

Shock Resistant Time Warp *

Alois Ferscha and James Johnson
Insitut für Angewandte Informatik

Universität Wien

Lenaugasse 2/8, A-1080 Vienna, AUSTRIA

Abstract

In an attempt to cope with time-varying workload, traditional adaptive Time Warp protocols are designed to react in response to performance changes by altering control parameter configurations, like the amount of available memory, the size of the checkpointing interval, the frequency of GVT computation, fossil collection invocations, etc. We call those schemes “reactive” because all control decisions are undertaken based on historical performance information collected at runtime, and come into effect in future system states. What must be considered a drawback of this class of approaches is that Time Warp logical processes (LPs) have most likely reached a state different from the one for which the control action was established - thus inducing performance control activities which are always outdated.

This paper develops environment aware, self adaptive Time Warp LPs implementing a pro-active performance control scheme, addressing the timeliness of control decisions. Opposed to reactive TW schemes, our pro-active control mechanism based on a statistical analysis of the state history $\dots, S_{t-2\Delta}, S_{t-\Delta}, S_t$ periodically collected in (real) time intervals of size Δ , forecasts a future LP state $\hat{S}_{t+\Delta}$. A performance control decision $CD(\hat{S}_{t+\Delta})$ is established, that is most appropriate for the expected future LP state, i.e. the state when the corresponding control activity would become effective. Depending on the forecast quality, a pro-active scheme will presumably exhibit performance superior to reactive schemes, at least for cases where state changes in the time frame Δ are very likely.

In this paper we study the ability of pro-active TW LPs, to adapt to sudden load changes, especially to abruptly occurring background workloads injected by other applications executing concurrently with the TW simulation on a network of workstations. Experimental results show that the protocol is able to capture abrupt changes in both computational and communication resource availability, justifying the title: “shock resistant Time Warp”.

1 Introduction

The parallel and distributed simulation (PADS) community has face the problem of integrating heterogeneous and dislocated computing resources into a single “virtual” simulation environment since the days of the introduction of

platform independent message passing libraries (like PVM or MPI). Whereas the classical distributed simulation synchronization protocols have been reimplemented with these libraries, the complexity of the message passing software and the inherent non-predictability of application and system behavior have made “traditional” performance tuning techniques appear inappropriate. In fact, so much so, a 1994 opinion leading paper [1] was indeed very successful in attracting considerable parallel and distributed simulation research in the area of adaptive performance tuning strategies. A (somewhat) recent survey browses through these achievements [2].

Not only was the PADS community attracted by the “new performance issues”, the whole parallel and distributed community was concerned [3, 4, 5, 6, 7]. Within PADS, early experiments tried to map several logical (simulation) processes (LPs) onto a single processor within a distributed execution environment. “Automated” (static) load balancing was studied in [8, 9] while dynamic strategies (mostly based on migration policies for LPs [10]) were followed in [11], [12], [13] and recently [14].

Many of the performance engineering techniques developed in other contexts of “network-of-workstation” (NOW) based distributed computing [15, 16, 17, 18] diffused into the PADS community, helping to define the state of the art in load sharing or load balancing for parallel and distributed discrete event simulation (PDES): Unfortunately, opposed to the traditional approach of defining “load” based on processor utilization, these metrics are impractical for establishing load control decisions in Time Warp (TW). The reason is because CPU time is used optimistically, possibly without significant contribution to the overall simulation progress [1]. The approaches of dynamic load balancing for PDES must use the event rate (committed events per unit CPU time) to dynamically establish control decisions for the redistribution of initial or forthcoming load configurations in order to optimize overall execution performance of a distributed simulator. The load metric is extracted from (dynamically collected) system state information, and appropriate load management decisions are being undertaken accordingly. Either a centralized load coordinator or a distributed scheme is used to put dynamic control decisions into effect.

Mainly two observations from the current “dynamic load balancing” literature within PADS have motivated to develop a new TW protocol:

- (i) Considering a NOW as the execution environment for a distributed TW simulation, traditional balancing

*This work was supported by the Human Capital and Mobility program of the EU under grant CHRX-CT94-0452 (MATCH).

approaches fail to deliver robust performance by neglecting background loads inherent to multiuser, multi-session, multitasking operating systems and nonpredictable IP traffic caused by standard Internet services. Control mechanisms based exclusively on LP state information are intuitively insufficient and inappropriate.

- (ii) Considering the nonpredictability of background CPU load (due to the nonpredictable amount and type of user interaction in background applications), and (consequently) the nonpredictability of LAN latency (due to nonpredictable communication requests by LAN users and the non-guaranteeing nature of IPv4), the implicit assumption of “timely” control effects is simply inadequate. Control decisions affecting all LPs executing in the NOW always apply with a lag in real time — more than that, the delay for becoming effective is different for each LP.

The TW protocol proposed in this work therefore attempts the following performance issues.

- (i) TW is a very irregular distributed application “*per se*”, with complex, data-dependent execution behavior, and dynamic, time varying resource demands [19]. A nontrivial confluence of performance factors makes it impossible to identify a single “typical” behavior or resource usage pattern upon which a load control strategy could be defined. In addition, the performance delivered by the execution platform is hard to predict because a NOW typically links heterogeneous CPUs, each responding differently to the application requests and often acting in an uncorrelated way (e.g. no coordinated process scheduling is performed). Since controlling the execution behavior of TW “*from outside*” is not enough, TW LPs must be *environment aware*.
- (ii) Potentially large performance fluctuations result from communication resources being shared by many users and their respective background processes, and several layers of software (e.g. operating system, run time support for parallel APIs, etc.) interacting asynchronously on each machine. Since load balancing mechanisms reliant to the same communication resources serving the TW application itself as well as background users (e.g. to gather state information in a central load coordinator and to broadcast control information to LPs) is prone to intrusion, TW LPs must be *self adaptive*.
- (iii) Since traditional NOW environments, particularly the ones involving TCP/IP based communication, typically exhibit comparably small computation to communication speed ratios, the (real) time gap between the state information collection and the launch of the respective control actions can never be neglected. Opposed to traditional approaches of balancing that “*react*” to changing load situations (reactive LPs), TW LPs must be *pro-active* in that they adapt to anticipated future system states.
- (iv) Since the injection of background load into a NOW typically occurs as sudden events, occupying the requested resources to an almost exhaustive extent (e.g.

a compile job for computational resources, or ftp for communication resources), i.e. induces “shocks” to the TW execution, TW LPs must be *shock resistant*.

In the sequel of this paper we will develop *environment aware, self adaptive, proactive, shock resistant* TW LPs. The paper is organized as follows: Section 2 presents the concept of a pro-active performance control strategy for TW. Section 3 develops the notion of TW execution trajectories in the space spanned by performance factors, for which Section 4 develops an incremental clustering technique. Clusters of approximately identical TW performance profiles are identified on the fly, and memorized during the TW execution evolution. In Section 5 we present results and empirical evidence of TW Logical Processes able to pro-actively adapt to load shocks. All experiments are executed in a network of workstation environment with TW LPs utilizing the MPI communication library. Conclusions are drawn in Section 6, encouraging the embedding of shock resistant TW in World Wide Web based simulation environments, which by nature lack quality of service guarantees and behave unpredictable with respect to the consumption of computational and communication resources.

2 Methodology

Opposed to dynamic load balancing strategies involving a centralized load manager [15], the concept followed here for making TW LPs “environment aware” is a fully distributed one. In our approach, each TW LP is enhanced by a mechanism able to autonomously control the local execution — independently of any other LP. As such, a TW LP once executing on a node in a heterogeneous NOW, considers all the other software processes executing on the same node as its “environment”. In a time varying way, an LP shares the local resources with its environment, such that the simulation performance of that LP is directly dependent on the resource consumption by the environment. A heavily background-loaded node in a NOW e.g. can throttle the simulation speed of the LP it hosts, which in turn has an indirect effect on the simulation speed of LPs executing on other nodes. Eventually, particularly in the case of closed message loops or recursive rollback cascades, an LP might suffer even from its own execution behavior. An explicit representation of all the performance factors of an LP’s execution speed hence appears intricate and impractical. We therefore propose that LPs — by observing their own performance behavior — implicitly develop a picture of the behavior of their environment, and act (develop performance control decisions and execute them) accordingly.

An environment aware LP is built around a “classical” TW LP (implementing the TW synchronization protocol on top of a discrete event simulation engine) by adding an LP control component (see Figure 1). (We call the “classical” TW LP *simulation component* in Figure 1.) The LP control component (LPCC) basically collects information on the incoming and outgoing message dynamics of the LP together with its execution dynamics, and combines those to an abstract state vector (sensory input analysis). Based on the trajectory described by the state vector history, the *state forecast* component makes an attempt to forecast the anticipated LP state at time $t + \Delta$, for which a *control decision* component develops an optimal performance control plan. The performance control plan is put in action via a set of

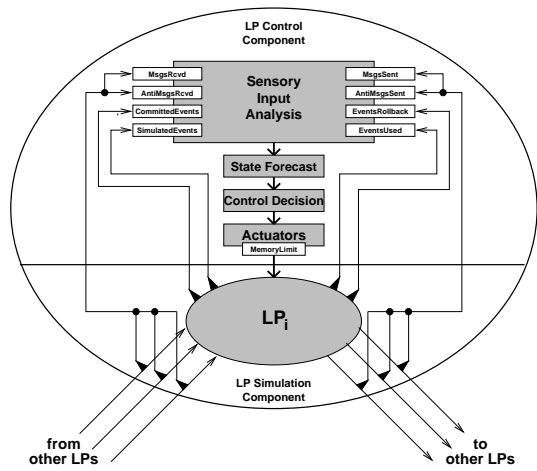


Figure 1. LP Control Component and LP Simulation Component

actuators, which directly affect the execution speed of the “controlled” LP.

Many different performance control mechanisms could be embedded in this generic frame of sensor/actuator interplay (e.g. performance control via the checkpointing interval, the frequency of cancelback calls, the frequency of GVT computations, etc.). In Figure 1 we follow a mechanism that limits the optimism in TW in order to control performance - but wish to stress that this is just one instance of a variety of possibilities. Translating the idea of memory based throttling (originally developed in the context of shared memory TW implementations [20]), we can control the performance of a TW LP by restricting the amount of available memory: whenever an event occurrence is to be simulated, an artificial throttle will delay the execution should insufficient memory be available to the simulation engine. The key rationale behind memory based throttling is to prevent LPs from progressing too far into the simulated future thus running into rollback hazards which degrade performance, but also to provide sufficient memory to prevent performance degradation caused by frequent cancelback calls. In this work, the key rationale behind using memory based throttling as an example is to demonstrate that only information local to an LP is used for conducting performance control.

2.1 The LP Control Component

In order to collect LP performance and state data the LPCC relies on *sensors* which are inserted into the program code. In the present implementation, a sensor is simply a global variable which is assigned the value of the corresponding performance/state metric at an appropriate point in the program code. Two types of sensors are defined: *point sample sensors* which register the momentary value of a performance/state metric and *cumulative sensors* which sum a specific metric over the duration of execution (e.g. the occurrence of run-time events).

The appropriate performance metric for TW simulation is the amount of meaningful simulation work accomplished. The corresponding sensor, *CommittedEvents*, is implemented as a global integer variable which is incremented

	Sensor/Actuator	Description
S_1	CommittedEvents	total number of events committed
S_2	SimulatedEvents	total number of events simulated
S_3	MsgsSent	total number of (positive) messages sent
S_4	AntiMsgsSent	total number of antimessages sent
S_5	MsgsRcvd	total number of (positive) messages received
S_6	AntiMsgsRcvd	total number of antimessages received
S_7	EventsRollback	total number of events freed during rollbacks
S_8	EventsUsed	momentary number of events in use

Figure 2. Time Warp Sensors

	Indicator	Description
I_1	EventRate	number of events committed per second (perf. indicator)
I_2	SimulationRate	number of events simulated per second
I_3	MsgsSentRate	number of (positive) messages sent per second
I_4	AntiMsgsSentRate	number of antimessages sent per second
I_5	MsgsRcvdRate	number of (positive) messages received per second
I_6	AntiMsgsRcvdRate	number of antimessages received per second
I_7	EventsRollbackRate	number of events freed during rollbacks per second
I_8	MemoryConsumption	average number of events in use

Figure 3. Time Warp Indicators

at fossil collection time in accordance with the number of events committed. The sensor *CommittedEvents* is consequently a cumulative sensor since its value increases steadily over the course of execution.

In order to achieve a more exact system state characterization, seven further sensors have been implemented. The six sensors *SimulatedEvents*, *MsgsSent*, *AntiMsgsSent*, *MsgsRcvd*, *AntiMsgsRcvd* and *EventsRollback* cumulate the number of events simulated, (positive) messages sent, antimessages sent, (positive) messages received, antimessages received and number of events rolled back respectively. Each sensor is incremented at an appropriate point in the program code. The point sample sensor *EventsUsed* reflects the momentary number of events which are currently in use by the simulation engine. The sensor *EventsUsed* is incremented when an event is taken from the free (memory) pool and decremented when an event is returned. Consequently, this sensor reflects the amount of memory TW is currently using in order to continue with simulation. Events which are “in use” are typically either 1) on the state stack waiting to be freed by fossil collection or rollback, 2) in the input queue holding an event contained in a message that has been received and not yet simulated or 3) in the output queue waiting to be sent out. Sensors are summarized in Figure 2.

Since cumulative sensors increase continuously during program execution and point sample sensors may fluctuate considerably from one sampling point to the next, the LPCC applies a smoothing function to all sensors S_i in order to obtain a metric which reflects the current tendency of the

sensor and which may be used to compare system states and assess their similarity. We refer to this derived metric as an *indicator*. In the present implementation, each indicator I_i is derived from exactly one sensor: $I_i = f(S_i)$. Indicators are summarized in Figure 3.

For cumulative sensors, the LPCC uses the *rate of increase* (in units per second) of the sensor in a moving window $[t, t - \Delta]$ as the corresponding indicator value and for point sample sensors, the *mean value* in the moving window. Assuming n observations of the sensor S_j in the current window, the value of the corresponding indicator I_j at sampling time t is calculated as:

Cumulative Sensor	Point Sample Sensor
$I_i(t) = \frac{S_i(t) - S_i(t - \Delta)}{\Delta}$	$I_i(t) = \frac{\sum_j S_k(t_j)}{n} \quad \forall j: t - \Delta \leq t_j \leq t$

In order to characterize the current performance of the executing simulation, one (or more) indicators are designated as *performance indicators*. For TW, this is naturally the number of events committed per second, the `EventRate`. The task of the LPCC is to optimize the value of the `EventRate` indicator. This is achieved by setting the values of executional parameters called *actuators*.

As with sensors, actuators are inserted at appropriate points in the program code in order to control execution characteristics. As previously stated, the control mechanism aims at throttling optimism by controlling the amount of memory available to the TW simulation engine. The actuator `MemoryLimit` prohibits the simulation of the next event if the value of the sensor `EventsUsed` is higher than the value of the actuator. The actuator throttle resides in the main simulation loop immediately preceding the actual event simulation step. Thus, if the simulator is blocked due to insufficient memory, the simulator loops back to the top of the simulation loop and may continue receiving messages and GVT calculation packets until enough memory is freed through fossil collection or rollback in order to proceed with simulation. Alternately, due to changes in the current system state, the LPCC may choose to raise the `MemoryLimit` which may ultimately allow for further simulation.

It is important to note here that the actuator `MemoryLimit` actually represents a *mean* value for the number of events in use (as it is derived from the indicator `MemoryConsumption` which is itself a mean value). Setting a hard ceiling on the number of available events at `MemoryLimit` would effectively limit the *mean* memory usage to a value substantially lower than `MemoryLimit` since all natural fluctuations in memory usage would be “cut off” at the value of `MemoryLimit`.

In order to allow for natural fluctuations in memory usage, the LPCC tests if it is plausible that the current memory usage at time t , `EventsUsed(t)`, stems from a normal distribution of $N(\text{MemoryLimit}, \hat{\sigma}_{mem})$ at a confidence level of $(1 - \alpha) = 95\%$ where $\hat{\sigma}_{mem}$ is an estimate for the standard deviation of memory usage calculated by the LPCC. Using the fact that for a random variable x , if $x \sim N(\mu, \sigma)$ then $(x - \mu)/\sigma \sim N(0, 1)$ and $P((x - \mu)/\sigma \leq z_\alpha) = \alpha$, the LPCC effectively tests if $(\text{EventsUsed} - \text{MemoryLimit})/\hat{\sigma}_{mem} \leq z_{0.05}$.

3 Sampling the Execution Trajectory

In a geometrical interpretation, the set of indicators can be envisioned as spanning an n_I -dimensional vector space,

the *indicator space*. Each vector of sampled indicator data, $\vec{s}(t) = (I_1(t), I_2(t), \dots, I_{n_I}(t))$, can be viewed as a point in this space. During the course of program execution, the sequence of sampled points consequently describes a *trajectory* through the n_I -dimensional indicator space. The execution trajectory thus represents the ordered set of all system states which have been visited during the course of program execution.

The task of the LPCC is to use the information contained in the execution trajectory to identify a set of similar system states and steer program execution towards those states which have better performance component(s), i.e. a higher `EventRate`. After a promising target state, $\vec{s}_{opt} = \vec{s}(t_{opt}) = (I_1(t_{opt}), I_2(t_{opt}), \dots, I_8(t_{opt}))$, has been chosen, the actuator value `MemoryLimit` is set to the indicator value `MemoryConsumption` of the target state. Execution now continues under the constraints set by the current actuator values.

Critical to the above is the identification of *similar* system states. Since the LPCC is free to set actuator values as it deems necessary, indicators derived from actuators (in the particular case `MemoryConsumption`) need not be considered when assessing system state similarity since (almost) any arbitrary value can be reached by setting the actuator(s) accordingly. Furthermore, as the LPCC aims at optimizing the performance indicator, the indicator `EventRate` also need not be considered since it is hoped that the `EventRate` will move towards that of the target state after the actuator have been set to those of the target state. Therefore, only the remaining *pure* indicators `SimulationRate`, `MsgsSentRate`, `AntiMsgsSentRate`, `MsgsRcvdRate`, `AntiMsgsRcvdRate` and `EventsRollbackRate` need be considered when assessing the similarity of system states. Two system states can therefore be considered similar when the Euclidean distance between their vectors of *pure* indicators is relatively small.

Recording every system state visited during the course of execution, however, is clearly not feasible due to the enormous amount of memory this would require. Even if this were possible, data retrieval would be much too time consuming to be of any avail. Consequently, a method for recording the execution trajectory information is required which is fast and also allows for quick retrieval.

Clustering fulfills these requirements, especially since the primary task of the LPCC is to identify similar system states. Static techniques, however, are not feasible due to the fact that new points must be repeatedly added to the data set and clustered. To overcome this difficulty, an incremental (“on-the-fly”) clustering technique is used for classifying similar system states.

4 Clustering Similar System States

Numerous incremental clustering techniques exist, each having its own specific advantages and disadvantages so that the choice of technique appears primarily motivated by the requirements of the problem to which it is to be applied. The algorithm chosen for implementation here is a variation of the *doubling algorithm* which has been applied successfully in the domain of information retrieval [21].

The doubling algorithm assumes a fixed set of n_C clusters C_i with centroids \vec{c}_i . When a new point is added for clustering, it is merged with an existing cluster if the distance to nearest cluster is less than the minimum intercluster

1. Initialization		
The first n_C data points become the centroids of the n_C clusters. All intercluster distances $d_{i,j} = \ \vec{c}_i - \vec{c}_j\ $ are calculated, sorted and stored on the heap. The “hit count” of all clusters is set to 1.		
LOOP	2. Add new data point and determine nearest cluster	
	The current vector of indicator data $\vec{p} = (I_1(t), I_2(t), \dots, I_{n_I}(t))$ is considered as the centroid of a new cluster C_{new} . Distances between the n_C cluster centroids and the new data point $d_{new,j} = \ \vec{p} - \vec{c}_j\ $, $j = 1 \dots n_C$ are calculated. The nearest cluster C_{j_0} with $d_{new,j_0} = \min_j d_{new,j}$ is determined.	
	3. Retrieve minimum intercluster distance	
	The minimum intercluster distance d_{j_1,j_2} is retrieved the heap.	
	4a. Merge with existing cluster	4b. Merge two existing clusters
	if $d_{new,j_0} \leq d_{j_1,j_2}$ then merge C_{new} and C_{j_0} into cluster C_{j_0} with new centroid $\vec{c}'_{j_0} = (h_{j_0} \cdot \vec{c}_{j_0} + \vec{p}) / (h_{j_0} + 1)$. The “hit count” of cluster C_{j_0} is set to $h'_{j_0} = h_{j_0} + 1$.	if $d_{j_1,j_2} < d_{new,j_0}$ then merge clusters C_{j_1} and C_{j_2} into C_{j_1} with centroid $\vec{c}'_{j_1} = (h_{j_1} \cdot \vec{c}_{j_1} + h_{j_2} \cdot \vec{c}_{j_2}) / (h_{j_1} + h_{j_2})$ and “hit count” $h'_{j_1} = h_{j_1} + h_{j_2}$. The new data point \vec{p} becomes the centroid of cluster C'_{j_2} with “hit count” $h'_{j_2} = 1$.
	5. Calculate minimum intercluster distances	
	Distances between all clusters and cluster C_{j_0} are calculated, sorted and stored on the heap.	Distances between clusters C_{j_1} and C_{j_2} and existing clusters are calculated, sorted and stored on the heap.

Figure 4. Incremental Clustering Algorithm

distance. If, however, the distance to the nearest cluster is greater than the minimum intercluster distance, the two existing clusters which have the least minimum distance are merged together and the new point becomes the centroid of a new cluster. Unlike the doubling algorithm, a “hit count” is recorded for each cluster. When a point is added to a cluster, its hit count is incremented by one. In the calculation of the new centroid for the cluster the hit count is used to weight the displacement of the centroid. Clusters having a large number of hits thus become less and less mobile as execution continues. The algorithm is described in Figure 4.

5 State Forecasting and Execution Control

The experimentation environment is a cluster of four Silicon Graphics workstations within the department’s LAN (Figure 5 left) with each LP executing on a single machine. The TW simulation engine was implemented in C using the MPI message passing library MPICH [22]. The structure of the simulation model (Figure 5 right) guarantees for a balanced load among the four LPs. The simulation model consists of 32 “nodes” which are interconnected by a torus topology and partitioned among the LPs according to the figure. Each node is initialized with a constant (and persistent) number of events which then circulate along closed loops. The virtual time increments of events between nodes in “horizontal” loops (i.e. between LP_0 and LP_1 and between LP_2 and LP_3) is $\Delta_h \sim N(100.0, 1.0)$ while increments in “vertical” loops is $\Delta_v \sim N(1.0, 0.01)$ which guarantees for a good deal of rollbacks.

An imbalance in processing power, however, exists among the machines, since one is an Indigo² running at 195 MHz while the other three are identical O2’s running

at 180 MHz. To aggravate the situation even further, a load “shock” was injected every 20 seconds on the O2 simulating LP_2 (gcc job executing for 5 seconds which consumed approx. 50% of the nodes processing power). The experiment was run for 100 hours (on four consecutive weekdays) against the unpredictable background load of the department’s LAN and user loads on the individual machines.

The results in Figure 6 show the aggregated means and standard deviations for each indicator over a 100 hour time period. The EventRate for the performance control method is approximately 37% higher than for unthrottled simulation which confirms the effectiveness of the approach. Furthermore, the standard deviation of the EventRate for controlled simulation is lower than for unthrottled showing that the LPCC is able to reproduce similar results despite unpredictable background loads.

As would be expected, the SimulationRate is highest in both cases for LP_0 which is executing on the fastest machine, and lowest for LP_2 which is executing on the “shocked” OS2. This consequently leads to a higher MsgsSentRate on LP_0 since faster simulation leads to higher message send-out rates. Due to the fact, that much of this simulation work must consequently be rolled back, LP_0 also presents the highest AntiMsgsSentRate and EventsRollbackRate for unthrottled simulation. In the case of controlled simulation, both indicators are considerably lower showing that the LPCC can effectively decimate the number of rollbacks (and their severity) which then leads to better performance. The MsgsRcvdRates and AntiMsgsRcvdRates mirror the values of the corresponding send rates as would be expected. Finally, MemoryConsumption is markedly lower for controlled simulation, confirming that an optimum “knee-point” [20] in the memory usage/event rate relation must indeed exist.

Perhaps the most important aspect of these results is the fact that the performance control method “harmonizes” the

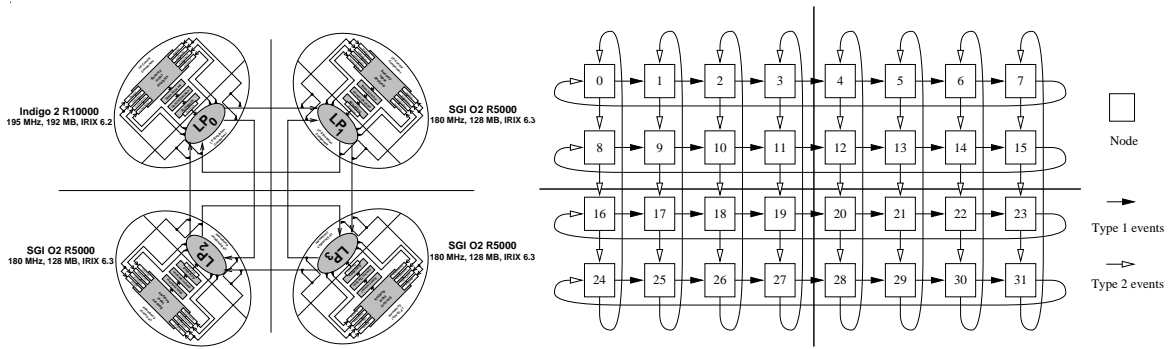


Figure 5. Experimentation Environment (left) and Simulation Model (right)

Method	LP_0	LP_1	LP_2	LP_3	LP_0	LP_1	LP_2	LP_3
	EventRate, Mean				EventRate, Sdev			
Unthrottled	981.6	983.4	981.7	983.3	345.2	334.6	345.1	334.5
Controlled	1282.1	1355.8	1282.0	1355.9	133.7	152.7	133.9	152.6
	SimulationRate, Mean				SimulationRate, Sdev			
Unthrottled	4153.5	2709.1	2162.8	3413.3	412.3	265.3	383.2	308.3
Controlled	3263.3	2362.4	1880.4	2568.8	336.2	242.1	188.3	229.6
	MsgsSentRate, Mean				MsgsSentRate, Sdev			
Unthrottled	743.5	496.4	410.8	629.0	52.6	51.8	62.3	36.8
Controlled	514.2	427.3	359.8	460.2	50.1	43.7	34.2	40.6
	AntiMsgsSentRate, Mean				AntiMsgsSentRate, Sdev			
Unthrottled	501.1	253.7	170.2	388.1	100.5	87.3	46.3	57.8
Controlled	202.0	96.7	47.6	129.6	33.4	26.5	15.7	33.0
	MsgsRcvdRate, Mean				MsgsRcvdRate, Sdev			
Unthrottled	420.6	680.9	691.8	486.4	59.8	55.5	66.8	54.1
Controlled	364.7	494.3	480.2	422.4	33.8	41.7	45.5	43.2
	AntiMsgsRcvdRate, Mean				AntiMsgsRcvdRate, Sdev			
Unthrottled	178.3	438.1	451.2	245.5	47.6	48.4	120.9	85.3
Controlled	52.5	163.6	168.0	91.8	13.5	37.3	24.2	22.2
	EventsRollbackRate, Mean				EventsRollbackRate, Sdev			
Unthrottled	3168.2	1722.1	1177.4	2426.3	255.5	299.7	166.7	172.1
Controlled	1980.4	1006.0	597.7	1212.4	260.8	171.7	133.6	185.8
	MemoryConsumption, Mean				MemoryConsumption, Sdev			
Unthrottled	3833.4	4106.7	2590.8	3612.9	1268.4	1292.1	1292.0	1679.2
Controlled	747.4	899.2	552.9	901.0	184.2	276.0	138.2	291.9

Figure 6. Mean Indicator Values for 100 Hour Experiment

mean indicator values across all machines which ultimately leads to a more balanced TW performance. For example, the AntiMsgsSentRate ranges from 170.2 to 501.1 antimessages per second for unthrottled (a difference of 330.9) while the span is only 154.4 for the performance control method.

Figures 7 through 12 show the eight indicators plotted against real time for two typical simulation runs. The respective group of 4 plots on the left-hand side show the indicator values over time for unthrottled simulation while the right-hand 4 plots show performance controlled simulation. The arrangement of the plots in each group of four is in accordance with Figure 5 with the upper left plot representing LP_0 and the lower right plot LP_3 .

Figure 7 shows the EventRate which exhibits practically identical curves on all machines due to the balanced nature of the simulation model (each LP must commit approximately the same number of events in each GVT epoch). Simulation ends at 164.4 seconds for controlled simulation while uncontrolled simulation takes 290.5 seconds to terminate. The respective SimulationRates in Figure 8 illustrate the relative processing power of the machines. The “shocks” introduced

on LP_2 (lower left plot) are visible for unthrottled simulation whereas for controlled simulation the effect of the shocks on simulation speed is not as evident.

The difference in MemoryConsumption between unthrottled and controlled simulation in Figure 9 shows that the LPCC has effectively “learned” that higher performance can be achieved by limiting the amount of available memory. It should be noted here that the LPCC knows nothing about the “meaning” of the indicators or the effect of actuators. For the LPCC, each system state is purely a point in the n_I -dimensional indicator space. The LPCC simply chooses a similar state which has been previously visited during the course of execution and sets the actuator(s) to those of the target state.

The AntiMsgsSentRates and AntiMsgsRcvdRates in Figures 11 and 12 exhibit complementary behavior for both unthrottled and controlled simulation (indicating also the difference in execution speed on the different machines) but rates are clearly reduced for controlled simulation. (The MsgsSentRates and MsgsRcvdRates (not shown) reflect the same a complementary behavior.) Most markedly, LP_0 and LP_3

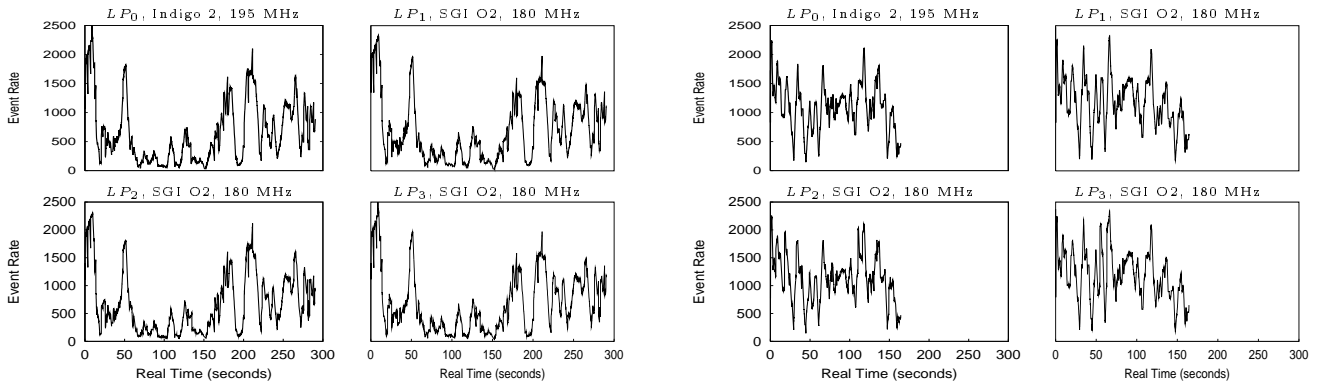


Figure 7. Average Event Rate: Unthrottled (left), Controlled (right)

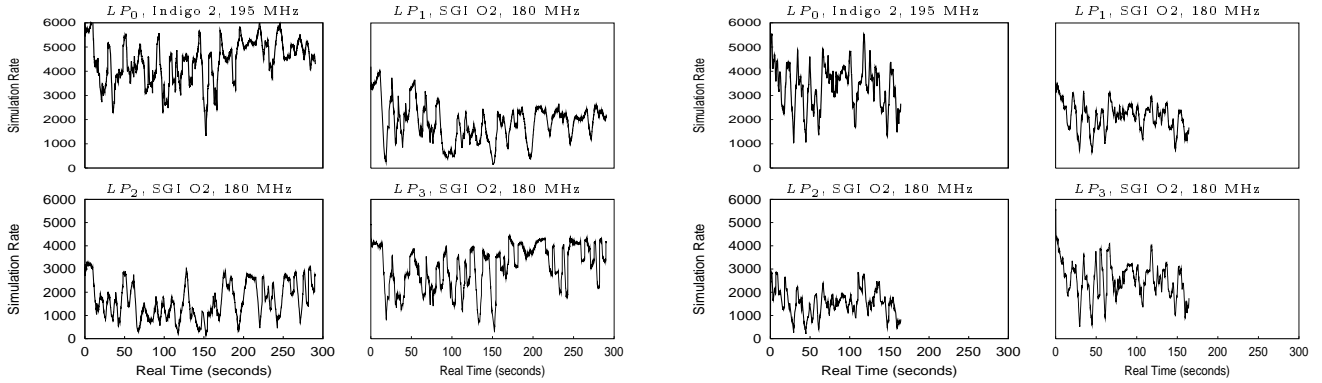


Figure 8. Average Simulation Rate: Unthrottled (left), Controlled (right)

have drastically reduced their `AntiMsgsSentRates` and thus no longer swamp the weakest machine (LP_2) with antimessages. Mirroring the antimessage behavior, the `EventsRollbackRates` (Figure 10) are effectively reduced on all LPs for controlled simulation.

6 Conclusions

Adaptive Time Warp protocols have been in the literature for a couple of years. The basic mechanism to achieve adaptivity is to collect simulation state information, local or global, to make decisions on how to modify performance control parameters “on the fly”. State information is deduced from e.g. the event list history, the past rollback behavior or message arrival patterns, and then this information is used to tune control parameters like e.g. simulation time windows, blocking delays, scheduling quanta or the amount of allocated memory. Common to all known approaches is the implicit assumption that the probability of a state change within the time frame from when the state information was collected and the time instant when the considered control action becomes effective is negligible. In this sense, and particularly for “network of workstation” execution environments, control decisions are always outdated once becoming effective - and thus most likely inadequate.

In this paper we have developed a framework for adaptive TW LPs which are

- (i) *environment aware* in the sense that by monitoring their own simulation behavior, they implicitly develop a model of the background load with which they share

resources (control decisions take the behavior of the environment into account),

- (ii) *self adaptive* in the sense that control decisions are established based on local information only (performance control is managed in a fully distributed way),
- (iii) *pro-active* in the sense that control decisions are established for anticipated future system states, in order to be able to launch the right control actions at the right time (performance control is no longer “re-active”),
- (iv) and *shock resistant* in the sense that an LP, by observing the environment it “lives in”, can react to sudden changes in the availability of system resources (performance control appears to be very responsive to sudden changes in the availability of resources).

As a particular instance of this framework, we have built control components based on a mechanism for memory based throttling to be associated with TW LPs. Technically, a plain TW LP through the insertion of *sensors* and *actuators* in its code is monitored and performance controlled by a *pro-active performance management module*. LP states are characterized by *indicators* derived from the current values of sensors and actuators and recorded for the purpose of analysis and, subsequently, for further LP state forecast. Interpreted geometrically, LP behavior is envisioned as a trajectory through an n-dimensional performance space (spanned by the set of indicator vectors). To record and classify LP (performance) states, a novel incremental clustering technique has been applied which records the trajectory of LP execution by grouping similar LP states

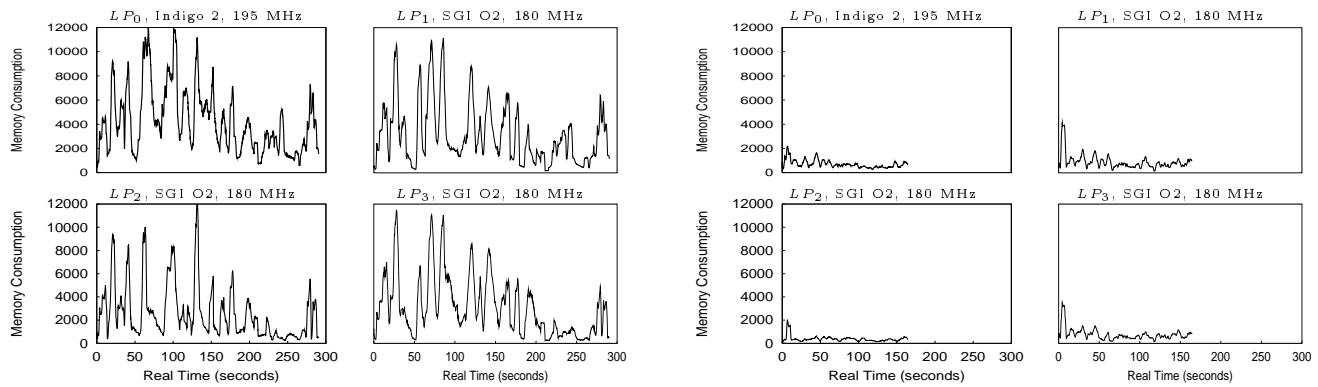


Figure 9. Average Memory Consumption: Unthrottled (left), Controlled (right)

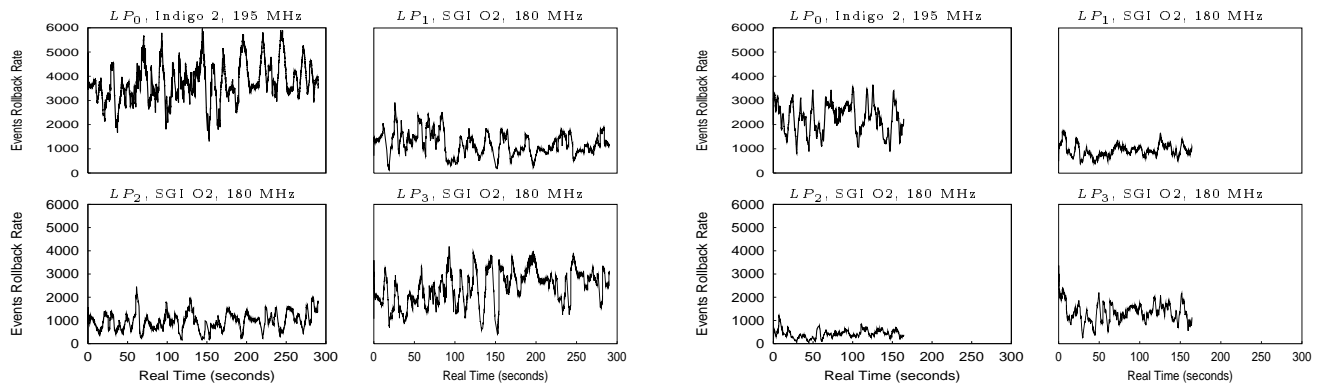


Figure 10. Average Events Rollback Rate: Unthrottled (left), Controlled (right)

together in clusters. In each update step, one cluster is selected which is similar to the current LP state in terms of the indicators, but at the same time promises higher performance. Control parameters are set via the mechanism of actuators to “steer” the LP towards this cluster of better performance.

As a case study, we have partitioned a balanced simulation model on a group of four LPs executing in a NOW with heterogeneous nodes. By intent, we have exposed one of those nodes (and by that the TW LP executing on that node) to repetitive “shocks”, by launching high priority, resource intensive jobs on that node (while TW was executing). The experiments give empirical evidence not only for the LPs being environment aware, self adaptive and pro-active, but also (and presumably most important) can be considered “shock-resistant” due to the very short reaction time to changing background loads.

References

- [1] D. M. Nicol and R. M. Fujimoto, “Parallel simulation today”, *Annals of Operations Research*, vol. 53, pp. 249–286, 1994.
- [2] S. R. Das, “Adaptive protocols for parallel discrete event simulation”, in *Proceedings of the 1996 Winter Simulation Conference*, J. M. Charnes, D. J. Morrice, D. T. Brunner, and J. J. Swain, Eds., 1996, pp. 186–193.
- [3] G. Shao, R. Wolski, and F. Berman, “Modeling the Cost of Redistribution in Scheduling”, in *Proceedings*

of the 8th SIAM Conference on Parallel Processing for Scientific Computing, 1997.

- [4] J.N.C. Arabe, A. Beguelin, B. Lowekamp, E. Seligman, M. Starkey, and P. Stephan, “Dome: Parallel Programming in a distributed computing environment”, in *Proceedings of the 10th International Parallel Processing Symposium (IPPS 96)*. April 1996, IEEE-CS Press.
- [5] M. Cermele, M. Colajanni, and S. Tucci, “Adaptive Load Balancing of Distributed SPMD Computations: A Transparent Approach”, Tech. Rep. DISP-RR-97.02, Dipartimento di Informatica, Sistemi e Produzione, Universita’ di Roma Tor Vergata, 1997.
- [6] J. Gehring and A. Reinefeld, “MARS - A Framework for Minimizing the Job Execution Time in a Meta-computing Environment”, *Future Generation Computer Systems*, vol. 12, no. 1, pp. 87–99, 1996.
- [7] J. Casas, D. Clark, R. Konuru, S. Otto, R. Prouty, and J. Walpole, “Adaptive Load Migration Systems for PVM”, in *Proceedings of Supercomputing 94*. 1994, pp. 390–399, IEEE-CS Press.
- [8] D. Nicol and L. F. Wilson, “Experiments in automated load balancing”, in *Proceedings of the 10th Workshop on Parallel and Distributed Simulation (PADS ’96)*. 1996, pp. 4–11, IEEE CS-Press.
- [9] D. Nicol and L. F. Wilson, “Automated load balancing in speedes”, in *Proceedings of 1995 Winter Simulation Conference (WSC ’95)*, 1995, pp. 590–596.
- [10] D. W. Glazer and C. Tropper, “On process migration and load balancing in time warp”, *IEEE Transactions*

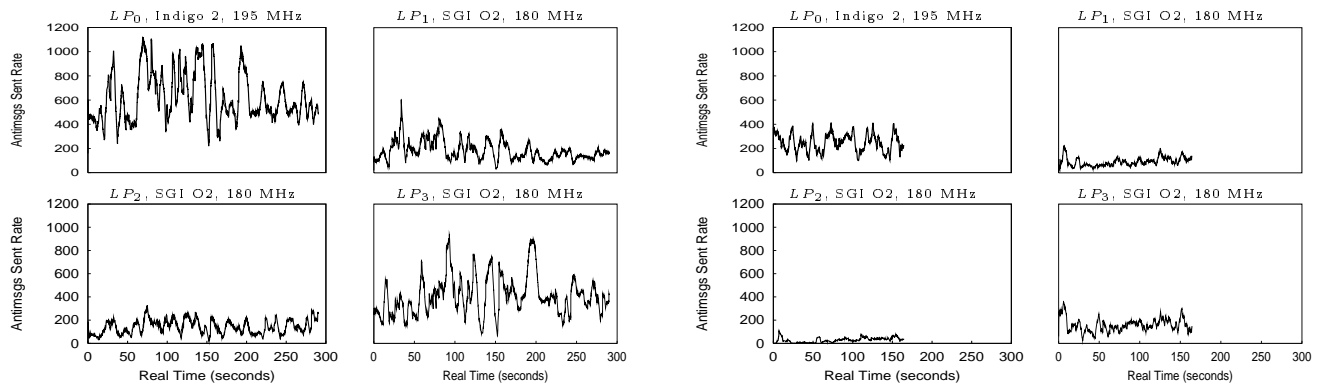


Figure 11. Average Antimsgs Sent Rate: Unthrottled (left), Controlled (right)

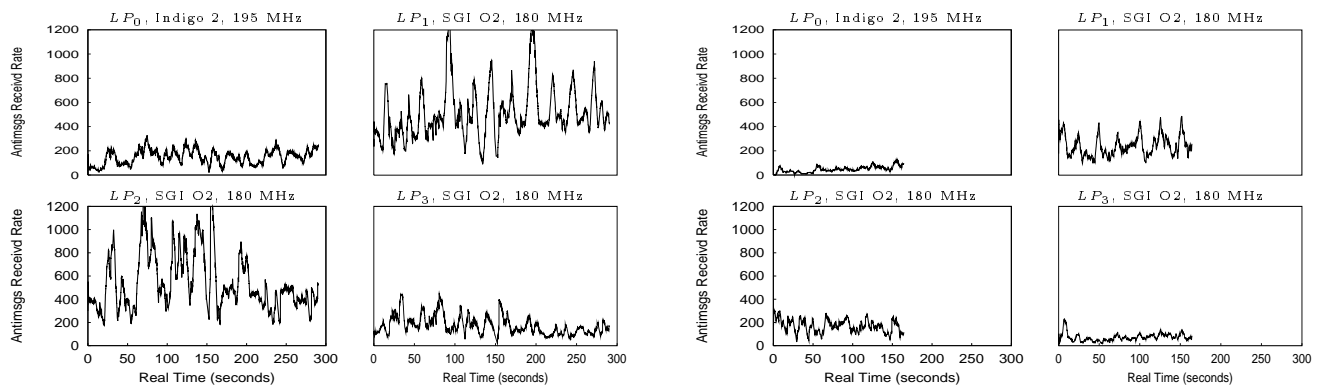


Figure 12. Average Antimsgs Received Rate: Unthrottled (left), Controlled (right)

on Parallel and Distributed Systems, vol. 4, no. 3, pp. 318–327, Mar 1993.

- [11] H. Avril and C. Tropper, “The dynamic load balancing of clustered time warp for logic simulations”, in Proceedings of the 10th Workshop on Parallel and Distributed Simulation (PADS '96). 1996, pp. 20–27, IEEE CS-Press.
- [12] F. Sarkar and S. K. Das, “Design and implementation of dynamic load balancing algorithms for rollback reduction in optimistic pdes”, in Proc. of the 5th Int. Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS'97). 1997, pp. 26–31, IEEE CS-Press.
- [13] A. Boukerche and S. K. Das, “Dynamic load balancing strategies for conservative parallel simulation”, in Proceedings of the 11th Workshop on Parallel and Distributed Simulation (PADS '97). 1997, pp. 20–28, IEEE CS-Press.
- [14] E. Deelman and B. K. Szymanski, “Dynamic load balancing in parallel discrete event simulation for spatially explicit problems”, in Proceedings of the 12th Workshop on Parallel and Distributed Simulation (PADS '98). 1998, pp. 46–53, IEEE CS-Press.
- [15] S. P. Dandamundi and M. K. C. Lo, “A hierarchical load sharing policy for distributed systems”, in Proc. of the 5th Int. Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS'97). 1997, pp. 3–10, IEEE CS-Press.
- [16] D. Arredondo, M. Errecalde, S. Flores, F. Piccoli, M. Printista, and R. Gallard, “Load distribution and balancing support in a workstation-based distributed system”, ACM Operating Systems Review, vol. 31, no. 2, April 1997.
- [17] T.V. Gopal, N.S. Karthic Nataraj, C. Ramamurthy, and V. Sankaranarayanan, “Load balancing in heterogeneous distributed systems”, Microelectronics Reliability, vol. 36, no. 9, September 1996.
- [18] M. J. Zaki, W. Li, and S. Parthasarathy, “Customized dynamic load balancing for a network of workstations”, in Proc. of the IEEE Int. Symp. HPDC., 1996, pp. 282 – 291.
- [19] A. Ferscha, “Parallel and distributed simulation of discrete event systems”, in Parallel and Distributed Computing Handbook, A. Y. Zomaya, Ed., pp. 1003 – 1041. McGraw-Hill, 1996.
- [20] S. R. Das and R. M. Fujimoto, “An adaptive memory management protocol for time warp parallel simulation”, in Proc. of the 1994 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems, Nashville, 1994. ACM, 1994, pp. 201–210.
- [21] M. Charikar, C. Chekuri, T. Feder, and R. Motwani, “Incremental Clustering and Dynamic Information Retrieval”, in Proceedings of the 29th ACM Symposium on Theory of Computing, 1997.
- [22] W. Gropp, St. Huss-Lederman, A. Lumsdaine, E. Lusk, B. Nitzberg, W. Saphir, and M. Snir, MPI: The Complete Reference (Vol. 2) - 2nd Edition, The MIT Press, 1998.