OVERCOMING OVER-OPTIMISM IN TIME WARP VIA AGGREGATION OF FAST PROCESSES

by

VINAY S. SACHDEV

(Under the Direction of Maria Hybinette)

ABSTRACT

The parallel Time Warp protocol is widely used to improve the performance of large-scale discrete event simulations such as simulations of the air traffic control system and the World Wide Web. However, a concern regarding optimistic simulation processing is that some logical processes (LPs) may progress far beyond others into the simulated future, causing an imbalance that may degrade performance. The degradation is due to long and excessive rollbacks, inefficient use of memory resources, and communication overheads. Allowing this "over-optimistic" processing may bring the simulation to a halt. This is unacceptable for long running simulations that may take days to complete. We present a new mechanism that controls over-optimistic processing through aggregation and isolation of fast processes and redistribution of the slow, less optimistic processes. Our performance results demonstrate that our techniques can improve the useful work by a factor of 1.75 and also improve execution time, while keeping the overhead small.

INDEX WORDS: Parallel Discrete Event Simulation, Time Warp, Over-Optimistic LPs, Process Migration

OVERCOMING OVER-OPTIMISM IN TIME WARP VIA AGGREGATION OF FAST PROCESSES

by

VINAY S. SACHDEV

B.E, The University of Mumbai, India, 1999

A Thesis Submitted to the Graduate Faculty of The University of Georgia in Partial Fulfillment of the Requirements for the Degree

MASTER OF SCIENCE

ATHENS, GEORGIA

2004

© 2004

Vinay S. Sachdev

All Rights Reserved

OVERCOMING OVER-OPTIMISM IN TIME WARP VIA AGGREGATION OF FAST PROCESSES

by

VINAY S. SACHDEV

Major Professor: Maria Hybinette

Committee: Eileen Kraemer
Thiab Taha

Electronic Version Approved:

Maureen Grasso
Dean of the Graduate School
The University of Georgia
May 2004

DEDICATION

I would like to dedicate this thesis to my late grandfather Moolchand Keswani, late maternal grandfather Rohat Nihalani, grandmother Anjana Keswani, late father Suresh Keswani, mother Deepa Keswani, sister Shalini Sachdev, wife Chetna Warade, and relatives whose support, blessings and best wishes made it possible.

ACKNOWLEDGEMENTS

I would like to thank my co-advisors Dr. Maria Hybinette and Dr. Eileen Kraemer for their constant support, useful suggestions, encouragement and help during the thesis. I would also like to thank Dr. Thiab Taha for his valuable time and for being part of my thesis committee. I would also like to specially thank Chris Carothers for helping us with this work.

TABLE OF CONTENTS

Page

ACKNOWLEDGEMENTS v

LIST OF TABLES vii

LIST OF FIGURES viii

CHAPTER

1 INTRODUCTION 1

2 PROBLEM AND MOTIVATION 3

3 BACKGROUND AND RELATED WORK 8

4 APPROACH 15

4.1 HIGH LEVEL APPROACH 15

4.2 IMPLEMENTATION 21

5 EXPERIMENTS 26

5.1 BACKGROUND 26

5.2 PERFORMANCE RESULTS 30

6 CONCLUSION AND FUTURE WORK 38

REFERENCES 40

LIST OF TABLES

Page

Figure 5.1: Simulation state during n^{th} cycle of process migration 27

Figure 5.2: Simulation state during $n+1^{st}$ cycle of process migration 28

LIST OF FIGURES

Page

Figure 2.1 : Snapshot after 5 msec 5

Figure 2.2 : Snapshot after 10 msec 6

Figure 4.1 : Initial State of Simulation 17

Figure 4.2 : Ranking of LPs 17

Figure 4.3 : Classification of LPs 17

Figure 4.4 : Identification of FAST-REPOSITORY 18

Figure 4.5 : Moves during Isolation of FAST LPs 19

Figure 4.6 : After Isolation of FAST LPs 19

Figure 4.7 : Moves during spreading of SLOW LPs 19

Figure 4.8 : After spreading SLOW LPs 19

Figure 4.9 : Software architecture for Distributed GTW 21

Figure 4.10 : Pseudo-Code for MonitorOPT 24

Figure 5.1 : Ratio of percentage improvement in useful work (Set 1) 31

Figure 5.2 : Execution time performance (Set 1) 33

Figure 5.3 : FAST LPs scaling performance 34

Figure 5.4 : Percentage of Useful Work (Set 2) 36

Figure 5.5 : Execution Time Performance (Set 2) 37

CHAPTER 1

INTRODUCTION

Parallel simulation based on the optimistic Time Warp [1] protocol is widely used to improve the performance of large-scale simulations of air-traffic control and the World Wide Web. However, a primary concern of optimistic simulation processing is that some logical processes (LPs) may progress far beyond others into the simulated future causing an imbalance that may degrade performance. This imbalance is due to excessive and long rollbacks, inefficient use of memory resources and communication overheads. Allowing this "over-optimistic" processing becomes intolerable and brings simulation to a halt. This is unacceptable for long running simulations that may take days to complete. We present a process migration scheme that controls over-optimistic processes by isolating their impact on other processes, while guaranteeing the progress of slower, less optimistic processes.

Parallel and distributed discrete event simulation is a type of simulation that is run on multiple processors or machines. The main reason for running a simulation in parallel and distributed environments is reduced running time compared to sequential execution of the simulation.

Discrete event simulation is defined as a computer model of physical system in which the state of the system changes only at discrete points in simulated time. Parallel simulation models a physical system of interacting physical processes. The physical processes are mapped to logical processes (LPs) and interactions are modeled by the "scheduling of an event". The scheduling of an event is accomplished by sending a time-stamped message between LPs. In our discussion "messages" and "events" are used interchangeably. LPs communicate exclusively by passing time-stamped messages to advance the simulation.

Two main synchronization protocols exist for parallel simulations: *conservative* and *optimistic*. A conservative protocol maintains the causality of event execution by disallowing the processing of out of

order events. In the optimistic protocol out-of-order time stamp messages are allowed. Whenever an LP receives messages whose time-stamp is in the past of the LPs current *logical clock (LVT)*, the LP rolls back its execution to the time-stamp before that of the arrived messages. Such out-of-order messages are called *straggler messages*. To accommodate rollback each LP maintains a history of state information. When rollback is necessary an LP reverts to the appropriate previous state. In order to "un-schedule" a message that was sent before a rollback, LPs keep a copy of each message sent, in an output queue. These messages are called *anti-messages*. If the LP that scheduled a message is rolled back, it sends the corresponding anti-message to annihilate the original message. Anti-messages may cause additional rollbacks called secondary rollbacks.

A message is guaranteed not to rollback if its time-stamp is earlier than the smallest time stamp of any unprocessed or partially processed message in the system. This lower bound is called *Global Virtual Time (GVT)*. GVT is the minimum time beyond which LPs can not be rolled back. Once a new GVT value is computed, memory for events and their corresponding state and anti-messages with time stamps before GVT can be reclaimed. This reclaiming of memory resources is called *fossil collection*.

In this thesis we are focusing on the optimistic Time Warp protocol developed by Jefferson and Sowizral [1], which uses state saving and rollback to allow processes to execute asynchronously in a distributed environment.

In this thesis we propose algorithms that counteract over-optimistic behavior by dedicating a processor to logical processes that are far ahead of others in simulated time, while less optimistic processes are redistributed among the remaining processors. Consequently, these more optimistic logical processes compete among themselves for CPU cycles and memory resources, slowing their progress while isolating their impact from other processes. The redistribution of slower processes guarantees that they will progress and advance in simulated time. We evaluate our approach using a synthetic benchmark application called *P-Hold*. Our results were found to be favorable in controlling over-optimism and improving the percentage of useful work when the application is unbalanced. When the application is balanced, the scheme was found to have minimal overheads.

CHAPTER 2

PROBLEM AND MOTIVATION

Our research addresses the problems of over-optimistic logical processes. In this section we discuss environments that may cause over-optimistic behavior of logical processes and how this behavior impacts the performance. Environments that are susceptible to such disparity among processes in simulated time are discussed next. Three characteristics are considered, they include: heterogeneous environments, external workload and application characteristics.

Heterogeneous Environment: Networks of workstations (NOW) are an important platform for large scale simulations. NOW systems are typically heterogeneous, in which computing and memory resources vary between machines. Accordingly, logical processes running on fast processors may progress faster in simulated time than logical processes running on slower processors, compounding the gap between them.

External Workload: A large scale simulation may run on a system that is shared between many different 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. Due to these imbalances, logical processes running on heavily loaded processors make less progress in simulation time compared to logical processes on less loaded processors.

Applications Characteristics: The manner in which a particular application is implemented can 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 mainly consist of LPs with a high degree of self-instantiation communicate with other LPs infrequently. Because of this infrequent communication, LPs may become over-optimistic because whenever an LP that is far behind sends a message to an LP that is far ahead, it causes long roll backs due to out of order messages. The implementation of a Personal Communication Systems (PCS), described by Carothers et al. in [4], that include LPs having a high degree of self-instantiation, is shown to exhibit over-optimistic behavior. The PCS simulation is described below,

Higher degree of self-instantiation (PCS simulation): A personal communication service network is a wireless communication network, which provides communication services for its subscribers. The service area of the PCS network is divided into cells. The entire service area is populated with radio transmitters and receivers called radio ports. To each radio port a set of radio channels have been assigned. Users can call and receive phone calls using the services of these radio channels. When a user that initiated a phone call in a particular cell moves to a different cell, the PCS network tries to allocate a new radio channel from the new cell to keep the call ongoing. This process is called *hand-off*. In certain instances the network is unable to obtain a new radio channel for the new cell and hence causes call termination. Simulations of the PCS network are implemented in order to model the behavior of the system. It helps the systems engineer to study the model and tune the real system in such a manner that the probability of a call termination is very low.

The PCS simulation is implemented such that LPs have different degrees of "self-instantiation". It is shown that it suffers from over-optimistic behavior. The self-instantiating behavior of LP communication leads to less synchronization between LPs as they communicate infrequently. When these LPs do communicate, they tend to be out of phase with each other, which results in long rollbacks.

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. This causes the LPs doing more work to progress more slowly than the

other LPs. Whenever LPs that are far behind in the simulation communicate with LPs that are far ahead, it causes long roll backs and anti-messages in the system due to out of order messages. For example, consider a scenario described by Fujimoto [6], and depicted in fig.2.1 and 2.2, where LP_a takes 1 unit of time to execute one event, while LP_b takes 5 unit of time to execute one event. Consequently, LP_a will advance by 5 events in the time LP_b takes to advance one event. Whenever LP_b sends a message to LP_a, it causes LP_a to roll back its computation as the arriving message time stamp will be in its past. Implementations of Asynchrous Transfer Networks (ATMs) are prone to uneven granularity of load per LP [5].

Uneven granularity of load (ATM simulation): Modern high bandwidth networks must provide acceptable quality of service to an increasingly diverse set of applications and traffic sources that vary from very low speed data transfer to very high quality distribution applications, e.g. television distribution. An ATM is used to integrate these kinds of diverse applications with different speed and resources requirements. In ATM network simulations, LPs are used to model different components of the networks such as switches, routers and traffic elements.

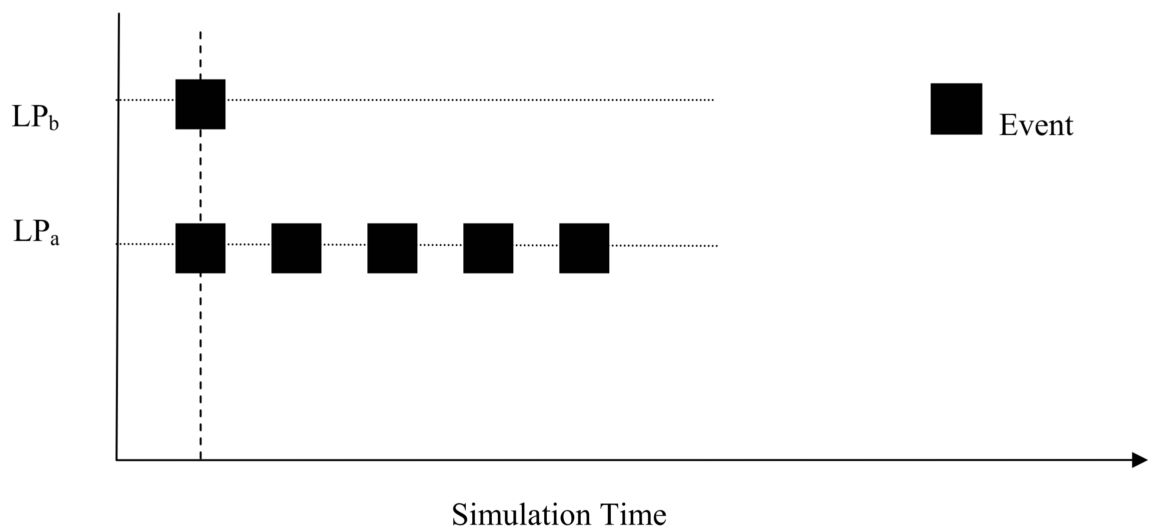

Fig. 2.1 Snapshot after 5 msec

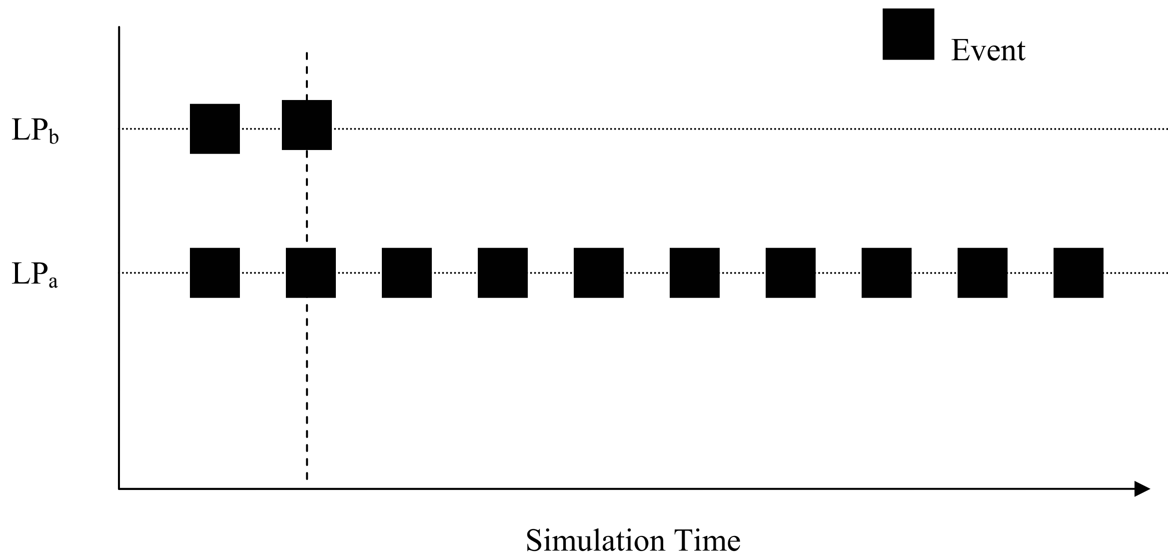

Fig. 2.2 Snapshot after 10 msec

ATM models are used by researchers to study large networks and collect statistics of real time operations. In order to incorporate these diverse elements in the simulation model, an important consideration is the size of an LP and which components should be included in a single process to maximize performance of the simulation. This may cause an imbalance because coarse-grained LPs may process more events in the same time that finer grained LPs process fewer events.

The previous section discussed characteristics that may cause over-optimistic behavior; we will now discuss *how* this behavior impacts the performance. Over-optimistic behavior of LPs can cause various inefficiencies; these include poor memory utilization, excessive rollbacks and communication overheads.

Poor memory utilization: Memory resources in a system are limited. A Time Warp application uses a fixed amount of memory to store the history of events, anti-messages and state information associated with LPs. Memory is reclaimed when GVT advances. Due to the over-optimistic behavior, some LPs may continually progress further ahead in simulation time than the other LPs, increasing the gap between them. Due to this behavior, much of state information needs to be kept in the system. The memory requirement may grow without bound and memory become exhausted if such over-optimism is not controlled.

Excessive and long rollbacks: Another inefficiency caused by over-optimistic behavior is excessive long rollbacks. As previously discussed, over-optimism may cause some LPs to proceed far ahead of others in terms of simulation time. When less optimistic LPs send a message to over-optimistic LPs, these cause long rollbacks. The processed events are rolled back and anti-messages are sent to cancel the events scheduled by the events currently being rolled back. This nullifies all the useful computation done by these over-optimistic LPs and the LPs to whom the anti-messages were sent. Useful CPU cycles are wasted by undoing the work already done.

Communication Overheads: As discussed above, the over-optimistic behavior of LPs can lead to long and excessive rollbacks. This result in a large number of anti-messages sent to cancel the events that were scheduled due to the processing of the event being rolled back. Anti-messages then utilize the bandwidth of the communication network which could have been otherwise used for other useful communication.

A goal of our heuristic is to use the CPU as a flow control mechanism for over-optimistic execution. We do this by dedicating one CPU to logical processes that are far ahead and make them compete for CPU cycles to slow them down.

CHAPTER 3

BACKGROUND AND RELATED WORK

Over-optimism causes two problems; one is excessive rollbacks and another is inefficient use of memory resources. In order to limit these problems and to control over-optimistic behavior, several techniques have been suggested. These fall into three broad categories: protocols using limited optimism, memory management protocols and adaptive techniques.

Optimism Limiting Protocols

There have been approaches suggested to limit over-optimism in order to reduce the amount of rollback that an application experiences during its execution. One of the commonly used techniques is called *blocking*. Blocking is based on limiting the progress of an over-optimistic LP beyond the advance of others in simulation time [7]. The idea is to hold back LPs that are far ahead in simulation time and to allow the lagging LPs to make progress and catch up. This is done by setting a time window beyond GVT, GVT + W, where W is the window size. LPs are not allowed to progress beyond GTW + W and are blocked until LPs that are far behind catch up. The window size can be determined statically at the start of the simulation or changed dynamically by monitoring the progress of the simulation.

Another approach avoids sending a message until it is guaranteed that it won't cause rollbacks. This is called the *aggressive no-risk protocol* [8]. The messages that are sent by an LP are stored in the buffer of the processor on which the sending LP is mapped and are not sent until GVT advances beyond the send timestamp of the message, assuring that the messages will not be rolled back later.

Look ahead information could also be used to assess whether processing a particular event is safe or not [9]. This could be done by a hybrid conservative and optimistic protocol, where one starts with

the conservative protocol to determine which events are safe to process and later adding the optimistic synchronization features to the events that are not safe.

One of the other approaches is to introduce *additional rollbacks* at stochastically selected intervals [10]. These additional rollbacks are added to prevent overly optimistic execution of certain LPs where an LP could be rolled back to GVT if a rollback decision is determined for that particular LP. This is controlled by probability vectors to determine if the LP should be rolled back or not.

The *breathing time protocol* [11] limits the number of events a particular LP can process beyond GVT. This is done by determining the minimum time stamp among events that will be produced in the future. This algorithm is operated in cycles, where in each cycle the parallel simulation will perform the following. First, whenever an LP sends a message, it is stored in the processor buffer on which the sender LP is currently mapped. Later each processor determines the minimum of the local time-stamps of all the collected messages in its buffer. Each processor then processes events locally as long as the event time-stamp is less than the minimum local time-stamp computed on that particular processor earlier. A global event time-stamp is then computed by collecting all of the local event time d stamps computed by each processor. All the messages that have a send time-stamp less than this global time stamp are then sent to their destination. This ensures that they won't be rolled back. The GVT is then advanced to this global time-stamp computed.

Memory Management Protocols

Two protocols used to limit the memory utilization in an over-optimistic simulation are *artificial rollback* [12] and *cancel back* [13]. These protocols are used when the system runs out of memory and fossil collection attempts cannot reclaim any more memory, which is needed for the simulation to progress. These schemes roll back some of the logical processes and utilize the memory that is freed by the state vector information that was saved by these LPs and also by the annihilation of the message/anti-message pairs.

Artificial rollback works by identifying the LPs that are furthest ahead in simulation time and then rolling them back. These LPs are the most over-optimistic in the simulation. *Cancel back,* on the other hand, achieves the same by sending back certain messages that have been received by an LP to the sender LP, which in turn will roll back the sender. This way memory can be freed and can be utilized to enable progress of the simulation.

Adaptive Techniques

S. Das and R. Fujimoto proposed an adaptive technique combining memory management and limited optimism synchronization protocols [28]. This is motivated by the fact that 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. A protocol was proposed that provides sufficient memory for Time Warp to execute efficiently, but does not provide so much memory that overly optimistic execution can occur. A mechanism was used to simultaneously address both the rollback thrashing and memory management issues. The approach is adaptive in that it monitors the execution of the Time Warp program, and automatically adjusts the amount of memory provided to the parallel simulator to maximize performance. An adaptive protocol was necessary because the appropriate parameters of both the synchronization and memory management protocols are dependent on characteristics of the application, e.g., the amount of symmetry and homogeneity among the simulation processes, as well as the minimum amount of memory required to execute the program using Time Warp. These characteristics may change dramatically during the course of a single simulation run. The synchronization and memory management protocols must adapt to these changing behaviors so that the Time Warp program can perform equally well in each one. The approach makes attempts that minimize total execution time rather than concentrating on one specific criterion, e.g., minimizing the amount of rolled back computation. This prevents the mechanism from optimizing for one aspect of the computation, e.g., minimizing rollback, at the expense of a disproportionate increase in another, e.g., memory reclamation (fossil collection) overheads.

The literature is rich in general dynamic load balancing algorithms, but only a handful of algorithms apply to parallel and discrete event simulation. In this section we describe the previous attempts at dynamic load balancing to control optimism in parallel discrete event simulation. The load balancing techniques described below could be considered as special cases of adaptive techniques.

C. D. Carothers and R. J. Fujimoto proposed a load distribution system for background execution of Time Warp [15]. The system is designed to use free cycles of a collection of heterogeneous machines to run a Time Warp simulation. Their load management policy consists of two key components: the processor allocation and load balancing. The processor allocation policy is used to determine the set of processors that can be used for a Time Warp simulation. This set is computed dynamically throughout the simulation. A processor is added or removed from the set when an estimate for the Time Warp execution time on that processor falls below a certain threshold. In this approach the LPs are grouped together in clusters by the application to reduce the moving overheads. Whenever a move is made it is as a cluster and all LPs that belong to a particular cluster will be moved together from one processor to another. The load balancing policy attempts to redistribute clusters among the processors in the system with the goal of equalizing the progress of all the processors, taking into consideration the external and internal workload, processor speeds, etc.

The central metrics for classifying the processors and individual clusters are called PAT (Processor Advance Time) and CAT (Cluster Allocation Time) respectively. These metrics are used in making various load balancing decisions. The load balancing policy moves clusters with a large PAT value to processors with a low PAT value. The goal is to minimize the maximum difference between the PAT values in any pair of processors. These metrics are defined as follows:

PAT (Processor advance time) : 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. Absence of rollback indicates only useful work that is committed is measured by ignoring the work that is rolled back.

CAT (Cluster allocation time) : Cluster Advance Time is defined as the amount of computation required to advance a cluster one unit of simulated time in the absence of rollback.

The load balancing algorithm attempts to minimize the maximum of $PAT_a - PAT_b$ for all values of a and b, where a and b are different processors. This algorithm executes after a specified interval of time and repeatedly moves clusters from the processor with largest PAT value. Clusters on this processor are scanned in the order of largest to smallest CAT value. For each such cluster the destination processors are scanned in the order from smallest to largest PAT value to redistribute this workload from a heavily loaded processor to a less loaded processor. If this move reduces the differences between the source and destination processor PAT values to some threshold, the move is accepted and this procedure is repeated. If the subsequent moves fail to reduce the differences in PAT values then the process is terminated and is started only at the start of the next load balancing interval.

C. Burdorf and J. Marti implemented load balancing techniques for the Time Warp distributed system for object-oriented simulation [20]. Their system distributes objects across nodes and provides optimistic concurrency control. The scheme consists of a static and dynamic load balancing monitors. The static monitor determines pre-assignment of objects to processors before the simulation begins execution. Once the assignments have been determined and the system starts execution, the static load balancer finishes its work and the dynamic load balancer takes over. The dynamic load balancing module monitors any unbalance in load. If the load is not balanced, it will initiate the migration of objects from one processor to another. The static balancer uses the bin-packing approach [21]. Initially the static balancer assigns objects to processors according to the load, which is gathered by running a presimulation or by data flow analyzer.[22,23]. During the simulation, the dynamic balancer records the smallest simulation time of all objects on each machine.

The dynamic load balancer uses simulation time (LVT) to reduce rollback. It assumes that by minimizing the distance between simulation time of the farthest ahead object and the furthest behind object, there will be fewer rollbacks, assuming these objects would interact with each other. In order to minimize the number of rollbacks, it tries to minimize the variance between each object's simulation

(LVT) times. The authors presented four schemes for load balancing. Three of them use Local Virtual Time (LVT) as a metric, and the fourth uses the ratio of total processed messages to the total number of rollbacks. Best results were achieved when objects which are furthest ahead are moved to machines which are furthest behind, and objects that are furthest behind are moved to machines which are furthest ahead.

Another interesting approach that combines load balancing and optimism limiting protocol is by S. Das and K. Jones [14]. They combined the throttling of over-optimistic processes and scheduling (or load balancing) to control over-optimistic behavior. *Throttling* is a mechanism similar to blocking where over-optimistic LPs are prevented from proceeding far ahead in the simulation time. Throttling is implemented by the *moving time window* protocol described above. In this approach a window is set beyond GVT which is used to block LPs that progress beyond this limit and let the LPs that are less optimistic and lagging behind to catch up. If the time window is set as Tw, then the LPs whose LVT is beyond (GVT + Tw) are not allowed to schedule any events for execution. Tw is a fixed parameter, determined experimentally. *Scheduling*, on the other hand is deciding how to map the LPs to different processors and where and when each event with a given time stamp should be processed. This involves migration of LPs to different processors periodically so to provide less optimistic LPs enough resources to make progress and catch up with the over-optimistic LPs. In this scheme 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. This will ensure that the slow LPs get enough CPU cycles to make progress.

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 use 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. Besides controlling over-optimistic LPs, we also spread the less optimistic LPs, by redistributing the LPs on the remaining PEs, guaranteeing sufficient CPU cycles to make progress. Also unlike other approaches, instead of allocating CPU cycles to a particular LP, we let the LPs decide among themselves how much CPU cycles they will need.

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, it uses blocking which waste 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 employed on a real distributed system on real processors.

CHAPTER 4

APPROACH

4.1 HIGH LEVEL APPROACH

The idea behind our approach is to aggregate fast logical processes on to one processor so that these fast LPs must compete for processor cycles, slowing their progress. On the other hand, slow logical processes are remapped to different processors to ensure that they get sufficient CPU cycles. Migration costs are minimized by moving few logical processes. This is done by selecting a fast repository (CPU selected to slow the progress of fast logical processes) that has the most fast LPs already mapped to it. The cost of moves are justified in our approach as the approach moves only LPs that are either too fast and have already wasted a lot of work, or too slow and likely require more resources.

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.

Our approach to controlling over-optimism involves the following steps: Ranking of LPs, classification of LPs, identification of a FAST-REPOSITORY, isolation of FAST LPs and spreading of SLOW LPs. We explain the meaning and intent of each of these steps in detail below:

(i) Ranking of LPs: The first step in our approach is to rank LPs in the simulation according to how far ahead in simulation time they are compared to the rest of the LPs in the simulation. We use a metric

called “GvtLag” to rank LPs in the simulation. GvtLag is the difference between an LPs current simulation time and the system’s GVT (GvtLag = LVT- GVT). LPs are ranked in descending order according to their GvtLag value. Fig.4.1 shows a hypothetical system consisting of four processors (A, B, C, D), each having five logical processes mapped to it. These logical processes are represented by the numbers inside these processors which represent the GvtLag metrics for these LPs. Fig 4.2 shows the state of the system at the end of step 1, where the numbers inside each processor represent the ranking of the corresponding LPs according to the GvtLag metrics. Here, the logical processes are ranked from 1 to 20, where 1 is the fastest.

***(ii) Classification of LPs**:* Once the LPs are ranked according to the GvtLag, the next step is to classify these LPs according to the classes: *FAST*, *MEDIUM* and *SLOW*. LPs classified as FAST are the LPs having the highest rank from step 1(the LPs further ahead), whereas LPs classified as SLOW are the ones with lowest rank (the slower LPs). LPs are identified as MEDIUM if they are not classified as FAST or SLOW. The number of FAST LPs is parameterized by our algorithm but the number of SLOW LPs is fixed according to the number of available processors. The parameter is experimental and depends on migration cost. Fig 4.3, shows the state of the simulation at the end of step 2 showing the classification of LPs in these three groups. In this case we select 7 LPs as FAST and 3 as SLOW.

***(iii) Identification of the FAST-REPOSITORY**:* One of the highlights of our approach is the use of a CPU to slow down the progress of over-optimistic LPs. We call this CPU a “FAST-REPOSITORY”. The CPU selected as a FAST-REPOSITORY is the one that has the maximum number of LPs labeled as FAST mapped to it beforehand. This reduces the number of moves that will be made in the following step to minimize overheads. Fig 4.4 shows the state of the simulation at the end of step 3. In this case we identify processor A as a FAST-REPOSITORY as it has the maximum number of FAST LPs mapped to it.

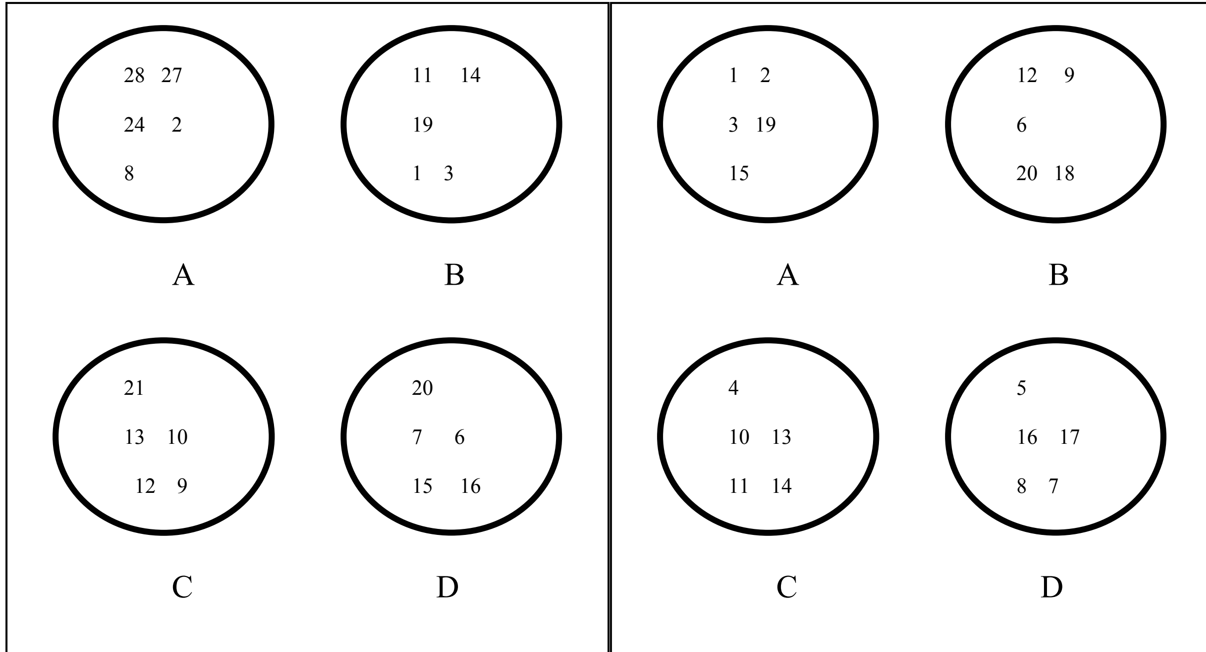

Fig. 4.1 Initial State of Simulation **Fig. 4.2 Ranking of LPs**

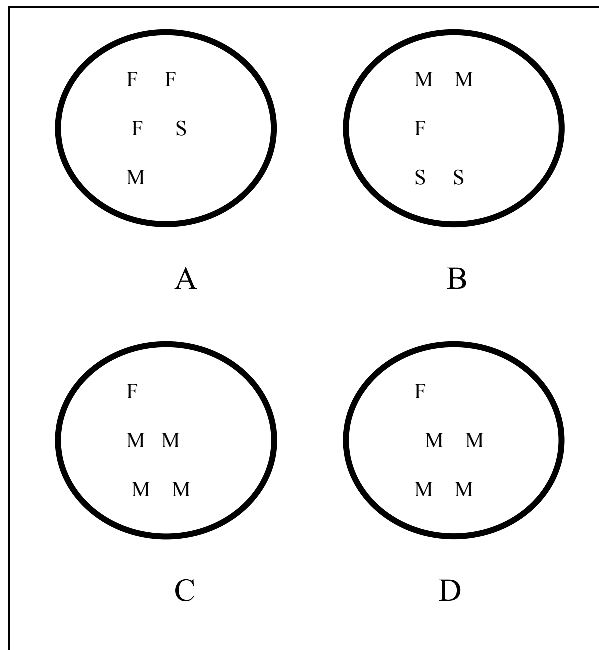

Fig. 4.3 Classification of LPs

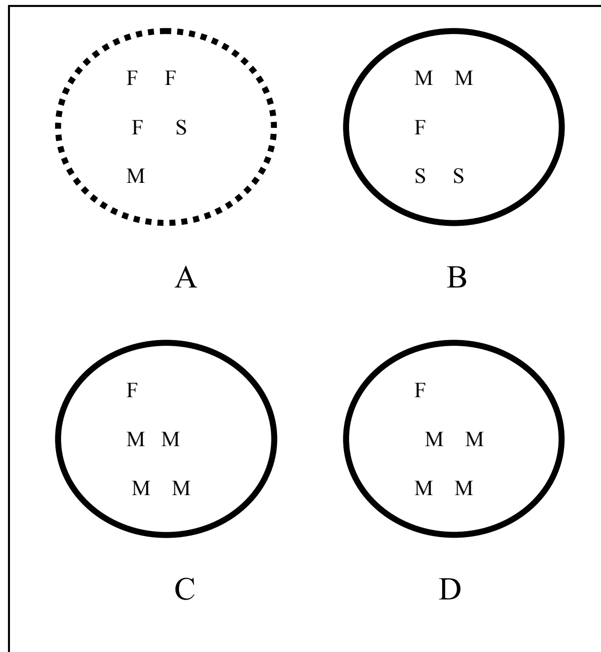

Fig. 4.4 Identification of FAST-REPOSITORY

(iv) ***Isolation of FAST LPs**:* The last two steps are essentially the heart of our approach. In this step we move all the FAST LPs to the FAST-REPOSITORY that are currently not mapped to the FAST-REPOSITORY. This is done to isolate their effects from the rest of the logical processes in the simulation and forces these optimistic LPs to compete for the CPU cycles of the FAST-REPOSITORY. Fig. 4.5 shows the FAST LPs that will be moved during this step, while Fig 4.6 shows the state of the simulation once the FAST LPs currently not mapped to the FAST-REPOSITORY are moved to it.

(v) ***Spreading of SLOW LPs**:* Once all the FAST LPs currently not mapped to the FAST-REPOSITORY are moved to the FAST-REPOSITORY, we redistribute the SLOW LPs to the remaining processors besides the FAST-REPOSITORY. At the end of this step, every SLOW LP will be on one processor each, providing them sufficient CPU cycles to catch up with LPs that are ahead in simulation time.

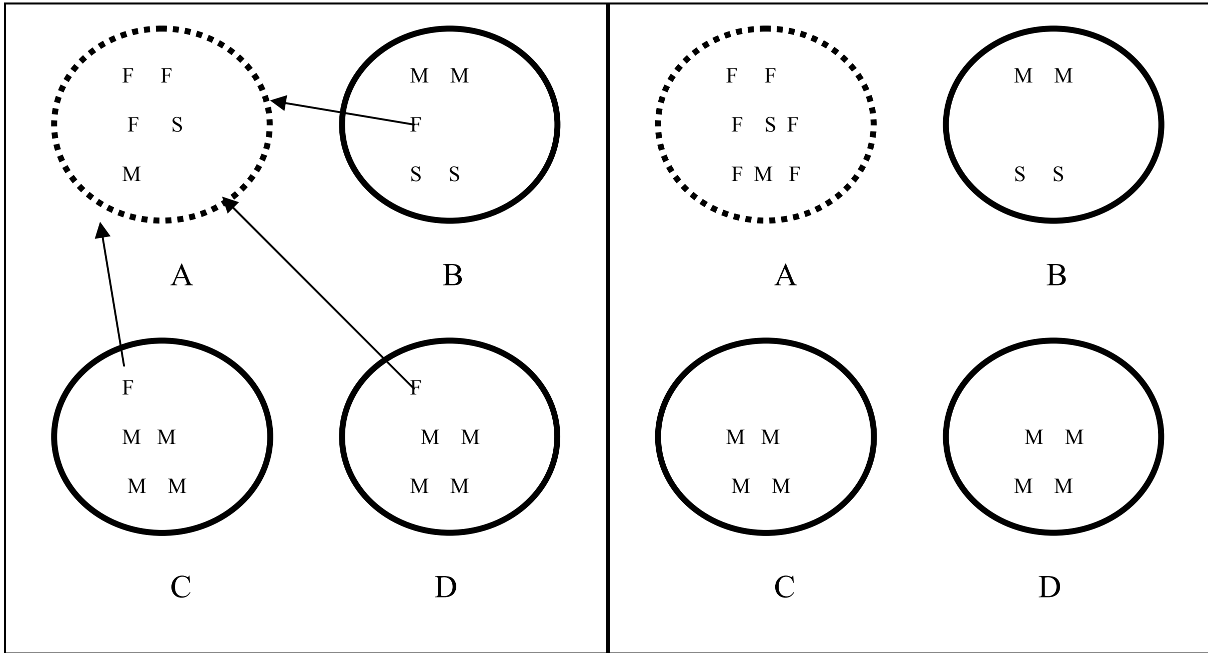

Fig. 4.5 Moves during Isolation of FAST LPs **Fig. 4.6 After Isolation of FAST LPs**

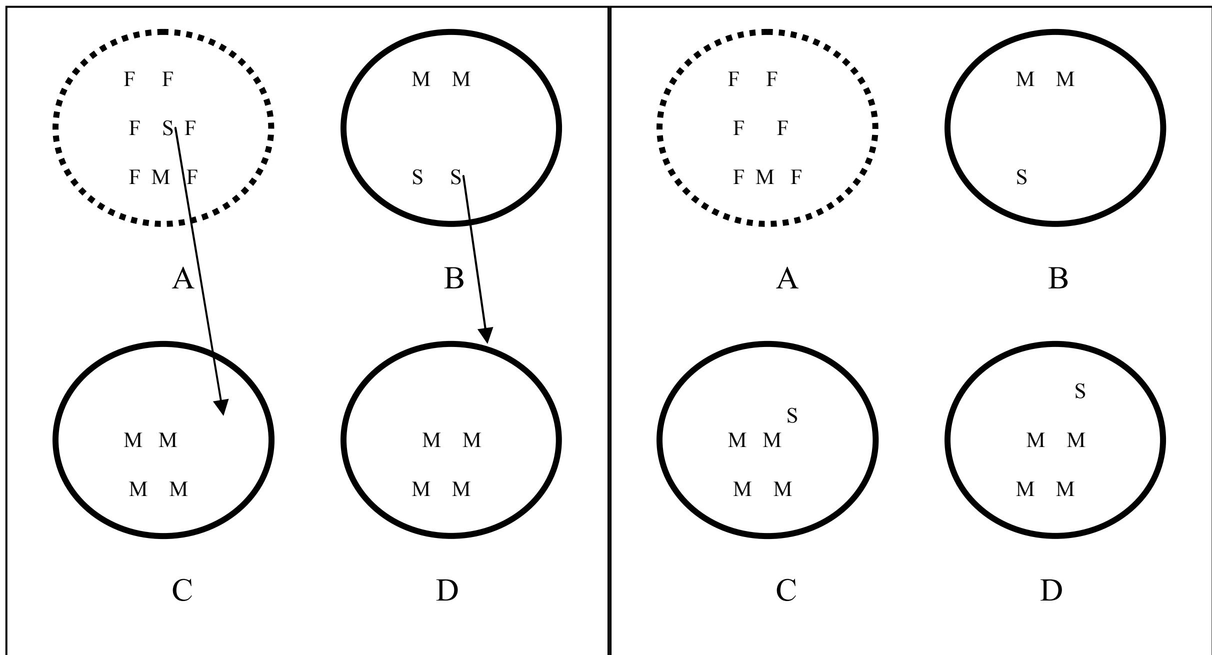

Fig. 4.7 Moves during spreading of SLOW LPs **Fig. 4.8 After spreading SLOW LPs**

Fig. 4.7 shows the moves of SLOW LPs to be made during this step, whereas, fig.4.8 shows the state of the simulation at the end of this step, once all SLOW LPs are redistributed. In our algorithm, we only move LPs classified as SLOW and FAST. MEDIUM LPs are not moved during the entire process.

4.2 IMPLEMENTATION

Our load balancing algorithm is implemented on the distributed Georgia Tech Time Warp system (GTW), which is a parallel and distributed discrete event simulation executive based on Jefferson' s Time Warp [16][25][17]. GTW runs on both shared memory machine and distributed memory machines. Details of GTW can be found in [25].

The distributed GTW consists of an additional thread which is created on each machine [15] that handles all the external communication with other machines. The Parallel Virtual Machine (PVM) communication library is used for remote communication [24]. We implemented our process migration algorithm in a separate thread called MonitorOPT on top of the distributed GTW.

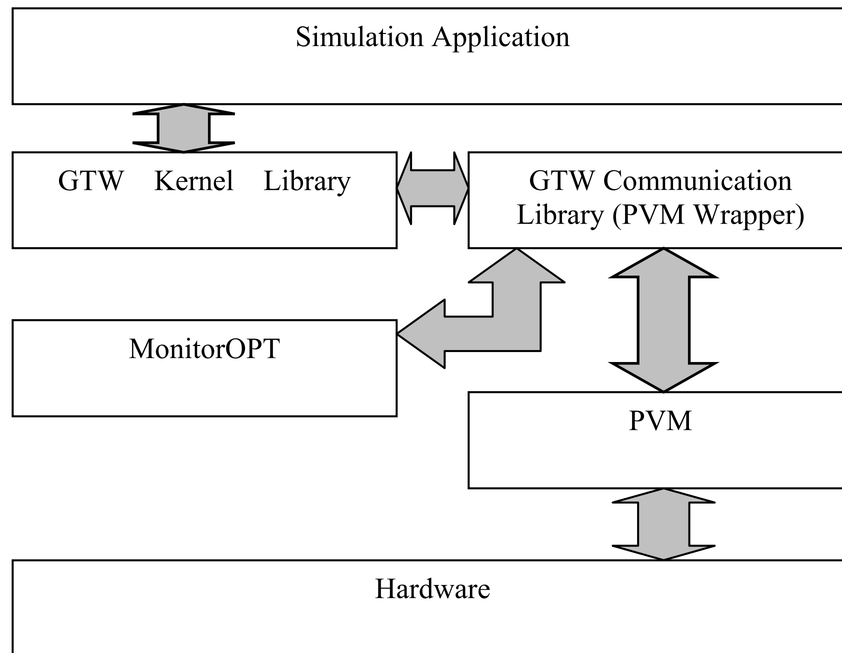

Fig. 4.9 Software architecture for Distributed GTW

The software architecture of the distributed GTW, including our thread, is shown in Fig.4.9. 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 functionalities. 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. The process of exchanging messages between the communication thread of the kernel library and MonitorOPTLB is described in detail below.

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.

We use "*GvtLag*" as a metric for our heuristics. GvtLag is the difference between the LP's current LVT and the system's GVT. GvtLag provides an indication of the degree of optimism of LPs. An LP having a higher value of the GvtLag than a second LP is more optimistic than the other LP and is further ahead in simulation time than the latter LP.

The MonitorOPT process collects statistics from each processor about the GvtLag statistics of each LP currently mapped to it. It then labels each LP as FAST, SLOW and MEDIUM, an indication of how optimistic one LP is relative to the other. The labels are determined as follows,

FAST : 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. K is

an experimental parameter and depends on the migration cost in the system. FAST LPs are the LPs that are furthest ahead in simulation time and are aggregated on a single processor to slow their progress.

SLOW: The N-1 LPs having the least value of GvtLag are labeled as SLOW, where N is the number of processor that are currently running the GTW kernel. These are the LPs that are the furthest behind in simulation time compared to other LPs and hence are redistributed to different processing elements.

MEDIUM: All the other LPs that are not labeled as FAST or SLOW are labeled as MEDIUM.

Once the labels for each LPs have been determined, the processor having the most number of FAST LPs currently mapped to it is identified. This processor is called the FAST-REPOSITORY. All the FAST LPs that are currently not mapped to this processor will be moved to the FAST-REPOSITORY.

After all the FAST LPs have been remapped to the FAST-REPOSITORY, the SLOW LPs are spread to the remaining N-1 processors, with one SLOW LP on each of the remaining processors.

Once the new mapping information of where each LP will be moved is computed a move-list is created having information about what LPs will be needed to move, the source PE from which it will be moved and to what destination PE will it be moved to. If MonitorOPT determines some moves 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 do the 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. The rollbacks are justified as over-optimistic LPs are already wasting CPU cycles by doing more rolls back than progressing forward.

After all the processors roll back their computations to GVT and 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 a particular LP will be moved to a destination processor. This is done so that the source processor can transfer all the information regarding this LP to the destination processor. Once the source PE transfers all the information regarding the LP to be moved to the destination PE, it sends a *MOVE-LP* message to the destination PE, in order to

complete the move. 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 the new mapping information which is the updated mapping of all LPs to processor to update the mapping structures used for migration of LPs in the future. 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. This process of MonitorOPT execution is shown below in fig.4.10.

```
while (Current Simulation Time <=  Simulation EndTime)
    After every Load Balancing Duration Do
                  Send UPDATE message to all PEs
                  All PEs reply with UPDATE-ACK message
                     sending GvtLag Statistics
                  Classify all LPs based on this GvtLag
                     Statistics as FAST, MEDIUM and SLOW
                  Identify FAST LPs to be moved to FAST-
                     REPOSITORY
                  Identify SLOW LPs to be redistributed to
                     PEs other than FAST-REPOSITORY
```

Fig. 4.10 Pseudo-Code for MonitorOPT

```
For every move identified fill the Move-List
   with LP to be moved, source and
   destination PE
If number of moves > 0
    For every Move in Move-List
         Send MOVE message to source PE
            sending destination PE and LP
            info
         Wait for MOVE-ACK from source
            PE
Compute the new mapping information of new
     mapping of LPs to PEs
Send MAPPING message to each PE
Wait for MAPPING-ACK from each PE
```

Fig. 4.10 Pseudo-Code for MonitorOPT contd…

CHAPTER 5

EXPERIMENTS

5.1 BACKGROUND

Many experiments were performed in order to compare the performance of the simulation running with and without process migration. The experiments presented in this study were performed in a heterogeneous environment consisting of 8 machines. These consist of one SGI Origin 2000 with sixteen 195 MHZ MIPS R 10,000 processor machine 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 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.

P-Hold

P-Hold is a benchmark application using a synthetic workload model [19]. 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 time stamp increments are fixed at 1. The total number of LPs for these experiments was fixed at 256. These LPs were evenly distributed on 8 processor with 32 LPs per processor. The experiments used a fixed message population of 6,400.

In order to affect 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, whereas in the one millisecond case processing included a 1 msec delay loop.

To make the application more imbalanced, besides having different event granularities or workloads, P-Hold was additionally configured as self-instantiated. This was done by varying the degree of self-instantiation of an event. The experiments used two degrees of self-instantiation: 50 and 200. 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 it. By "d generations" we mean 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. Here d refers to the degree of self-instantiation.

Our algorithm aggregates the FAST logical processes to slow their progress. A typical run is shown in Table 1 and 2. They show two snapshots of a simulation running on 8 processors with 256 logical processes. In this case we classify 32 FAST LPs and 7 SLOW LPs. Table 1 shows the state of the simulation during the nth cycle of our optimistic control algorithm. Table 1 shows the number of LPs classified as FAST, MEDIUM and SLOW mapped to each of the processors by MonitorOPT. Our algorithm identifies processor 4 as the FAST-REPOSITORY as it has the largest number of FAST LPs mapped to it. It then moves all the other FAST LPs currently not on processor 4 to the FAST-REPOSITORY. The SLOW LPs identified are all currently mapped to processor 3. These SLOW LPs are redistributed to the remaining processors besides processor 4 which is the FAST-REPOSITORY.

Table 5.1: Simulation state during n^{th} cycle of process migration

Processor	# of FAST LPs	# of MEDIUM LPs	# of SLOW LPs
0	10	22	0
1	3	29	0
2	1	31	0
3	0	25	7

4	11	21	0
5	0	32	0
6	7	25	0
7	0	32	0

Table 2 shows the state of the simulation during the (n+1) st optimistic control cycle. Here we observe that the previous "FAST-REPOSITORY" at processor 4 now does not have any FAST LPs mapped to it. In the current cycle, processor 4 has all the SLOW LPs, besides the other LPs which are classified as MEDIUM. This demonstrates that all the FAST LPs from last cycle have been slowed down and in this cycle have been classified as SLOW and MEDIUM. Whereas, processor 3 which in the previous cycle had all SLOW LPs mapped to it, during the (n+1) st cycle has only MEDIUM LPs. This shows by redistribution of SLOW LPs, the algorithm helped the SLOW LPs to make progress and catch up with FAST LPs.

Table 5.2 : Simulation state during n+1st cycle of process migration

Processor	# of FAST LPs	# of MEDIUM LPs	# of SLOW LPs
0	22	1	0
1	0	30	0
2	0	32	0
3	0	26	0
4	0	46	7
5	0	33	0
6	10	16	0
7	0	33	0

For the first set of experiments, two classes of restrictions involving both logical processes and events are as follows:

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. The reason for two classes of LPs and messages is to induce rollbacks. This occurs because LPs progress at different rates independently and infrequent messages from an LP in the past will induce a rollback.

A second set of experiments were performed for comparison. While keeping every 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 without any delay (null event granularity).

Class [200: null]: Events with degree of self-instantiation 200 and LPs execute events with 1 millisecond delay (1 msec event granularity).

If logical processes are moved too frequently overheads dominate because of migration costs. We determined empirically that a sample period of 50 seconds was appropriate for MonitorOPT. All the results presented here are the average over four trials.

5.2 PERFORMANCE RESULTS

5.2.1 Experiments under Set 1

In figure 5.1 below, we show the performance of three variations of our migration algorithm: (1) with fast repository and with slow LP spreading, (2) without fast repository and with slow LP spreading, (3) with fast repository and without slow LP spreading. We measure performance as the ratio of the useful work with process migration to the useful work without process migration.

The three variations are described below:

1) *With Fast Repository and with Slow LP spreading:* Here we classify FAST, SLOW and MEDIUM LPs. The number of FAST LPs is in terms of processors and is scaled with a factor of k, which is an experimental parameter. Our algorithm classifies N * k as FAST LPs (where N is the number of processors). In our experiment k = 5 and N = 8. The numbers of LPs classified as SLOW are fixed to 7 (N-1, where N is the number of processors in the system). LPs not classified as FAST and SLOW are classified as MEDIUM.
2) *Without Fast Repository and with Slow LP spreading*: Here we classify only SLOW and MEDIUM LPs. None of the LPs are classified as FAST. This is done by setting k = 0. We classify 8 (N, where N is the number of processors in the system) LPs as SLOW and remaining as MEDIUM.
3) *With Fast Repository and without Slow LP spreading*: Here we classify only FAST and MEDIUM LPs. None of the LPs are classified as SLOW. In our experiment we set the value of k = 5 which classifies 40 LPs (N *5, where N is the number of processors) as FAST. Remaining LPs are classified as MEDIUM.

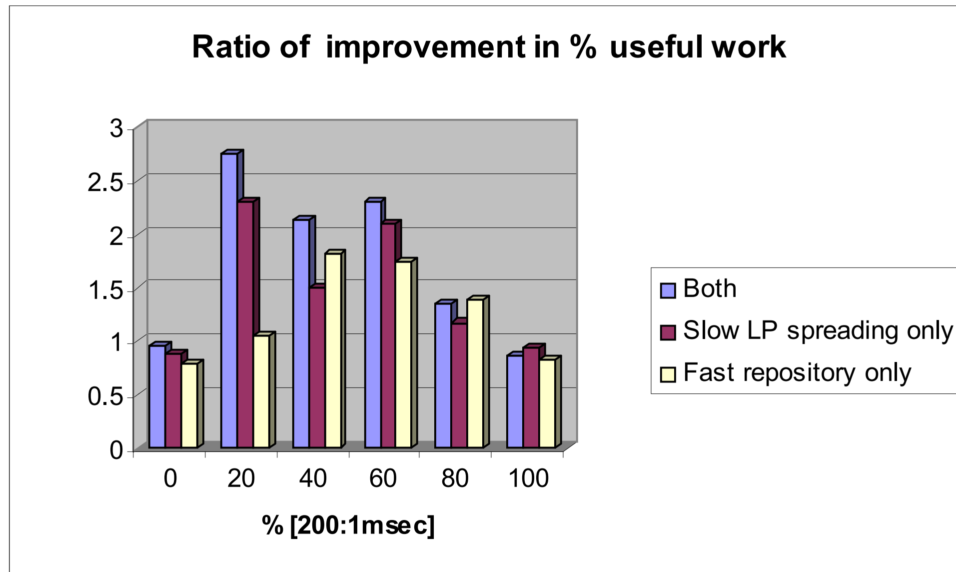

Fig. 5.1 Ratio of percentage improvement in useful work as compared to without process migration, (Set 1)

In figure 5.1, the simulation performs better with process migration and its variations 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%, when fast repository with slow spreading performs approximately 2.75 times better than the simulation without process migration. None of the three variations of process migration perform better than the simulation without process migration at data points 0 and 100. This is because the simulation is balanced; here 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] . Due to having the same event granularity, all the LPs progress at the same pace, causing less rollback. The simulation application has diverse levels of optimism between data point 0 and 100 due to LPs having different degrees of event granularities. The overhead of process migration is the cause of useful work of less than 1.

When the application is unbalanced, process migration and its different variation 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 most of its time rolling back its computation 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 roll backs.

The two variations of process migration also perform better than no process migration when the application is unbalanced (data points 20% – 80%). Each of these variations alternatively performs better than each other at various levels of optimism in the application. This happens because one of the variations balances the less optimistic LPs and the other try to balance the over-optimistic LPs. Hence, 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 process migration algorithm. In this combined approach, we control both the less optimistic and over-optimistic LPs providing better performance consistently, when the application is unbalanced.

Fig. 5.2 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 benefits in terms of execution time when the optimism is balanced(data point 0 and 100 when all LPs have same event granularity).

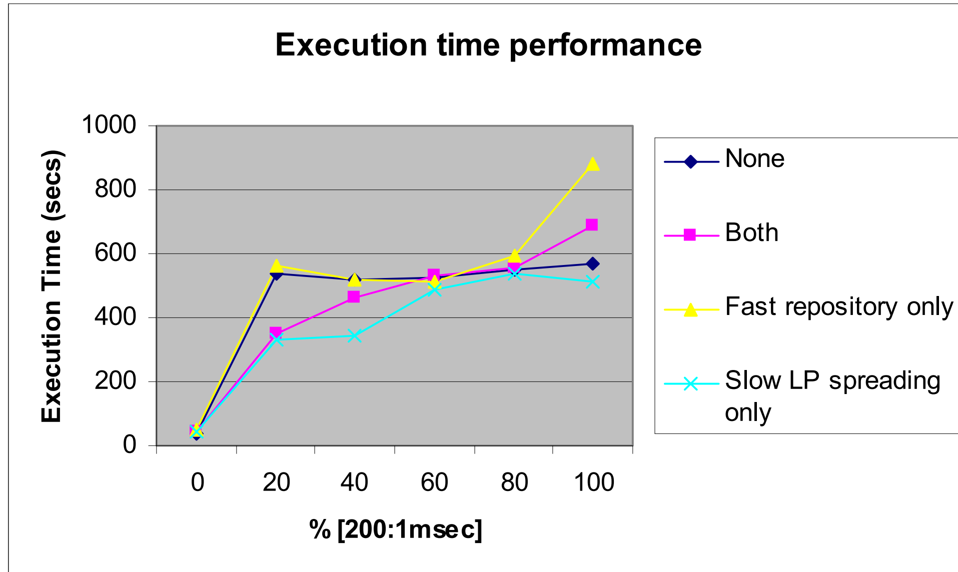

Fig 5.2 Execution time performance (Set 1)

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 better percentage of useful work as shown in fig. 5.1. For data points 60 and 80 it is observed that process migration takes almost equal time compared to without process migration. It is evident from Fig. 5.2 that even though we do not benefit in terms of execution time at these two data points but we benefit in terms of useful work as shown in fig. 5.1. This is because migration costs dominate. 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. This combined performance of process migration is beneficial in the case of external workload as fewer CPU cycles are used to perform useful work as compared to without process migration which could have been otherwise used for other user processes in the system.

For point 100, without process migration results performs better for both the percentage of useful work in fig. 5.1 and the execution time in fig. 5.2 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 progress at the

same pace. The overheads of process migration exceed the benefits obtained when the application is the balanced and least optimistic.

Fig. 5.3, shows the results of varying the value of K to classify number of fast LPs. This experiment is performed for the data point 40 of the previous experiment (60 percent of LPs have self-instantiation degree of 200 and 1 msec event granularity and 40 percent of LPs have self-instantiation degree of 40 with null event granularity). It is observed that as the number of fast LPs is increased, the

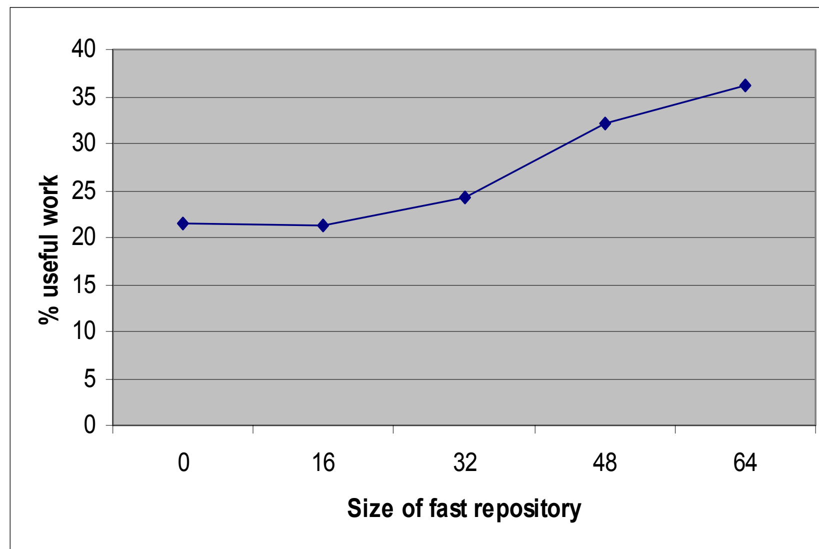

Fig. 5.3 FAST LPs scaling performance

percentage of useful work also improves. However, if k is large migration cost will dominate.

5.2.2 Experiment under Set. 2

Another set of experiments were 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]).

Fig. 5.4 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 Fig 5.4, it was observed that process migration performs better than without it at all the levels except at data point 0. This is because of the simulation application being balanced at data point 0. At data point 0, 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. 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.

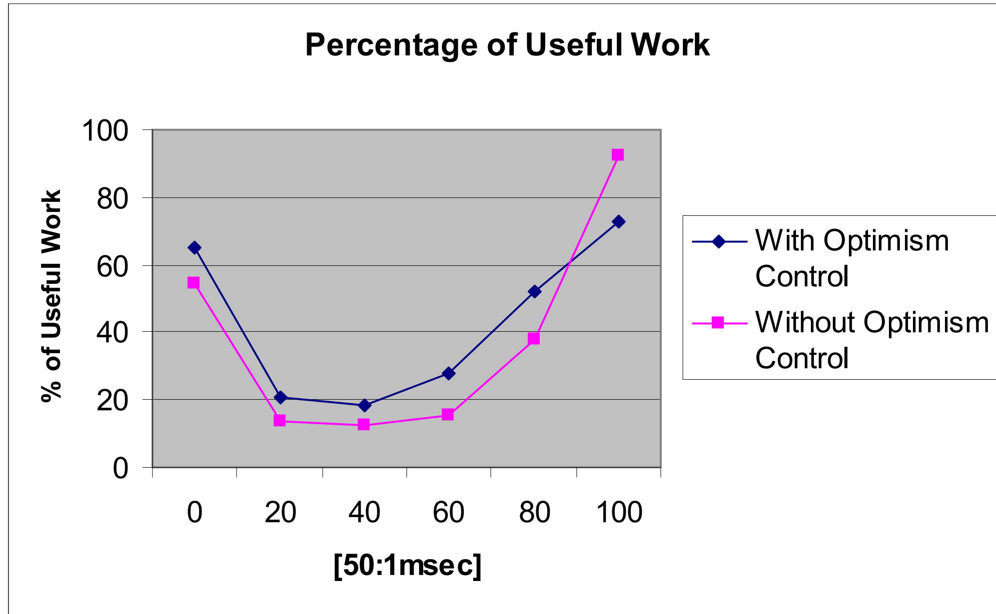

Fig. 5.4 Percentage of Useful Work (Set 2)

In fig. 5.5, 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 either of the two classes of self-instantiation and event granularity respectively, we do not observe any benefits in execution time. At data point 0, the simulation without process migration performs better that one 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. This combined performance for our algorithm at data point 100 is still useful as the simulation running with process migration utilizes fewer CPU cycles to make progress which could provide other user processes in the system sufficient CPU cycles. A similar scenario exists at point 20, 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.

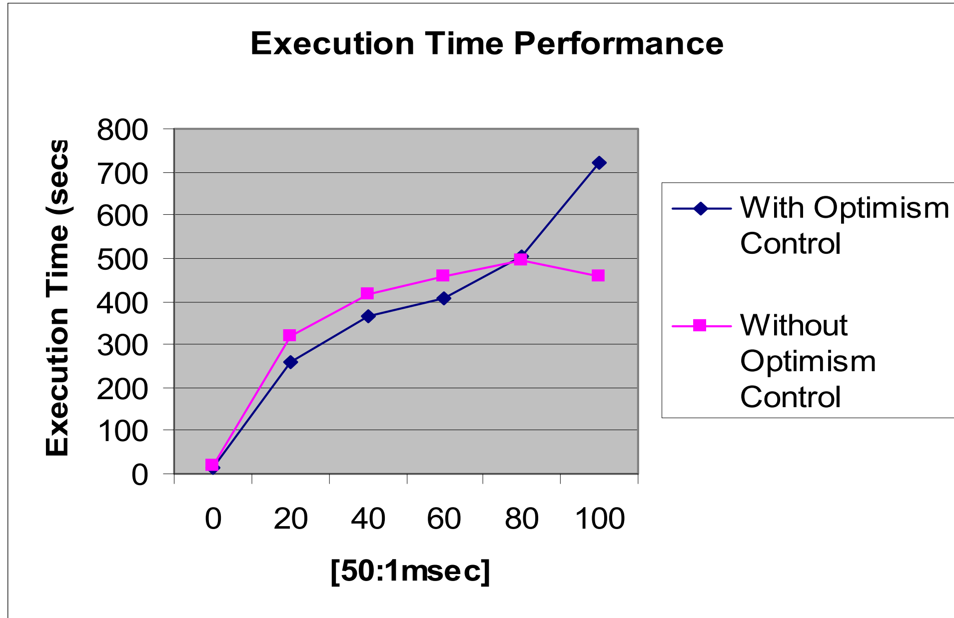

Fig. 5.5 Execution Time Performance (Set 2)

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 is some cases. One of the reasons for this penalty is due to the PVM communication which is expensive. The experiments were performed on older hardware, due to resource limitations. By updating the hardware and running the experiments on those, 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 may but 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.

CHAPTER 6

CONCLUSION AND FUTURE WORK

In a standard optimistic parallel event simulation, there are no restrictions on how far an LP can proceed ahead than 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 statistics collected to classify an LP is called *GvtLag*, which is the difference of an LPs current simulation time and the systems GVT. We aggregate the FAST LPs on one processor called as 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. On the other hand we spread the SLOW LPs, by redistributing each SLOW LP on 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 in two classes having either self-instantiation degree of 50 or 200 and with and without 1 msec delay in event processing.

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

We also propose to cluster LPs similar to [15], in order to reduce the process migration cost. This would need a different metric for evaluating process migration. We propose to evaluate such a possibility and observe its performance compared to without process migration.

REFERENCES

[1] D. R. Jefferson and H. Sowizral, *Fast Concurrent Simulation Using the Time Warp Mechanism, Part – I : Local Control,* ACM Transactions Programming Languages and Systems, vol 7, no. 3, pp 404-425, July 1985.

[2] J. Briner, *Fast Parallel Simulation of Digital Systems*, Advances in Parallel and Distributed Simulation, pp 20-28, June 1997.

[3] F. Wieland, E. Blair and T. Zukas, *Parallel Discrete Event Simulation (PDES): A Case Study in Design, Development and Performance Using SPEEDES*, In Proceedings of the 9th Workshop on Parallel and Distributed Simulation (PADS '95), pages 103 -110, June 1995.

[4] C. Carothers and R. Fujimoto, *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), pages 2-6, 1994 .

[5] F. Hao, K. Wilson, R. Fujimoto and E. Zegura, *Logical Process Size in Parallel Simulations*, In Proceeding of the Winter Simulation Conference , pp 645-652, December 1996

[6] R. M. Fujimoto, *Parallel and Distributed Simulation Systems*, Wiley Series on Parallel and Distributed Computing, Reading Pg 137 – 138, 2000.

[7] P. L. Reiher, F. Wieland, and D. R. Jefferson, *Limitation of Optimism in the Time Warp Operating System*, In Proceedings of the Winter Simulation Conference, pages 765-770, 1989.

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

[9] B. D. Lubachevsky A. Shwartz and A. Weiss, *Rollback Sometimes Works…if Filtered*, In Proceedings of the Winter Simulation Conference Proceedings, pp. 630 – 639, Dec 1989.

[10] Madisetti V. K, Hardaker, D.A. , and Fujimoto, R. M. 1983, *The MIMIDIX Operating System for Parallel Simulation and Supercomputing*, J. Parallel and Distributed Computing, vol 18, no. 4, pp. 473-483, August 1993.

[11] Steinman, J. S. 1983, *Breathing Time Warp*, In Proceedings of the Seventh Workshop on Parallel and Distributed Simulation, vol 23, pp. 109-118, May 1993.

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

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

[14] K. Jones and Samir R. Das, *Combining Optimism Limiting Schemes in Time Warp based Parallel Simulations*, In Proceedings of the 1998 Winter Simulation Conference, pages 499--505, Dec. 1998 .

[15] C. D. Carothers and R. M. Fujimoto, *Efficient Execution of Time Warp Programs on Heterogeneous, NOW Platforms*. IEEE Transactions on Parallel and Distributed Systems, Vol. 11, No. 3, March 2000.

[16] D. R. Jefferson, *Virtual Time*, ACM Trans. Programming Languages and Systems, vol. 7, no. 3, pp. 404-425, July 1985.

[17] S. Das, R. Fujimoto, K. Panesar, D. Allison and M. Hybinette, *GTW: A Time Warp System for Shared Memory Multiprocessors*, In Proceedings of the Winter Simulation Conference 1994, pp. 1,332-1,339, December 1994.

[18] Y. M. Teo, Y. K. Ng and B. S. S. Onggo, *Conservative Simulation using Distributed Shared Memory,* In Proceedings of the 16th Workshop on Parallel and Distributed Simulation (PADS.02) 1087-4097/2.

[19] R. M. Fujimoto, *Performance of Time Warp under Synthetic Workloads*, In Proceedings of the SCS Multiconference on Distributed Simulation, vol. 22, pp. 23-28, Jan. 1990.

[20] C. Burdorf and J. Marti, *Load Balancing Strategies for Time Warp on Multi-User Workstations*, The Computer Journal, vol 36, no.2, pp. 168-176, 1993.

[21] D. Knuth, *The Art of Computer Programming, Vol 3*, Addison-Wesley, Reading, MA 1969.

[22] J. P. Fitch, *Compiling for Parallelism*, In Proceeding of the European Workshop of Parallelism and Algebra, pp. 19-32, 1989.

[23] J. Fitch and J. Marti, *The Bath Concurrent Lisp Machine*, Springer-Verlag, Berlin 1983.

[24] V. S. Sunderam, *PVM: A Parallel Virtual Machine for Parallel Distributed Computing*.

[25] 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-CC97-18, College of Computing, Georgia Inst. of Technology , July 1997 .

[26] K. M. Chandy and J. Misra, *Distributed Simulation: A Case Study in Design and Verification of Distributed Programs*, IEEE Transactions on Software Engineering, SE-5:5, pp. 440-452, 1979.

[27] R. Ayani, and H. Rajael, *Parallel Simulation Using Conservative Time Windows*, In Proceedings of the Winter Simulation Conference, pp. 709-717, 1992.

[28] S. Das and R. M. Fujimoto, *Adaptive Memory Management and Optimism Control in Time Warp*, ACM Transactions on Modeling and Computer Simulation (TOMACS), Volume 7, Issue 2, 1997.