Pro-active Performance Management of Distributed Applications *

Alois Ferscha, James Johnson, Gabriele Kotsis
Universität Wien, Austria
ferscha,johnson,kotsis@ani.univie.ac.at

Cosimo Anglano
Università di Torino, Italy
mino@di.unito.it

Abstract

Self-managing systems able to dynamically re-configure with respect to time-varying workload mixes and changing system resource availability are of growing importance in heterogeneous multicomputer environments. Advanced performance evaluation techniques are needed to induce and assess the impact of such re-configurations where guaranteeing timeliness of reconfiguration activities is particularly challenging. A whole new class of methods supporting "pro-active" adaptivity based on the predicted system state at the re-configuration instant are needed to alleviate the shortcomings of "re-active" systems which bring reconfigurations in effect after the system state has changed. In this paper we argue for self-contained performance management of distributed applications, i.e. integrated performance tuning with the ability to automatically adapt the application behavior to the performance characteristics of the execution environment. Particularly, we study pro-active performance management for distributed simulation based on the Time Warp protocol executing on a network of workstations (NOWs).

1. Introduction

The advances in distributed software technology ease the aggregation of heterogeneous and geographically separated computing resources (e.g. CPU, memory, mass storage, data) into a single computational environment and their convenient use as a distributed execution platform. Yet, developing distributed applications which achieve satisfactory performance remains a challenging task, mainly because of the inadequacy of the traditional performance tuning techniques used for "classical" parallel and distributed programs. The inherent non-predictability of application and system behavior, together with the overwhelming complexity resulting from the co-influence of factors determining performance, makes the standard tuning cycle inappropriate where a programmer repeatedly modifies the application code, executes and monitors it, and tunes its performance by performing further modifications. Such a "static" approach might be appropriate if no changes in application and system behavior could occur, but is evidently inadequate for emerging classes of applications and computing platforms characterized by a high dynamicity. For these dynamic systems and environments, *adaptive* [3, 2] performance tuning strategies appear necessary.

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

Adaptive strategies are able to *react* in response to changes by altering the system configuration, so that the action most appropriate to preserve (or even to increase) the efficiency of the application is taken when a change occurs. Although the whole dynamic load balancing research area [10, 1, 5] has followed this approach, one faces the severe drawback of lack of timeliness in "reactive adaptivity": When an action takes place, the system may be in a state different from the one in which it was when the above action was considered appropriate. The time required to determine that the system has entered a given state, to decide what action is appropriate, and to actually perform that action, may indeed be longer than the time taken by the system to enter a different state. In the worst case, the effect of this action may be exactly the opposite of the desired one, so performance may decrease even further instead of increasing.

To overcome this problem, we introduce in this paper the concept of *pro-active* performance tuning which is based on the principle of predicting the future evolution of the system in order to be prepared to perform a given action at the appropriate time. While a reactive strategy acts only *after detecting* that a given event (e.g. load imbalance) has occurred, a pro-active one is already prepared to act when the above event occurs, or even to take appropriate actions in order to avoid entering an undesirable state. Particularly, we study pro-active performance tuning strategies for distributed simulation based on the Time Warp [9] protocol executing on a network of workstations.

Methodologically in our approach, performance data is periodically relayed from the application to the pro-active performance tuning module through the insertion of sensors into the program code. Based on the metrics delivered by sensors, indicator values are derived which serve to characterize the application's momentary behavior. Program execution can thus be envisioned as a trajectory through the n-dimensional indicator space. In order to record the execution trajectory in a memory and time efficient manner, incremental clustering techniques are applied. The resulting dynamically evolving clusters serve to characterize reachable program states and the coordinates of their centroids serve as guideposts for steering program execution towards higher performance.

The paper is organized as follows: Section 2 presents our pro-active performance management strategy, while Section 3 introduces the concept of execution trajectory in the indicator space. Section 3.1 presents the incremental clustering technique used in the experiments with an MPI-based implementation of Time Warp on a NOW as described in Section 4. Finally, Section 5 concludes the paper and outlines some future research directions.

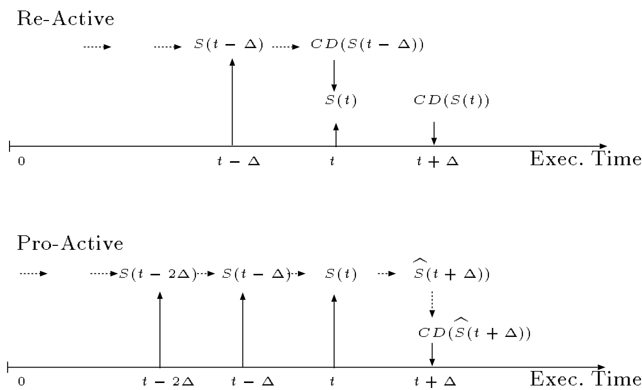

Figure 1. Re-active (top) vs. Pro-active (bottom) Performance Management

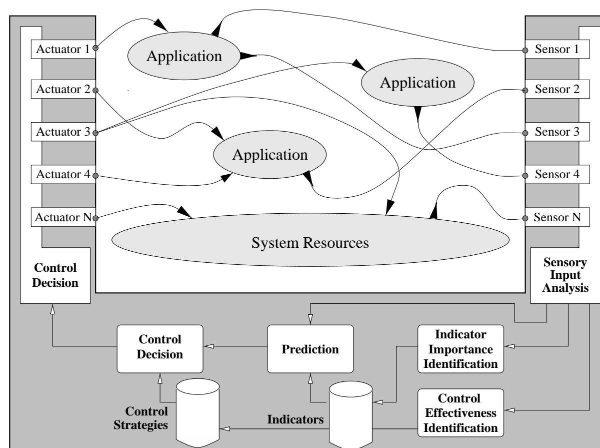

Figure 2. Pro-Active, Self-Contained Performance Management

2. Methodology

2.1. Pro-active Performance Management

Adaptive load balancing policies [1, 5] are either driven by a centralized load coordinator which is in charge of maintaining the global system state, and all other nodes must consult the coordinator for state information, or a distributed scheme where global system state information is gathered in a distributed way, thus avoiding the coordinator bottleneck, but involving state exchange overheads and potentially redundant state collection. Scalability to large system sizes is the main argument in favor of distributed schemes [5].

The main concern of this paper is the way in which state information is sampled (centralized or distributed), and its relation to the timeliness of load control decisions. The traditional load management literature (to the best of our knowledge) follows a scheme of collecting state information $S(t)$ at certain (mostly periodic) instants of time t, and enacts control decisions $CD(S(t))$ based on $S(t)$ (see Figure 1). Since establishing $CD(S(t))$ uses system resources and takes time, $CD(S(t))$ in most cases will not be timely. Assuming Δ time having passed between the time the control decision was made and the time when the control decisions becomes effective, the probability of a system change in the time frame Δ can then be considered a quality parameter for the "timeliness" of control decisions. Most of the literature avoids a discussion of these issues, arguing for this probability being negligibly small. Opposed to these schemes (which we call *"re-active"* because load management is a re-action to a certain system state) we propose *"pro-active"* load management. As explained in Figure 1, in a pro-active scheme, based on a statistical analysis of historical state information, $\ldots, S(t-2\Delta), S(t-\Delta), S(t)$, the future state of the system $\widehat{S}(t+\Delta)$ is estimated for the time when the control decision $CD(\widehat{S}(t+\Delta))$ is supposed to become effective. Depending on the forecast quality, a pro-active scheme will presumably exhibit performance superior to re-active schemes, at least for cases where state changes in the time frame Δ are very likely. While we identify the statistical analysis and forecasting of future system states as challenging topics, particularly the tradeoff between forecast accuracy and the induced computational overhead, we shall focus here on the ability of a pro-active scheme to adapt to heterogeneous load situations, especially to time varying background workloads injected by other applications executing at the same time on the NOW.

2.2. Architecture of a Pro-active System

The pro-active load management scheme that we have developed as a system of automated performance tuning (not involving a human performance analyst) follows the distributed approach, i.e. avoids the performance hazard of a centralized coordinator with respect to growing system size. To avoid state exchange overheads and to preserve scalability, the scheme restricts itself to local state information and enacts control decisions locally.

As outlined in Figure 2, a distributed application, composed of a set of application processes executing remotely on different nodes of a NOW, is embedded in a performance management environment which automatically controls the applications' performance. As a matter of system performance observation, each application process executing on one workstation node is monitored by a dedicated *"sensor"*. Sensors are inserted into a number of concurrently executing application processes to relay state information to the PPMM. Additionally, sensors may deliver information directly from the underlying system resources. Sensory input is reported to a statistical analysis component that extracts the relative importance of sensor values, and by that establishes *"performance indicators"*. A prediction component uses the indicator history together with the sensory input to forecast the next system state, upon which a control decision is made. *"Actuators"* then expose the application process(es) to the control decision, thus closing the performance management loop. Many different application processes can be embedded into this feedback-based control system by just enrolling applications into the performance management environment. The rest of the paper, however, is focussed on the case where a single application process per node is subject to automated performance management.

2.3. Sensors, Indicators and Actuators

In order to collect execution performance and state data from the application, *sensors* are inserted in the program code. In its simplest implementation, a sensor is defined as a global variable which is assigned the current value of

its corresponding performance/state metric at appropriate points in the program code. Although this appears sufficient for most cases, more complex sensor implementations may require a more extensive data collection strategy and/or additional preprocessing of performance/state metrics before assigning a value to a sensor variable. The current implementation of our pro-active performance management module (PPMM) allows for *point sample sensors* which register the *momentary* value of a performance/state metric and *cumulative sensors* which cumulate a specific metric (e.g. occurrence of an event).

Assuming n_S sensors have been defined, the update of the PPMM is invoked with the function $\mathrm{PPMM_update}(S_1, S_2, \ldots, S_{n_S})$. Based on the momentary value of the sensors, the PPMM then calculates the value of n_I *indicators* which are used the characterize the current state of program execution. In the most general case, the value of an indicator may be a function of any number of sensor values $I_i = f(S_1, S_2, \ldots, S_{N_S})$.

Since cumulative sensors continuously increase as execution progresses and point sample sensors deliver values which may fluctuate considerably from one sample to the next, the performance module applies a smoothing function to the sensor values in order to obtain an indicator metric which accurately reflects the current tendency of the sensor which is calculated based on observations in a moving window $[t-\Delta, t]$. For cumulative sensors, the performance module uses the *rate of increase* of the sensor (in units per second) as the corresponding indicator value and for point sample sensors, the *mean value* in the moving window. Assuming n observations of the sensor S_k in the current window, the value of the indicator I_k at sampling time t is thus calculated as: $I_k(t) = \frac{S_k(t) - S_k(t-\Delta)}{\Delta}$ for a cumulative sensor and $I_k(t) = \frac{\sum_i S_k(t_i)}{n}$ $\forall i : \; t - \Delta \leq t_i \leq t$ for a point sample sensor.

In order to characterize the current performance of the executing application, one indicator is designated as the *performance indicator*. As with the other indicators, the value of the performance indicator may be a function of any number of sensors. The task of the PPMM is to maximize the value of this indicator by setting the values of *actuators* according to the control decision based on the analysis of the indicator data.

As opposed to sensors which simply relay state information from the executing application to the PPMM, *actuators* are inserted at appropriate points in the program source which *set* the values of execution program parameters which affect the run-time behavior of the application directly. Except for the fact that the value of an actuator can be modified, actuators are treated in the same fashion as sensors by the PPMM. As with sensors, indicator values may also be derived from the momentary values of the n_A actuators: $I_i = f(S_1, \ldots, S_{n_I}, A_1, \ldots A_{n_A})$. In addition to the indicator function, however, inverse functions must be given that allow the actuator values to be calculated based on a given indicator value I_i: $A_k = f^{-1}(I_i, S_1, \ldots, S_{n_I})$.

3. Execution Trajectories in Indicator Space

In a geometric interpretation, each vector of sampled indicator data $\vec{p}(t) = (I_1(t), I_2(t), \ldots, I_{n_I}(t))$ can be viewed as a point in an n_I-dimensional vector space. The sequence of sampled points gathered during the course of program execution thus describes a *trajectory* through the n_I-dimensional *indicator space*. With each point of the trajectory representing a system state that has been visited, the entire execution trajectory can be viewed as the ordered set of all system states which have been recorded during the course of program execution.

Since the objective of the PPMM is to identify reachable states similar to the current state and to steer program execution towards them, a method which classifies similar states appears adequate for "storing" the trajectory data. Clustering fulfills these requirements in that similar states can be grouped together, the centroid of the resulting cluster then serves as a representative for all points within the cluster. Static clustering methods, however, are not feasible due to the dynamic nature of the data since new points must be repeatedly added to the data set and clustered.

3.1. Incremental Clustering

A viable alternative to static clustering is incremental clustering which has been applied successfully in the domain of information retrieval [4].

The algorithm implemented here is a variation of the *doubling algorithm* [4] with the major difference being the manner in which the coordinates of cluster centroids are calculated after the merging of a new point or the merging of two clusters. The doubling algorithm assumes a collection of n_C clusters C_i with centroids $\vec{c}_i$. In the initialization phase, the first n_C data points become the centroids of the n_C clusters.

In each update step, the current vector of indicator data $\vec{p} = (I_1(t), I_2(t), \ldots, I_{n_I}(t))$ is considered as the centroid of a new cluster C_{n_C+1}. Since only n_C clusters are allowed, the new point must now either be merged with an existing cluster or two existing clusters must be merged to form a single cluster thus allowing the creation of a new cluster with centroid $\vec{p}$.

In order to provide for maximum intercluster distance, the minimum distance between all $n_C + 1$ centroids must be determined. Given that in previous update steps the pairwise distances between the existing n_C clusters have been calculated, sorted and stored on the heap, it remains to calculate the distances between the n_C cluster centroids and new data point $\vec{p}$: $d_{n_C+1,j} = ||\vec{p} - \vec{c}_j||$, $j = 1 \ldots n_C$ and determine the nearest cluster C_{j_0} with $d_{n_C+1,j_0} = \min_j d_{n_C+1,j}$.

If the minimum distance from the new data point to an existing cluster d_{n_C+1,j_0} is less than the minimum intercluster distance d_{j_1,j_2} (retrieved from the heap) the new point is merged with cluster C_{j_0}. Contrary to the doubling algorithm which chooses arbitrarily between C_{j_0} and the new data point and sets the centroid of the resulting cluster to $\vec{c}_{j_0}$ or $\vec{p}$ respectively, the new centroid of cluster C_{j_0} is calculated as: $\vec{c}^{\,'}_{j_0} = \frac{h_{j_0} \cdot \vec{c}_{j_0} + \vec{p}_i}{h_{j_0}+1}$ and $h^{'}_{j_0} = h_{j_0}$ where h_{j_0} ("hit count") is the number of points which have been merged with cluster C_{j_0} in previous update steps. The new centroid $\vec{c}^{\,'}_{j_0}$ now minimizes the intracluster variances with respect to the h_{j_0} points it represents. The $n_C - 1$ intercluster distances between cluster C_{j_0} and the other clusters are removed from the heap, recalculated and reinserted. The maximum intercluster distance provides an upper bound for the maximum cluster diameter.

On the other hand, if $d_{j_1,j_2} < d_{n_{C+1},j_0}$ then clusters C_{j_1} and C_{j_2} are merged into $C^{'}_{j_1}$ and $\vec{p}_i$ becomes the centroid of a new cluster $C^{'}_{j_2}$. The respective centroids of the clusters are calculated as $\vec{c}^{\,'}_{j_1} = \frac{h_{j_1} \cdot \vec{c}_{j_1} + h_{j_2} \cdot \vec{c}_{j_2}}{h_{j_1} + h_{j_2}}$ with $h^{'}_{j_1} = h_{j_1} + h_{j_2}$ and $\vec{c}^{\,'}_{j_2} = \vec{p}_i$ where $h^{'}_{j_2} = 1$. As before, the intercluster distances between clusters C_{j_1} and C_{j_2} and the other clusters ($2n_C - 3$ distances) are removed from the heap, recalculated and reinserted.

3.2. Performance Forecasting Techniques

With each update of the PPMM, a decision must be made to assign the actuators values which promise higher performance based on the current system state. The setting of actuators, as stated previously, is analogous to a displacement in the subspace spanned by the actuators. Given the clustering information, which represents the execution trajectory in a compact form and thus a set of reachable states, the PPMM must choose to move (in the actuator subspace) towards those clusters which are reachable and have a higher performance component.

In a geometric interpretation, the set of states which can be reached from the current state is the set of points resulting from the intersection of the execution trajectory with the actuator subspace. Each of these points is *"similar"* to the current state (in terms of the given indicators) except for the value of their actuator components. All other components of their coordinates ("pure" sensory indicators) are identical to those of the current state. Since, however, the exact execution trajectory cannot be reconstructed and the points of intersection calculated, the decision as to which direction to move is reduced to finding those clusters with better performance components which intersect with the actuator subspace.

Once a cluster has been selected, the PPMM steers the execution behavior towards that cluster. It sets the current actuator values to those of the selected cluster and returns control to the application. Selecting the most similar cluster with a higher performance component is crucial. In the following case study, we present a blend of such methods.

4. Performance Management Case Study

4.1. Optimistic DDES with Time Warp

Distributed discrete event simulation (DDES) using the optimistic Time Warp protocol appears as an irregular application with complex, data-dependent execution behavior, and dynamic, time varying resource requirements [7]. The variety of performance influencing factors makes it impossible to identify a single "typical" behavior or resource usage pattern [8]. The performance delivered by a distributed simulation program like Time Warp is hard to predict, since (i) the machines are typically heterogeneous, respond differently from each other to the application requests, and often act in an uncorrelated way (e.g. no coordinated process scheduling is performed); (ii) the computing and communication resources are shared by many users that generate background load exhibiting potentially large performance fluctuations; (iii) several layers of software (e.g. operating system, run time support for parallel APIs, etc.) interact asynchronously on each machine. Neither high processor speed nor fast communication can on its own be considered

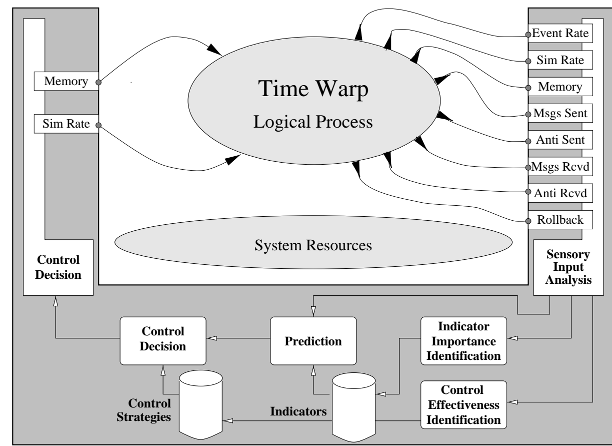

Figure 3. Actuators and Indicators for Optimistic Distributed Simulation

to favor high simulation speed, it is rather the degree of fit among the computational and communication requirements of the simulation model and the hardware communication to computation speed ratio that is responsible for good performance.

In this work we study the distributed execution of Time Warp logical processes (LPs) under the performance supervision of the PPMM environment (Figure 3). State and performance characteristics of a Time Warp LP are sampled at runtime via the sensors listed in Table 1. Time Warp is known require a *sufficient* amount of memory to be operable. Providing just the minimum of memory may cause the protocol to execute fairly slow, due to frequent memory stalls necessitating costly *fossil collection* and *global virtual time* (GVT) calls. Fossil collection as a mechanism to recover memory no longer used due to the progress of GVT may however fail to reclaim enough memory to proceed with the simulation. In this case, additional memory must be freed by cancellation mechanisms like *message sendback, cancelback* or *artificial rollback*. With a decreasing amount of available memory, therefore, absolute execution performance also decreases due to more frequent cancelbacks until it becomes frozen at some point. Increasing the amount of available memory will reduce the cancellation frequency, such that absolute performance should have positive increments. But this at the same time will increase the rollback frequency, such that eventually the rollback overhead will exceed the gain from reduced cancellation overheads [6]. Indeed, one would like to run Time Warp in the area of the "knee-point" of absolute performance, and a successive adaptation to that optimal point is desired. The task of the actuator $\mathrm{MemoryLimit}$ is now to increase or decrease the amount of memory allowed to the LP, such that PPMM will implicitly drive the execution towards the desired optimal performance configuration – even if changing background load causes other throttling effects within the LP.

4.2. Experiment Setup

Our experimentation environment is a heterogeneous cluster of four Silicon Graphics Workstations within the department's LAN (see Figure 4). A fully functional, general purpose Time Warp simulator has been implemented based on MPI, and executes a simulation model decomposed into

	Sensor/Actuator	Type	Description
S_1	CommittedEvents	CS	total number of events committed
S_2	SimulatedEvents	CS	total no. of events simulated
S_3	EventsUsed	PS	momentary no. of events in use
S_4	MsgsSent	CS	total no. of (positive) messages sent
S_5	AntiMsgsSent	CS	total no. of antimessages sent
S_6	MsgsRcvd	CS	total no. of (positive) messages received
S_7	AntiMsgsRcvd	CS	total no. of antimessages received
S_8	EventsRollback	CS	total no. of events freed during rollbacks
A_1	MemoryLimit	PS	momentary no. of allowed events

	Indicator	Description
I_1	EventRate	number of events committed per seceond (perf. indicator)
I_2	SimulationRate	no. of events simulated / sec.
I_3	MsgsSentRate	no. of (positive) messages sent / sec.
I_4	AntiMsgsSentRate	no. of antimessages sent / sec.
I_5	MsgsRcvdRate	no. of (positive) messages received / sec.
I_6	AntiMsgsRcvdRate	no. of antimessages received / sec.
I_7	EventsRollbackRate	no. of events freed during rollbacks / sec.
I_8	MemoryConsumption	average no. of events in use

Table 1. Time Warp: Sensors and Indicators

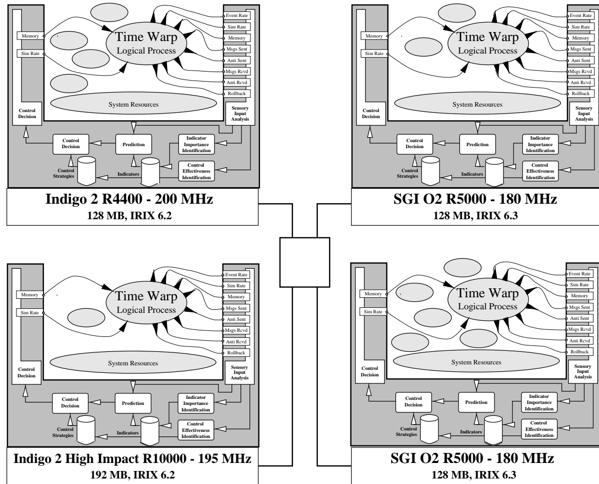

Figure 4. Experimentation Environment

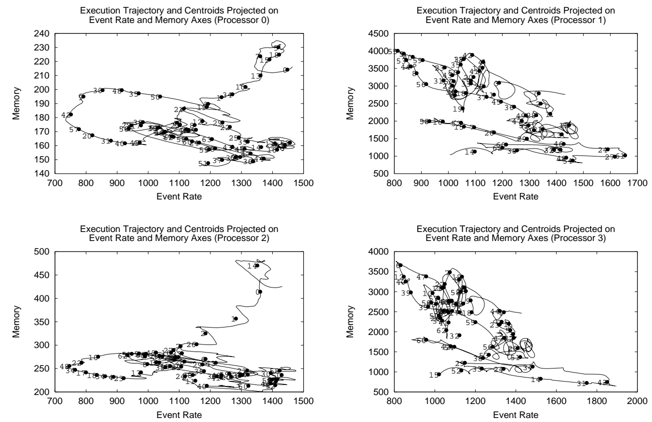

Figure 5. Program Execution Trajectory: Event Rate vs. Memory Consumption

four LPs in our experiments. The structure of the simulation model guarantees balanced load among the four LPs, and the communication structure among LPs adheres to a ring topology.

The eight sensors outlined in Table 1 have been inserted into the Time Warp code. The point sample sensor (PS) as well as each of the cumulative sensors (CS) are transformed into a corresponding indicators (see Table 1). Since the number of events committed per unit CPU time (the event rate) relates to the useful amount of simulation work, this indicator was chosen to be the performance indicator to be maximized by the PPMM. The PPMM will control the actuator (number of events allowed) by limiting the amount of memory available for storing events. Particularly, the throttle is implemented by $m+2*s$, where m and s represent the mean value and standard deviation of memory currently in use. Limiting the available memory will prevent an LP to generate new events once the bound has been reached, thus forcing it to pause its simulation.

Figure 5 shows for each of the four LPs executing in our experimentation setup a two-dimensional projection of the execution trajectory in the event rate/memory plane. The pictures clearly express the need for a performance management strategy able to dynamically and individually control the actuator: although working on balanced subparts of the simulation model, the different LPs evolve in quite different indicator trajectories. The reason for this is twofold: on the one hand, the machines are heterogeneous with respect to CPU power and memory access time. On the other hand, there are unpredictable fluctuations in the background load on each machine and in the network.

Figure 6 left gives a more detailed view of the execution trajectory of one machine plotted in a three-dimensional subspace spanned by the performance indicator (event rate), the indicator derived from the actuator (the memory consumption) and the rate of received messages. The schematic diagram to the right shows a subset of the clusters. Assuming that the system is currently in a state represented by the grey cluster, the PPMM has to select a similar cluster with a higher event rate.

A straightforward method is to try to reach the nearest cluster (wrt. the Euclidean distance) in the execution trajectory which gives better performance, in the example: cluster 1. But attempting to “move” towards the nearest cluster is not always a guarantee for actually reaching it: since MemoryLimit is the actuator of choice, navigation is limited to the corresponding performance dimension. Modifying the level of MemoryLimit therefore induces a move towards the targeted cluster, but only with respect to the memory consumption dimension (perhaps not with respect to other indicators).

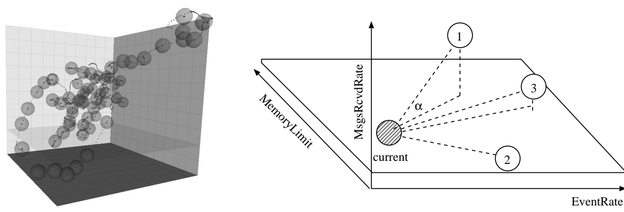

Figure 6. Three-Dimensional Visualization of the Execution Space

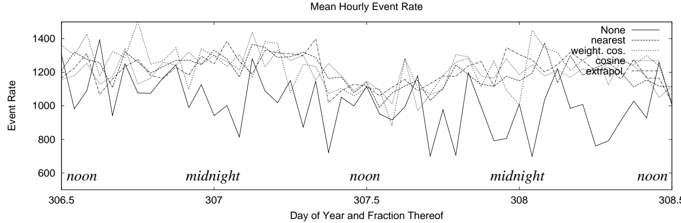

Figure 7. Two-Day Window from the Five-Day Experiments

An improved method would consider the distance of potential target clusters to the performance plane (MemoryLimit/EventRate), in order to avoid approaching clusters distant from that plane. The cosine of the angle α of the vector connecting the current and potential target cluster with the MemoryLimit/EventRate plane can serve as the selection criterion here. The higher $\cos\alpha$, the closer the cluster is to the current plane, with 1 being the optimum value (a cluster in the same plane with better event rate). Using this rule, cluster 2 would be chosen.

Yet another improvement would be to try accept a less optimum cluster with respect to the cosine criterion, if it gives a significantly higher improvement in the event rate. This weighted cosine method would head for cluster 3.

Finally, a cluster may be reached, where no cluster with better performance can be found. But this does not necessarily mean, that the optimum performance level has yet been found. To avoid being trapped in such a sub-optimum solution, an extrapolation can be applied. If the PPMM cannot identify a cluster with higher event rate, it tries to steer the execution towards a fictive cluster extrapolated from the movement from the previous to the current cluster.

4.3. Results

To investigate the effects of the four prediction strategies, nearest, cosine, weighted cosine, and extrapolation, measurements were conducted on the SGI network of workstations. A time period of five consecutive days (Saturday to Wednesday) has been chosen to obtain a good coverage of periods with varying (department) background load.

Figure 7 shows the rate of committed events per second (EventRate) as an per hour average over a time window of two days within the observation period. The high event rates at the beginning are explained by a low background load, day 306 was a Sunday. At this background load level, all methods perform quite well, even plain Time Warp (the solid line labeled None) yields high event rates. The drops in performance during the night hours are caused by maintenance activities which introduced heavy network load (e.g. a backup of user directories to a network attached tape), while the drops in performance during working daytime is caused by users working on the machines. When background load varies, Figure 7 clearly reflects the ability of the investigated methods to cope with these changes and to maintain a good performance level while plain Time Warp suffers from significant drops in performance. We thus have empirical evidence that the pro-active management techniques are able to optimize the overall performance by controlling individually the simulation speed of each LP.

Table 2 gives a more detailed view of the mean values and standard deviations for selected indicators. As expected, the method of using a weighted cosine obtained best results, but at the costs of a slightly higher variability. The mean simulation rate decreases when memory throttling is applied, but the mean event rate increases simultaneously corresponding to a higher percentage of useful work. Thus, all pro-active techniques were able to utilize the available processing power more efficiently.

As memory consumption is the indicator derived from the actuator, we would expect this value to reduce significantly when the control mechanism is applied. In fact, the reduction is about 50 % for all methods. What is interesting to note is, that the weighted cosine (which obtained best results with respect to the performance indicator) allows for the highest memory consumption (on average). This indicates, that the other methods are too cautious in freeing memory.

The number of messages sent per second is only slightly reduced, because the structure of the simulation model requires frequent exchange of messages among LPs. A similar observation holds for the rollback rate. All techniques were able to reduce the number of rollbacks, but the best technique (weighted cosine) has higher roll back rates. Time Warp needs rollbacks as a kind of synchronization mechanism, but they have to occur at the right moment, i.e. neither slowing down the simulation progress too much nor allowing a particular LP to progress too far ahead in the simulation future. It seems, that the PPMM on each LP was able to find just the right balance among all LPs when applying the weighted cosine method.

5. Conclusions and Future Work

We have developed a framework and the corresponding implementation of a *pro-active performance management* environment for distributed applications. To avoid the possible bottleneck which a centralized load coordinator could induce, a distributed scheme has been followed in which state information collection, analysis and control decisions are all conducted locally.

Through the insertion of *sensors* and *actuators*, program execution is monitored and controlled by the *pro-active performance management module* (PPMM). System states are characterized internally by *indicators* derived from the current values of sensors and actuators and recorded for the purpose of analysis and subsequent performance prediction. In a geometrical interpretation, program execution is envisioned as a trajectory through an n-dimensional vector space described by the sequence of vectors of periodically sampled indicator data. In order to record and classify reachable system states, a novel incremental clustering technique has been applied which records the trajectory of program execu-

Method	Mean EventRate				Std. Dev. EventRate			
	Proc 1	Proc 2	Proc 3	Proc 4	Proc 1	Proc 2	Proc 3	Proc 4
none	1029.9	1009.5	1030.2	1009.4	351.4	332.7	351.4	332.7
nearest	1218.0	1209.0	1218.3	1209.5	163.9	144.7	164.0	145.1
cosine	1220.8	1196.8	1221.2	1197.2	168.8	137.8	168.6	137.9
weight. cos.	1246.0	1232.1	1246.3	1232.4	243.4	227.6	243.4	227.6
extrapolate	1218.5	1194.5	1218.8	1194.8	163.9	141.3	163.9	141.5
	Mean SimulationRate				Std. Dev. SimulationRate			
none	4331.3	2769.6	2302.4	2965.9	569.6	263.1	398.0	383.0
nearest	4031.5	2340.3	2047.6	2469.0	289.7	213.7	236.9	213.8
cosine	4042.8	2344.6	2086.2	2547.9	307.7	205.0	242.5	198.6
weight. cos.	4341.0	2751.4	2248.3	2802.3	377.8	240.3	290.3	248.7
extrapolate	4049.8	2394.9	2106.2	2495.6	292.4	216.3	244.4	214.0
	Mean MemoryConsumption				Std. Dev. MemoryConsumption			
none	4386.6	3491.8	3401.6	3057.3	1422.3	1215.1	1461.0	1397.9
nearest	2152.7	1211.5	1811.0	1118.7	505.2	499.1	481.4	520.4
cosine	2167.6	1251.5	1790.6	1152.0	524.5	549.9	465.6	557.7
weight. cos.	2838.4	2229.8	2335.8	2034.6	949.8	963.5	819.5	975.3
extrapolate	2190.6	1235.5	1812.6	1127.2	493.5	516.8	433.2	532.7
	Mean MsgsSentRate				Std. Dev. MsgsSentRate			
none	780.5	524.4	433.4	558.1	75.0	53.7	67.7	54.2
nearest	656.7	421.6	385.2	448.6	46.5	38.3	43.6	36.6
cosine	664.1	420.3	388.5	461.9	49.4	37.9	44.8	33.2
weight. cos.	725.5	500.4	418.2	515.1	56.6	45.3	52.4	41.0
extrapolate	665.2	430.4	391.7	453.1	47.6	38.2	44.7	36.6
	Mean EventsRollbackRate				Std. Dev. EventsRollbackRate			
none	3297.0	1757.1	1267.8	1953.3	413.2	314.5	243.1	213.4
nearest	2811.3	1130.8	827.1	1258.9	265.9	171.6	167.4	199.3
cosine	2819.7	1147.2	863.0	1350.3	287.4	170.6	176.8	196.7
weight. cos.	3091.4	1517.4	998.5	1567.9	341.3	217.9	182.3	244.4
extrapolate	2828.9	1199.8	885.1	1300.3	267.6	184.4	170.8	216.3

Table 2. Observed Time Warp Performance Indicators

tion by grouping similar system states together in clusters. In each update step of the PPMM, one cluster is selected which is similar to the current state in terms of the indicators and at the same time promises higher performance. Program execution is steered towards this cluster by setting the actuator values to those of the selected cluster. A case study using distributed simulation based on the Time Warp protocol executed on a heterogeneous network of workstations was presented providing empirical evidence for the validity of the presented approach. Even for time varying and unpredictable background loads, the PPMM was able to automatically find the performance optimal resource consumption policy of the distributed execution environment, and readjust this policy if necessary.

Several aspects of the approach remain the subject of future work. Owing to the wide range and possible variations of incremental clustering algorithms, an investigation into the effect of the clustering algorithm on performance will be conducted. Furthermore, methods of cluster selection, being critical for the effectiveness of the approach, are currently being studied in more detail. The potential impact of our case study with respect to parallel and distributed simulation techniques is obvious: further experiments are planned to help developing a “shock-resistant” Time Warp implementation.

References

[1] 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, 31(2), April 1997.

[2] J. Casas, D. Clark, R. Konuru, S. Otto, R. Prouty, and J. Walpole. Adaptive Load Migration Systems for PVM. In Supercomputing 94, pp. 390–399. IEEE-CS Press, 1994.

[3] M. Cermele, M. Colajanni, and S. Tucci. Adaptive Load Balancing of Distributed SPMD Computations: A Transparent Approach. Technical Report DISP-RR-97.02, Dip. di Informatica, Sistemi e Produzione, Universita' di Roma, 1997.

[4] 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.

[5] S. P. Dandamundi and M. K. C. Lo. A Hierarchical Load Sharing Policy for Distributed Systems. In MASCOTS'97, pp. 3–10. IEEE CS-Press, 1997.

[6] S. R. Das and R. M. Fujimoto. An Adaptive Memory Management Protocol for Time Warp Parallel Simulation. In ACM Sigmetrics 1994, pp. 201–210. ACM, 1994.

[7] A. Ferscha. Parallel and Distributed Simulation of Discrete Event Systems. In A. Y. Zomaya, editor, Parallel and Distributed Computing Handbook, pp. 1003 – 1041. McGraw-Hill, 1996.

[8] J. Ferscha, A. Johnson and St. Turner. Early Performance Prediction of Parallel Simulation Protocols. In Y.M. Teo, W.C. Wong, T.I. Oren, and R. Rimane, editors, Proceedings of the 1st World Congress on Systems Simulation, WCSS'97, Singapore, Sept. 1–3, 1997, pp. 282–287. IEEE CS Press, 1997.

[9] R. M. Fujimoto. Parallel Discrete Event Simulation. Communications of the ACM, 33(10):30–53, October 1990.

[10] T.V. Gopal, N.S. Karthic Nataraj, C. Ramamurthy, and V. Sankaranarayanan. Load balancing in heterogeneous distributed systems. Microelectronics Reliability, 36(9), September 1996.