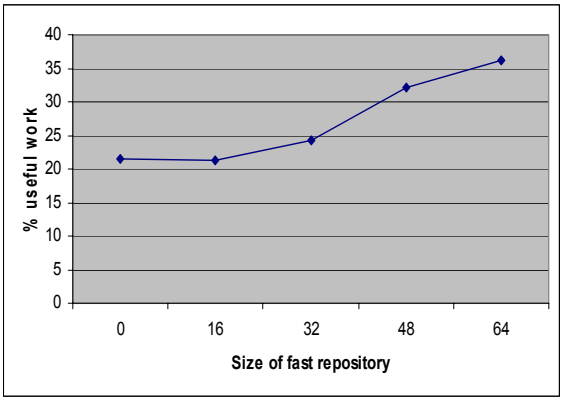

Figure 4: FAST LPs Scaling Performance

4.3 Experiments under Set 2

Another set of experiments was performed, in which the self-instantiation degree of messages and the delays of LP execution were swapped in the two classes of operating restrictions to create imbalance in the application. With all the other experimental parameters being the same as set 1, but instead of varying the messages with self-instantiation of 200 with LPs having event granularity one millisecond as in previous experiments, we combined messages with self-instantiation 50 with LPs having event granularity of 1millsecond. Again in this set of experiments we vary the percentage Class [50: 1msec] from 0 to 100 in increments of 20 percent. The remaining LPs in the system execute with null event granularity (Class [200: null]).

Figure 5 shows the performance in terms of percentage of useful computation for simulation runs with and without process migration. Each of the two scenarios were plotted to show the performance of the simulations with various levels of optimism created by varying LPs with different event granularity. As shown in Figure 5, it was observed that process migration performs better than without it at all the levels except at data point 100. This is because the simulation application is balanced at data point 100. At data point 100, all the LPs had one millisecond event granularity. This caused all LPs to progress at the same rate. In this scenario the overhead of process migration exceeded the benefits obtained by it and hence, the simulation without process migration performed better. In all the other instances, when the simulation is unbalanced, simulation with process migration performs 1.2 to 1.79 times better than without process migration.

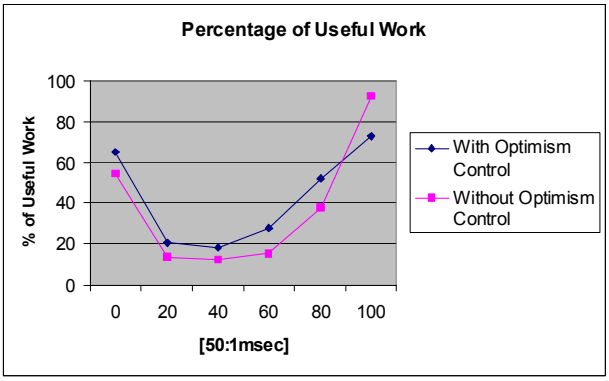

Figure 5: Percentage of Useful Work (Set 2)

In Figure 6, we compare the execution time performance for the above experiment with simulation runs with and without process migration. As mentioned above, due to the simulation application being balanced at data point 0 and 100 as all the messages and LPs belong to one of the two classes of self-instantiation and event granularity, we do not observe any benefits in execution time. At data point 0, the simulation without process migration performs better that that with process migration both in terms of execution time and percentage of useful work. Whereas at data point 100, we obtain benefits in terms of percentage of useful work but not in terms of execution time. A similar scenario exists at point 80, where we obtain the benefits in terms of useful work but not in execution time. For the remaining proportions of Class [50:1msec], benefits in both execution time and computation performance were obtained when the simulation was run with process migration as compared to without process migration. When the application is unbalanced, the improvements in execution time by doing process migration were approximately 10 to 20 percent.

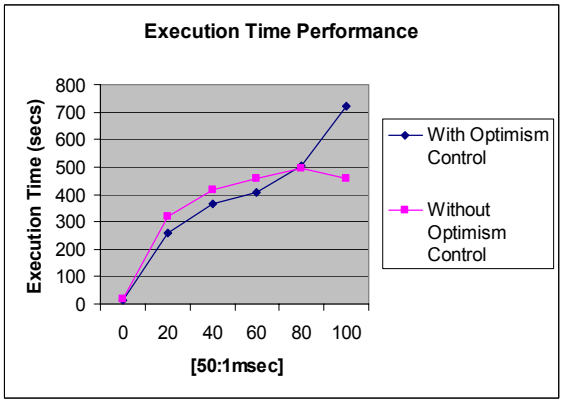

Figure 6: Execution Time Performance (Set 2)

We observed that our algorithm performs better in terms of useful work in all the experiments except when the application is balanced. We observed that the migration cost of LPs do incur a penalty in terms of execution time in some

cases. One of the reasons for this penalty is that the PVM communication is expensive. Due to resource limitations, the experiments were performed on older hardware. By updating the hardware we might reduce this migration cost to some extent. Also it was observed that a higher value of k improves the performance in terms of useful work, but may incur higher migration cost in terms of execution time. Hence, an appropriate value of k should be selected as a tradeoff between useful work and migration cost.

5 CONCLUSIONS AND FUTURE WORK

In a standard optimistic parallel event simulation, no restriction exists on how far an LP can proceed ahead of other LPs. This imbalance may degrade performance as LPs may spend more time rolling back than progressing forward. In order to control such over-optimistic behavior, we implemented a process migration system on distributed Georgia Tech Time Warp (GTW). The process migration system classifies LPs with respect to optimism as FAST, MEDIUM and SLOW depending on how far each LP is making progress in simulation time compared to other LPs. The statistic collected to classify an LP is called *GVTLag*, the difference between an LP's current simulation time and the system's GVT. We aggregate the FAST LPs on one processor, the FAST-REPOSITORY, to make them compete with each other for available CPU cycles to slow their progress and consequently isolate their impact on other logical processes. At the same time we spread the SLOW LPs, by redistributing each SLOW LP to a different processor. This provides SLOW LPs with sufficient CPU cycles to make progress and help them to catch up with the FAST LPs.

We performed various experiments with a synthetic benchmark application called *P-Hold*. We implemented P-Hold using two computation event granularities: null and 1msec. Additionally, P-Hold was configured to be self-instantiating. In order to create imbalance in the application LPs and events were classified into two classes having a self-instantiation degree of either 50 or 200 and with or without a 1 msec delay in event processing.

It was observed that whenever the application is unbalanced, the simulation with process migration performs 1.25 to 2.75 times better in terms of useful computation than that without process migration. Also, the execution time under unbalanced application conditions is either better than or equivalent to the time taken without process migration.

On the other hand, when the application is balanced and over-optimism is minimal, simulations without process migration perform better in terms of useful work and execution time. This results from the overheads of process migration exceeding the benefits obtained when the application is balanced.

Due to high memory requirements and resource limitations, we could not experiment with more than 256 LPs in the simulation. We propose to experiment with more than 256 LPs in the future and observe how the process migration performs when the number of logical processes increases in the simulation. Also, we propose to modify the process migration algorithm to make the process of selecting the number of fast LPs dynamic and dependent on how the simulation is performing. This could help in improving the amount of useful work done and execution time compared to the simulation without process migration.

A problem our approach may encounter in extreme cases is that logical processes might aggregate on the same processors, leaving other processors sparsely populated. For example, we may encounter a situation in which a million LPs are on one processor and one logical process is on each of the remaining processors. Another problem could be the movement of logical processes that are already balanced. One of the possible solutions to alleviate such scenarios is to use a threshold to determine the number of processes classified as fast.

We also propose to cluster LPs in a manner similar to (Carothers and Fujimoto 2000), in order to reduce the process migration cost. This would require a different metric for evaluating process migration. We propose to evaluate such a possibility and observe its performance compared to without process migration.

REFERENCES

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

Carothers, C. and R. Fujimoto. 1994. Distributed Simulation of Large-Scale PCS Networks. In *Proceedings of the Second International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems(MASCOT'94),* 2-6.

Carothers, C.D. and R. M. Fujimoto. 2000. Efficient Execution of Time Warp Programs on Heterogeneous, NOW Platforms. *IEEE Transactions on Parallel and Distributed Systems*, 11(3):299-317.

Das, S., R. Fujimoto, K. Panesar, D. Allison and M. Hybinette. 1994. GTW: A Time Warp System for Shared Memory Multiprocessors, In *Proceedings of the 1994 Winter Simulation Conference*, ed. D.A. Sadowski, A.F. Seila, M.S. Manivannan, and J.D. Tew,332-339. San Diego, California: Society for Computer Simulation International.

Das S. and R. M. Fujimoto. 1997. Adaptive Memory Management and Optimism Control in Time Warp, *ACM Transactions on Modeling and Computer Simulation (TOMACS),* 7(2):239-271.

Dickens, P.M. and P. F. Reynolds, Jr. 1990. SRADS with local rollback. In *Proceeding of the SCS Multiconference on Distributed Simulation*, 22:161-164.

Fujimoto, R. M. 1990. Performance of Time Warp under Synthetic Workloads, In *Proceedings of the SCS Multiconference on Distributed Simulation*, 22: 23-28.

Fujimoto, R. M. 2000. *Parallel and Distributed Simulation Systems*, Wiley Series on Parallel and Distributed Computing, 137 – 138.

Fujimoto, R.M., S. R. Das, K.S. Panesar, M. Hybinette and C. Carothers, 1997. Georgia Tech. Time Warp Programmer's Manual for Distributed Network of Workstations, Technical Report GIT-CC97-18, College of Computing, Georgia Inst. of Technology, Atlanta, GA.

Hao, F., K. Wilson, R. Fujimoto and E. Zegura. 1996. Logical Process Size in Parallel Simulations. In *Proceeding of the 1996 Winter Simulation Conference*, ed. J. Charnes, D. Morrice, D. Brunner, and J. Swain, 645-652. New York, New York: ACM Press.

Jefferson, D.R. 1985. Virtual Time, *ACM Transactions on Programming Languages and Systems*, 7(3):404-425.

Jefferson, D. R. 1990. Virtual Time II: The Cancel Back Protocol for Storage Management in Distributed Simulation. In *Proceedings of the 9th Annual ACM Symposium on Principles of Distributed Computing*, 75-90.

Jefferson, D. R., and H. Sowizral. 1985. Fast Concurrent Simulation Using the Time Warp Mechanism, Part – I : Local Control, *ACM Transactions Programming Languages and Systems*, 7(3): 404-425.

Jones K. and S. R. Das. 1998. Combining Optimism Limiting Schemes in Time Warp based Parallel Simulations, In *Proceedings of the 1998 Winter Simulation Conference*, ed. D.J. Medeiros, E.F. Watson, J.S. Carson and M.S. Manivannan, 499-505. Los Alamitos, California: IEEE Computer Society Press.

Lin, Y.-B., and B. R. Preiss. 1991. Optimal Memory Management for Time Warp Parallel Simulation, *ACM Transactions on Modeling and Computer Simulation*, 283 – 307.

Lubachevsky, B.D., A. Shwartz and A. Weiss. 1989. Rollback Sometimes Works…if Filtered. In *Proceedings of the 1989 Winter Simulation Conference*, ed. E.A. MacNair, K.J. Musselman, and P. Heidelberger, 630-639. New York, New York: ACM Press.

Madisetti V. K, D.A. Hardaker, and R. M. Fujimoto. 1993. The MIMIDIX Operating System for Parallel Simulation and Supercomputing, *J. Parallel and Distributed Computing*, 18(4): 473-483.

Reiher, P.L. F. Wieland, and D. R. Jefferson. 1989. Limitation of Optimism in the Time Warp Operating System. In *Proceedings of the 1989 Winter Simulation Conference*, ed. E.A. MacNair, K.J. Musselman, and P. Heidelberger, 765-770. New York, New York: ACM Press.

Steinman, J. S. 1993. Breathing Time Warp, In *Proceedings of the Seventh Workshop on Parallel and Distributed Simulation*, 23:109-118.

AUTHOR BIOGRAPHIES

VINAY SACHDEV received the MS in Computer Science from the University of Georgia in the spring of 2004.

MARIA HYBINETTE is an Assistant Professor of Computer Science at the University of Georgia. She earned the Ph.D. (2000) and M.S. (1994) in Computer Science from the Georgia Institute of Technology and a B.S.(1991) in Math and Computer Science from Emory University, all in Atlanta, Georgia.

EILEEN KRAEMER is an Associate Professor of Computer Science at the University of Georgia. She earned the Ph.D. in Computer Science from the Georgia Institute of Technology in 1995, an M.S. in Computer Science from the Polytechnic University in Brooklyn, NY in 1986, and a B.S. in Biology Hofstra University in Hempstead, NY in 1980.