To appear in Proc. 2nd Int'l. Conf. Systems Integration, IEEE, 1992.

Adaptive Load-Balancing Strategies for Distributed Systems

Pankaj Mehra and Benjamin W. Wah
University of Illinois

Abstract

This paper describes SMALL, a system for learning load-balancing strategies in distributed computer systems. The load balancing problem is an ill-posed optimization problem because its objective function is ill-defined. Realistic state-space representations of this problem do not satisfy the Markov property: evaluation of a state depends on past states and decisions. Experimentally feasible learning environments for load balancing exhibit delayed, evaluative feedback. Such aspects complicate the learning of strategies for load balancing. SMALL uses comparator neural networks *for learning to compare objective-function values of states resulting from a set of alternative moves. The problem of learning from delayed evaluative feedback, also called the* credit-assignment problem *of reinforcement learning, has been solved only for Markovian problems. This paper presents a novel credit-assignment procedure suitable for load balancing and other non-Markovian learning tasks.*

I. Introduction

Problems of decision, control, and optimization are formulated as decision-making tasks. Computer programs for solving these tasks use problem-solving strategies to search for solutions; a solution is described by either a sequence of problem-solving operators or, equivalently, as a sequence of states resulting from operator application. Good solutions lead to desirable states quickly; similarly, good strategies lead to good solutions quickly and efficiently. Intelligent problem solvers need good strategies which can be either supplied by a human programmer or automatically learned by a strategy-learning system; in this paper, we address the latter case.

Management of resources in large-scale distributed systems is a source of several complex decision-making tasks. We are especially interested in the following *load balancing* problem: Given a (distributed) stream of independent jobs arriving at various sites, to determine dynamically, for each job, the site at which that job will finish in the shortest time. Each job competes with others at its site of execution for *resources of contention*, such as CPU, memory, and I/O. The net resource utilization of the jobs at each site constitutes that site's "background workload". The completion time of a job on a site depends upon (i) the loads on the resources of contention at that site; (ii) the amount of service required by that job from each resource; and, (iii) the rate at which the hardware devices of that site can service resource requests. The resource-usage patterns due to background workload vary from one installation to another; the exact service demands of jobs are rarely known ahead of time; and, hardware devices may be upgraded or replaced from time to time. Under such circumstances, manual design of load-balancing strategies that work well for every installation and at all times is not only costly but also impractical. Therefore, load balancing is a good candidate for application of automated strategy-learning techniques.

Load-balancing strategies have two parts [1]: (i) *decision metrics*, which convey status information about the load on each site; and, (ii) *decision policy*, comprising rules for deciding where jobs should be executed. This paper is about SMALL, a system that learns new decision metrics as well as policies. It uses the measured completion times of jobs as feedback during learning. To measure completion times under realistic and reproducible background workloads, SMALL uses the DWG experimentation environment described elsewhere [10].

For learning decision policies, the measured completion times must be normalized into *feedback signals* (scalar values indicating how well the current strategy is performing). Feedback signals are then used in modifying the current strategy in such a fashion that the modified strategy can be expected perform better than the original strategy. Policy learning requires rational solutions to the problems of *evaluation* (generation of feedback signals) and *credit assignment* (assimilation of feedback signals). Our approach differs from the traditional approach to these problems exemplified by the work of Sutton et al. [11,21]. For learning intelligent decision metrics, we draw upon the work of Hwang et al. [8] in the area of load balancing and the work of Tesauro et al. [22,23] in the area of machine learning.

II. Load balancing

Background workload is generated by processes, and can be defined in terms of the individual loads on various resources of contention. Principal among these resources are (i) computational resources such as the CPU and floating-point processor; (ii) communication resources such as networks and buses; (iii) primary storage resources such as RAM, and cache; and, (iv) secondary storage resources such as magnetic disks. Primitive measurements of load on each resource include queue length (the number of processes blocked waiting for each resource) as well as amounts and rates of utilization of resources. All of these are highly dynamic quantities. Associated with each resource is a fixed but unknown metric representing its service rate (the rate at which the resource can service requests issued by the processes). Of these metrics, only the utilization information can be systematically varied for

Research supported by National Aeronautics and Space Administration Contracts NCC 2-481, and NAG-1-613.

the purposes of experimentation. For this and other reasons explained later, we adopt utilization-based decision metrics.

Time-series analysis of utilization-based metrics (Appendix) reveals that these are independent quantities with significant autocorrelation [5]. Also, since they represent the current load on resources of contention, their values at the time of introducing a job influence its completion time. Other determinants of completion time, such as the numbers and rates of service requests issued by a job on various resources, are nearly impossible to estimate in practical situations. The benefits of migrating an incoming job to a remote site are therefore impossible to estimate in absolute terms. A realistic alternative is to estimate the *relative* benefits of migration, using the loads on each site's resources as the only determinants of relative completion time [8].

Traditional load-balancing algorithms [4,25] make several simplifying assumptions; (i) completion time of a job is not affected by the loads on resources other than the CPU; (ii) some moving average of CPU queue length is a significant determinant of completion time; and, (iii) simple decision rules, such as always sending a job to the least loaded site, can be determined a priori and perform well. In reality, CPU-based decision metrics are appropriate only for purely computational tasks; their values are not useful for comparing alternative sites for running a given job. Good decision procedures take into account estimated benefits as well as measured costs of past migrations. Most of the information used for designing decision metrics and decision rules is installation-specific, and not available to an operating system designer. However, with experience, good metrics and policies can be 'learned' by observing the completion times of test jobs under a variety of loading conditions and scheduling policies. SMALL provides the basic environment for collecting such data and using it for experimentally improving both policies and metrics.

III. Strategy-learning tasks

Given a problem P, a problem solver S, and a learning environment E, a *strategy-learning system* L uses feedback from E for optimizing the strategy used by S in solving P. Various characteristics of P, S, and E affect the design of L. P, the load-balancing problem, is defined as follows: Minimize an objective function O defined in terms of the completion times of test jobs by using a parameterized decision process D(W) to decide where to execute each job. D(W) is a distributed decision process, with one component per site deciding where each job arriving at that site is executed. Let $D_M(W_M, L, a)$ denote the decision rule for site M, where $L(t)$ is the information about background workload available on site M at time t; and, $a \in A$, where A is the set of possible actions that can be performed upon a job arriving on site M at time t.

The problem solver S internally represents O, A, $L(t)$, and D(W). It implements the actions of A; manages the execution of jobs; collects load-related information, such as measurements of CPU utilization and network traffic; and, computes or measures goal-related information, such as completion times of jobs. D(W), the load balancing policy, may be implemented either structurally (for example, using a neural network) or procedurally (for example, by a program that explicitly stores its parameters W in a configuration file).

The learning environment E reacts to the decisions made by S; it produces feedback for optimization of D(W) by evaluating the 'goodness' of load balancing. E may also provide prior knowledge of the load-balancing problem, for instance, by providing time-series models of $L(t)$, or analytical relationships between $L(t)$, D(W), and the feedback signals.

The triple (P,S,E) constitutes a *strategy-learning task* T. Before designing a learning system L, one must examine the relationships between various components of T and L.

IV. Strategy-learning systems

Architecturally, the learning system (L) sits on top of the problem solver (S). We assume that all modifiable parameters of S are represented by the vector W. L uses the feedback F from the environment (E) in order to calculate changes ΔW to the vector W so that future decisions made using D(W) will elicit better feedback from E. Several choices exist for definition and use of the feedback signal F. In any case, current values of F evaluate the decisions taken since the previous evaluation. The problem of converting F into ΔW is called the credit-assignment problem [12], and is often divided into two stages. In the first stage, called temporal credit assignment (TCA), the feedback is divided among the decisions taken recently, according to their eligibility to receive feedback [20]. Next, in structural credit assignment (SCA), the feedback for each decision is divided among various parameters leading to that decision [2]. Parameters involved in making several decisions can and should receive more feedback than others. L has two separate components, a critic C and a learning element LE, which are responsible for solving, respectively, the TCA and SCA problems.

V. The learning task in load balancing

In this section, we examine various aspects of T(P,S,E), the strategy-learning task in load balancing, and study the effect of these aspects on the design of L(C,LE), the learning system. We hope to illustrate the difficulty of T by showing that the strategy-learning task in load balancing poses complex problems of estimation and modeling. In later sections, we describe how SMALL combines various engineering techniques with innovative uses of neural networks in order to address these problems.

A. Objective function

Until now, we have been intentionally vague about the exact specification of O, the objective function. Many choices exist for such a function. Irrespective of the specific form of O, performance objectives at user level are defined in terms of completion times of jobs. Let us use T_j^L and T_j^{NL} to denote, respectively, the completion times of a job j with and without load balancing. (In the latter case, all jobs are executed locally at their sites of origin.) Table I shows several common definitions of the objective function O. The first two functions attempt to optimize absolute completion time, which cannot be estimated from load information alone. The last three functions target optimization of completion time under load balancing relative to the completion time under local execution. Of these three, the last two seek to improve the overall completion time of a

Table I. Objective functions for load balancing

Metric (O)	Comments
T_j^L	Cannot be predicted
$\sum_j \frac{T_j^L}{N}$	
$\frac{T_j^{NL}}{T_j^L}$	Independent of job size; Compared using statistics for given sets of jobs
$\frac{\sum_j T_j^{NL}}{\sum_j T_j^L}$	More complex strategies; Choosing correct job for migration impossible without information on relative sizes
$\frac{\sum_m \sum_j T_j^{NL}}{\sum_m \sum_j T_j^L}$	Delay in propagating decision information

batch of jobs rather than improve the probability of speeding up a random job. These two functions also suffer from another drawback: optimization requires prior information about job size, and requires scheduling of multiple jobs at each decision point. The lauer requirement introduces combinatorial explosion into the scheduling algorithm and delays generation of feedback. Since we are concerned with independent jobs and cannot assume prior knowledge of job sizes, we adopt the third function in the table as our choice of O.

Irrespective of the specific choice of O, its dependence on completion time makes O ill-defined. Without using complete and accurate models of jobs, sites, and background workload, it is impossible to state O as a closed-form function. Instead, O is a measured quantity. Such optimization problems are said to be ill-posed. Ill-posedness precludes prior specification of good strategies and decision metrics. Instead, L must use the measured completion times in order to learn both goal-related metrics $F_M(L_M)$ and goal-achieving policies $D_M(W_M,F_M(L_M),a)$.

B. Time-varying parameters

An instance of load balancing is defined in terms of primitive load measurements $L(t)$, which are dynamic multi-dimensional quantities. The dynamic nature of load makes the delays in measurement of completion time significant. The time spent waiting for resources depends upon the loads of individual resources. As the load on each resource changes with time, so does the overall delay experienced by a job. Problems of inferring $F_M(L_M)$ therefore require (i) using recently observed loading patterns as indicators of future patterns; and, (ii) reducing multi-dimensional time-series information into scalar decision metrics which can then be used for comparing projected completion times of a job at different sites.

C. Violation of Markov property

Markov property asserts that future behavior of a system depends only on its current state. Its manifestation in decision making is the *path-independence axiom*: optimal decision in a state does not depend upon the state-space path leading to that state. In the simplest sense, violation of this property can be illustrated by studying the autocorrelations of the $L(t)$ time series (Appendix). Nonzero partial autocorrelations at lags greater than one are exhibited by the load on all four resources: CPU, disk, memory, and network. This is indicative of the importance of past values in determining future values of load given the current state. Because of the importance of future load values in determination of completion time, past values of load on various resources must be taken into account during decision making.

The path-independence axiom allows one to associate scalar evaluations with states. The violation of Markov property, as well as the nature of completion-time-based objective functions, forces us to associate evaluations with state sequences rather than individual states. As we shall see later, this fact complicates TCA immensely.

Markov property is a characteristic of the internal representation used by S. If S expands its notion of current state to take into account a sufficient number of past load values, one may be able to satisfy the Markov property. However, practical limitations on the size of state representations usually rule out such a possibility.

D. Delayed, evaluative feedback

The amount of information implicit in F, the feedback signal, determines the complexity of T. If a correct outcome a^* is known for all instances of decision making, then L can use the error between a and a^* to compute ΔW. On the other hand, when only an evaluation of decisions is available, L needs to search among several alternative directions and magnitudes for ΔW before choosing one likely to yield favorable evaluation in future.

There is an inherent delay between the moment a scheduling decision d concerning a job j is made, and the times of occurrence of feedback signals evaluating d. Perhaps the most important of these signals is the one produced upon completion of j. Decisions cannot be evaluated independently because the decision to migrate a job affects future loads on both the source and destination sites. While j is active, other decisions made at the site running j are inevitably influenced by the extra load placed by j on that site's resources. A portion of the feedback for those other decisions should therefore be given to decision d.

The nature of F as well as the interdependence between various decisions complicate credit assignment. In a distributed multiprocessor system at equilibrium, jobs finish at least as fast as new jobs arrive. Therefore, feedback signals are produced at roughly the same rate as the overall ariival rate of jobs. Efficient and intelligent procedures are required for both storage of recent decisions and subsequent assignment of credit to each decision in order to fully utilize the information contained in each feedback signal.

Returning to Table I, while objective functions based on batch completion times (such as the last two entries in the table) reduce the frequency of F signals, they increase

the delay between a decision and its feedback. This increased delay increases the number of candidate decisions which, in turn, increases both the storage requirements and the computational complexity of TCA. The increased complexity of TCA is another reason for avoiding batch-completion-time-based objective functions.

In summary, learning a decision metric for load balancing entails regression of measured relative completion time on the time-series data describing the loads on various resources. Such learning is complicated by an ill-defined objective function that can be neither estimated absolutely nor measured independently for different states. Learning of decision policies involves complex TCA problems because of non-markovian representations and delayed evaluative feedback. Therefore, the overall strategy-learning task T facing SMALL is quite complex. In the next section, we discuss previous work on design of strategies for load balancing. Subsequent sections describe SMALL's approach to strategy learning.

VI. Previous work

Table II surveys various approaches to intelligent load balancing along with examples and comments. The few of these that actually use learning of load balancing policies (SLA and the gradient-based approach) do not use measured performance as feedback. Instead, they use analytical simulation models in which states can be evaluated independently and immediately. All these approaches have been demonstrated only in highly stylized simulation environments wherein job-arrival processes are Poisson, background workload is ignored, and the resources of each site can be neatly modeled as a product-form queueing network.

VII. SMALL: Problem decomposition

The overall learning task T described in Section III is too complex to solve all at once. While designing SMALL, one of the key problems we faced was to decompose T into more manageable subproblems. The first line of divide is the natural one: to decompose learning of strategies into learning of goal-related decision metrics and learning of decision policies.

A. Learning goal-related indices

Since jobs are independent, in the ideal case one would like to simultaneously maximize $\frac{T_j^{NL}}{T_j^{L}}$, the speed-up of each job due to load balancing. Typically, a load balancing experiment involves several jobs. A multi-objective optimization problem with ill-defined objective functions would be too complex to solve, and it would be even more complex to learn strategies for solving it. If possible, one would like to combine these multiple objectives into a single objective function to optimize. However, it is not clear how to combine the speed-ups of different jobs into a scalar objective function for evaluating strategies. Prediction of relative merits of alternative actions at any decision point can only be made on the basis of the relative values of such a 'combined' objective function. In order to bootstrap the prediction process, one needs data relating observable parameters ($L(t)$) with the predicted value. Therefore, the learning of goal-related indices for load balancing can be split into three subproblems: choosing a combined objective function (deciding what to predict); collecting data to bootstrap the prediction process; and, using the data to build models of indices related to the combined objective function. The rest of this subsection describes our approach to these three problems.

Deciding what to predict: There are many ways of combining the speed-up of single jobs into a single scalar quantity to be optimized globally. The easiest is to just add them up ($O = \sum_j \frac{T_j^{NL}}{T_j^{L}}$). Such an O would allow large speed-ups on a few jobs to dominate small speed-ups (and even slow-downs) on several others, thus resulting in policies which may (unfairly) favor a few jobs over all others. One solution to this problem of 'unfairness' in combination is to (nonlinearly) transform speed-up values using a monotonically increasing function that asymptotically achieves a maximum value. That will bound the positive contribution to O from each job while simultaneously favoring large speed-ups.

It is possible that load balancing may actually slow down the completion of some jobs relative to local execution. If left uncorrected, this aspect of load balancing policies can result in *starvation*: a few jobs may be slowed down excessively in order to speed up all others. This is to be avoided. Thus, we prefer slower overall speed-up for all jobs over large speed-ups for most jobs and slowdown for some other jobs. The objective function must be suitably adjusted to reflect such starvation-freeness. Our solution to starvation problem is to attach increasingly negative penalty to speed-up values close to zero.

Due to the reasons already described, we would like O to have the form $\sum_j G(\frac{T_j^{NL}}{T_j^{L}})$, where $G(\cdot)$ is a

Table II. Previous research on intelligent load balancing

Classifier Systems	Midgard [18]	No learning
Bayesian Decision Theory	BDT [19]	Hand-coded utility
Learning Automata	Probabilistic [6]	Static policy
Reinforcement Learning	SLA [13]	Analytical performance ; No multiprogramming
Team Theory	Utility-driven control [15]	No learning
Planning	Expert system [9]	No learning
Empirical Learning	Gradient-based approach [16]	Analytical performance

monotonically increasing function, bounded from above by a positive constant, and unbounded below. The form of O enables us to evaluate jobs independently. Thus, at the time of scheduling, one can use the predicted improvement in completion time of the job being scheduled as a parameter of decision making. This is exactly the quality we desire a decision metric to possess. However, because *absolute* completion times are not predictable using the information available at the time of making a decision, and because *relative* completion times are sufficient for choosing among alternatives, we formulate the following *prediction problem*:

To learn functions $F_m(L_m)$ of local load so that $F_m(L_m)-F_n(L_n)$ approximates $\frac{T_j^m(L_m)-T_j^n(L_n)}{T_j^0(L_0)}$, where T_j^i represents the predicted completion time of a job j at machine i.

An intuitive explanation of this formulation is that if a job j is to be sent to a site k having the smallest F_k, then j's predicted completion time on k relative to its completion time at a fixed site 0 should be the least among all the sites considered. Assuming that the true completion time function permits linear ordering of (load, site characteristics) pairs, k will also be the site where the completion time under load balancing will be the least.

The normalization of $T_j^i(L_i)$ with the factor $T_j^0(L_0)$, is an attempt to factor out the job-specific characteristics of completion time. The assumption is that delays due to resource contention grow linearly in the size of the job. That, along with the assumption of linear ordering, ensures that (i) each F_i can be predicted using only the information about the load on site i; and, (ii) F_i values can be numerically compared for different sites across the network in order to choose the site with the least predicted completion time.

Relative to the parameters W, the targets of the learning process, the completion time of a job under no load balancing (T_j^{NL}) is a constant quantity. Therefore, *for fixed values of parameters* W, F_i is a suitable decision metric for characterizing the current and past load on site i. Due to the non-markovian nature of load-related time series, the function $F_i(\cdot)$ must take into account past values of $L_i(t)$.

As we shall see later, the policy-learning problem complements the decision-metric-learning problem. The former modifies the parameters W of the decision process *given fixed functions* F_m. That separate learning of W and $F_i(\cdot)$ is meaningful, is an assumption of the decomposition process.

Data collection: Collecting data for learning the function $F_m(\cdot)$ requires measurement of current and past values of load, and information about completion time of a job introduced under that load. One also needs information about the completion time of that job under no load at a fixed site. Also, comparision of completion times with and without load balancing requires the ability to execute a load-balancing algorithm under controlled reproducible loading conditions, with identical job arrival patterns. Utilities for doing this are part of SMALL's workload generation package, which is described in a separate paper [10].

Model building: Estimation of F_i's is a problem in (nonlinear) regression of T_j^i on time series $L_i(t)$. A typical job in the system can take anywhere from 15 seconds to a few minutes to complete. Load values are typically sampled at a frequency of 10 to 100 Hz. Autocorrelation plots of $L_i(t)$ (Appendix) show a generally decreasing trend; partial autocorrelation plots indicate significant non-Markovian components. One can clearly rule out pure autoregressive (AR) or moving average (MA) models on the basis of these plots [5]. But, the possibility of low-order (less than 5) mixed ARMA models is not ruled out, and is in fact very likely.

Autocorrelations in observed load are due to locality in the behavior of the current process population which changes at approximately the same rates as the arrival rate of new jobs in the system. Therefore, we expect that a few minutes worth of past load values should be considered for prediction of relative completion time. That gives us a window of approximately 1000 samples.

A naive approach to model-building ignores the autocorrelation structure of the time series and simply formulates F_i as a function of past 1000 values of $L_i(t)$. The number of parameters involved in such a naive formulation unnecessarily complicate the learning of decision metrics. However, for time series data without any obvious ARMA tendencies, the naive approach may be the only recourse. Examples of the naive approach include the use of time-delay neural networks in speech recognition [24].

A smarter approach to model-building decomposes the modeling process into two stages: smoothing and fitting. These are described in detail in subsequent sections. Essentially, smoothing isolates the trends in time-series data and produces "filtered" data which can then be sampled at lower frequencies. The lower sampling frequency allows smaller window sizes. The fitting process is then able to model relative completion time as a function of (smaller) windows of current and past (filtered) load values.

B. Learning decision policies

The policy-learning problem complements the learning of decision metrics. Test data for this problem include jobs and load readings. The jobs are organized into streams, one per site, and the recorded loads recreated using the SMALL artificial workload generation package [10]. Policy learning is concerned with generation and assimilation of feedback, corresponding respectively to the processes of evaluation and credit assignment described in Section I. This, then, is the natural way to decompose the policy-learning problem.

Feedback generation: Initially, all test jobs are executed locally and their completion times recorded. Next, the current load-balancing policy is activated and speed-ups over local execution computed for each job as it finishes. A nonlinear function $G(\cdot)$ of speed-up is computed, as described in Section VII-A. The particular function we use (Fig. 1) is:

$$G(s) = 1 - e^{-c\frac{s-1}{s+1}}.$$

The constant c can be chosen by deciding how much penalty to associate with, say, 10 percent slowdown. This function has all the properties mentioned previously, namely, monotonicity, asymptotic bound on the positive side, and exponential unbounded negative values for s smaller than 1 (slowdown).

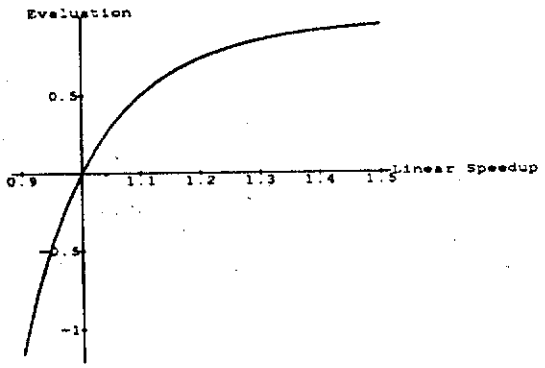

Figure 1. Feedback per job versus speed-up

The site of origin of each job keeps track of completion times of jobs with and without load balancing. A software module at each site computes the G value for each job originating there, and broadcasts the value over the network along with the finishing job's identification number. These broadcasts become the feedback signals F for the strategy-learning task T described in Section III.

Temporal credit assignment: Policy learning aims to modify the parameters W_m of the decision processes D_m in such a fashion that future actions recommended by the D_ms will maximize the sum of feedback values $G(\cdot)$ for future jobs. The first stage in the assimilation of feedback is to perform TCA. When feedback upon completion of a certain job j is received, all decisions affecting j's performance receive their share of j's evaluation. Incremental assignment of credit is possible only because the chosen objective O is a linear function of measured completion-time values.

In order to make TCA efficient, each site maintains — along with a history of recent decisions — information about the *causal and temporal scopes* of each decision. Briefly, causal scope is a binary value indicating whether the decision in question could have influenced the performance of job j whose evaluation is contained in the current feedback signal F. For instance, if j was never run at a site m, and the sites at which j ran did not exchange any tasks with m while j was running, then decisions made at m are not eligible to receive any portion of F. Simple rules such as these constitue a causal model of E, the learning environment. Each decision also has a finite temporal scope, spanning the time axis from the start until the end of execution of the job being scheduled. A decision can only receive feedback from those jobs whose execution overlaps with its temporal scope. In order to minimize the number of past decisions needing storage, decisions whose temporal scopes contain no active jobs can be 'forgotten.'

The critic, whose responsibility it is to perform TCA, computes an *eligibility* value by multiplying a binary causal parameter with a continuous-valued temporal parameter. The temporal parameter is based on a temporal model describing how eligibility of a decision varies with time elapsed since making the decision. It is reasonable in our case to assume maximum eligibility of 1 initially, and decaying (say, linearly) to 0 at the time of completion of the job being scheduled. The feedback signal is divided among the decisions in proportion to their eligibility.

Structural credit assignment: Once a decision receives its share of the feedback signal, various parameters of the decision process that participated in making that decision are given their share of the feedback. This is the SCA problem and its solution depends on the specific representation of D(W). If F were a corrective error signal, and D(W) implemented using feedforward networks with deterministic outcome, one could use an algorithm such as back-propagation [17] or any of its variants for SCA.

F is an evaluative signal. Much work on learning with such signals has been done in the areas of reinforcement learning using stochastic decision rules [3]. Such learning algorithms have been demonstrated only in the binary case where the automaton has a choice between action-inaction. Load balancing is a problem involving multiple alternatives at each decision point. Rules for multiple-action case with binary feedback signals were proposed by Narendra et al. [14]. We, on the other hand, have continuous-valued feedback signals. However, the sign of the feedback signal is significant; this suggests the following approach to learning D(W).

First of all, we interpret $D_m(W_m,F_m(L_m(t)),a)$ as the probability of choosing an action a. Naturally, then, we would like to impose the constraint

$$\sum_{a \in A} D_m(W_m,F_m(L_m(t)),a) = 1.$$

When the feedback signal is positive we increase the probability of the chosen action towards 1, while simultaneously driving the other probabilities towards 0. Similarly, when the feedback is negative, we can drive the probability of the chosen action towards 0 while pushing the other probabilities closer to 1. The modifications should be such that the constraint above is never violated. Thus, we interpret the value of $D_m(\cdot)$ as a probability and the process just described gives us $\Delta D_m(\cdot)$. We have now reduced the problem of learning from evaluative feedback into one of learning from prescriptive feedback. We can now use error-correcting procedures similar to back-propagation to compute ΔW_m.

The process outlined above has one drawback: it loses the information contained in the numerical value of the feedback signal. A learning scheme that can accommodate continuous-valued feedback is the S-model of Thathachar and Narendra [14]. It, however, requires that F be bounded from above and below. Therefore, if we modify our feedback-generation scheme so that some (sufficiently low) negative value is chosen as a lower bound for F, we can use the S-model.

The two schemes described above work only if we employ a stochastic decision policy. If we have a deterministic decision policy, the search for alternative parameter values must be moved to the learning process. In this case, one might use a genetic search in the space of

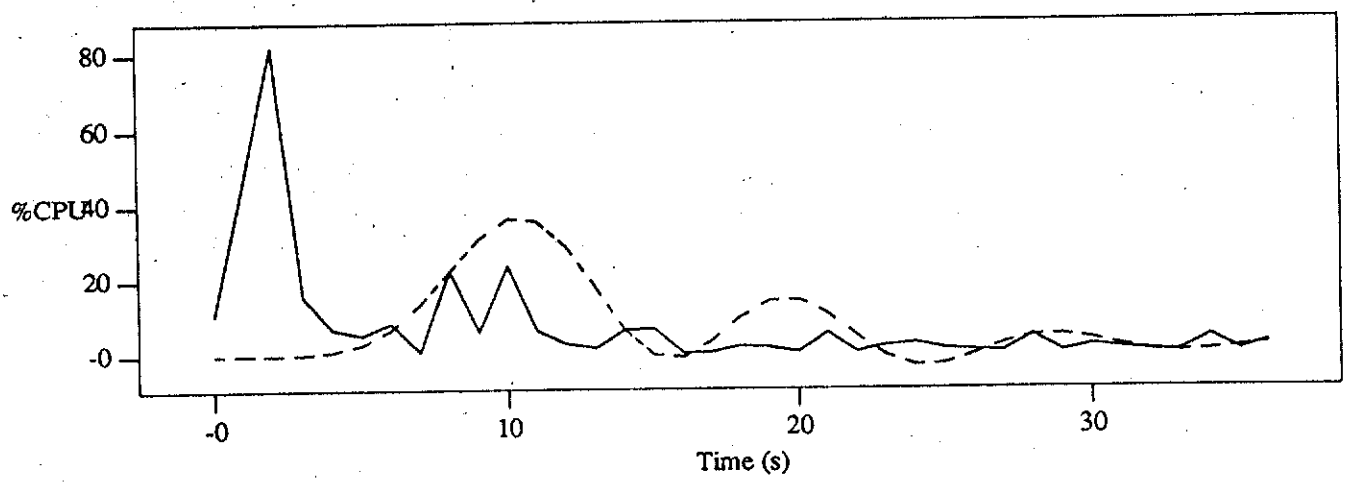

Figure 2. Observed CPU usage smoothed using an IIR filter

parameters, experimenting with randomly chosen parameter values in the neighborhood of the current ones. New values may be accepted if net positive feedback is accumulated over a training set and confirmed over a test set. In addition, one can employ annealing-like procedures to stabilize the learning process towards the end while exploring a large set of parameter values initially.

At this point, we have not implemented any of these schemes. Our choice will be largely governed by the deterministic versus stochastic nature of the load-balancing rules as well as the importance of the numerical information contained in the feedback signal (versus only considering its sign).

VIII. Smoothing

In previous sections, we have noted that the workload parameters exhibit substantial autoc... This was revealed by examining the autocorrela...ots of their time series. Similar information can also be obtained by frequency-domain or spectral analysis of the load data. That is the approach taken in signal processing applications. Techniques are available in these areas for designing *digital filters* which can, among other things, remove undesirable high-frequency jitter, leaving behind a smoothed signal. What is more, the 'degree' of smoothing can be specified in terms of the sampling frequency. Corresponding time-series approaches, called ARMA models, allow identification of the underlying pattern but without the flexible control possible with digital filters.

Before we get into the details of our approach, we would like to illustrate the effects of smoothing with digital filters. Consider Fig. 2 showing the observed CPU usage with solid lines and the filtered signal with dashed lines. In this case, filtering was done with a Butterworth filter [7] of order 9 and cut-off frequency at 15% of sampling frequency. This implies that anything more than 30% of the maximum (unaliased) frequency in the signal got truncated. A Butterworth filter is designed by first specifying certain parameters of the "pass band" and the "stop band."

Notice in Fig. 2 that the filtered output extracts the essential trend of the load data while simultaneously introducing a delay. This delay is inherent to smoothing processes and is proportional to the order of the filter. One might prefer lower order filters having shorter delays but their transfer functions do not cut off sharply and, therefore, some high-frequency components may pass through.

In practice, one does not know how much to smooth a given signal before its trend will become obvious and the removed high-frequency components will not contain information vital to prediction. Ideally, therefore, one would like to have an *adaptive filter* (one that 'discovers' the right cut-off frequency based on the accuracy of prediction). The problem is that whereas Butterworth filters and certain other designs are stable by design (their output remains bounded for bounded input), adaptive filters may not necessarily be stable. In order to guarantee stability, constrained continuous optimization in the space of filter coefficients may be required. One quickly starts getting into areas where available technology is not mature enough to offer off-the-shelf solutions.

A more feasible alternative is to design a 'bank' of stable filters which reliably achieve a wide variety of cut-off frequencies. During prediction, one can pick those that yield better prediction of completion time on the average. SMALL uses neural networks for adaptively weighting these filtered signals.

IX. Fitting

Fig. 3 shows the schematic of the comparator neural network. A window of recent smoothed values from different loading situations are made available at the inputs of a pair of feedforward sigmoidal neural networks. The pairs exhibit perfect symmetry of interconnection, and could actually be implemented by only one network. The figure shows that the 'teaching' input is the difference in completion times normalized for comparabilty by division with the completion time on an unloaded reference site. If the

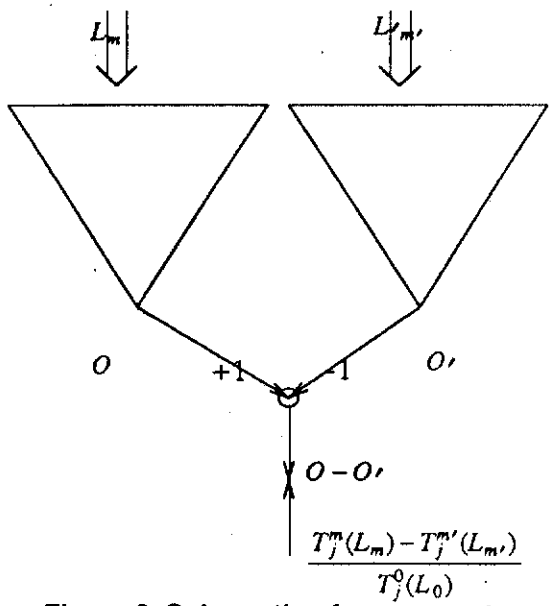

Figure 3. Schematic of a comparator network

network is trained by an LMS algorithm like back-propagation, it will learn to bring the difference in the outputs of the network-pair closer to the measured differences in completion times.

It is, however, possible that the paired networks develop disparate models for the function F_m. This situation should be avoided. One way out is to 'link' the corresponding weights in the two nets so that the actual change in each parameter is the average of the recommended changes for the left and right parts. Another way is to make the training set complete under anti-commutativity: whenever the network produces a certain difference for a pair of inputs, it should be required to produce the negative of that difference for the same inputs with sides switched. In preliminary experiments with toy functions, we have found that networks trained under anti-commutativity tend to develop symmetric weights. The first approach, using linked weights, has been demonstrated by Tesauro [23].

X. Characteristics of our approach

We propose a distributed approach to learning and problem solving in load balancing. The key reason for the feasibility of a distributed learning algorithm is the (designed) linearity of objective function in the distributed parameters, while allowing complex nonlinearities for combination of locally available parameters. Otherwise, a distributed solution may incur too much communication overhead during learning.

We make explicit the knowledge that goes into designing TCA procedures: especially, the use of causal and temporal scopes both in computing eligibilities of decisions and in limiting the storage needed for past decisions. Our approach may not be suitable for problems in which decisions have large or infinite temporal scopes. However, load-balancing decisions have finite well-defined causal and temporal scopes. There may be yet other problems for which scopes may need to be learned empirically. An environment like SMALL supports controlled experimentation and data gathering; it may be used for gathering empirical information about the causal and temporal relationships in load balancing if prior models are found to be inadequate.

XI. Final remarks

Traditional approaches to load-balancing address either elegant but unrealistic optimization issues or down-to-earth programming and implementation issues. We have attempted to bridge this gap by formulating load balancing as an abstract strategy-learning task while emphasizing measurability and efficient implementation.

References

[1] R. Alonso, "The Design of Load Balancing Strategies for Distributed Systems," *Future Directions in Computer Architecture and Software Workshop*, pp. 1-6, Seabrook Island, SC, May 5-7, 1986.

[2] C. W. Anderson, "Strategy Learning with Multilayer Connectionist Representations," *Proc. Fourth Int'l. Workshop on Machine Learning*, pp. 103-114, Morgan Kaufmann, June 1987.

[3] A. G. Barto and P. Anandan, "Pattern-Recognizing Stochastic Learning Automata," *Trans. Systems, Man, and Cybernetics*, vol. SMC-15, no. 3, pp. 360-375, IEEE, 1985.

[4] K. Baumgartner and B. W. Wah, "GAMMON: A Load Balancing Strategy for a Local Computer System with a Multiaccess Network," *Trans. on Computers*, vol. 38, no. 8, pp. 1098-1109, IEEE, Aug. 1989.

[5] G. E. P. Box and G. M. Jenkins, *Time Series Analysis: Forecasting and Control, 2nd ed.*, Holden-Day, San Francisco, 1976.

[6] R. M. Glorioso and F. Colon-Osorio, "Engineering Intelligent Systems," in *Engineering Intelligent Systems*, Digital Press, Burlington, MA, 1980.

[7] R. W. Hamming, *Digital Filters*, Prentice-Hall, Englewood Cliffs, NJ, 1989.

[8] K. Hwang, W. J. Croft, G. H. Goble, B. W. Wah, F. A. Briggs, W. R. Simmons, and C. L. Coates, "A UNIX-based Local Computer Network with Load Balancing," *Computer*, vol. 15, no. 4, pp. 55-66, IEEE, April 1982.

[9] Ajoy Kumar, Mukesh Singhal, and Ming T. Liu, "A Model for Distributed Decision Making: An Expert System for Load Balancing in Distributed Systems," *Proc. COMPSAC*, pp. 507-513, IEEE, 1987.

[10] P. Mehra and B. W. Wah, "Physical-Level Synthetic Workload Generation for Load-Balancing Experiments," *Int'l. Conf. on Parallel Processing*, 1992 (submitted for review).

[11] T. K. Miller III, R. S. Sutton, and P. J. Werbos (eds.), *Neural Networks for Control*, MIT Press, Cambridge, MA, 1990.

[12] M. Minsky, "Steps Toward Artificial Intelligence," pp. 406-450 in *Computers and Thought*, ed. E. A.

Feigenbaum and J. Feldman, McGraw-Hill, New York, 1963.

[13] R. Mirchandaney and J. A. Stankovic, "Using Stochastic Learning Automata for Job Scheduling in Distributed Processing Systems," *Jrnl. Parallel Disbtd. Computing*, pp. 527-552, Academic Press, 1986.

[14] K. Narendra and M. A. L. Thathachar, *Learning Automata: An Introduction*, Prentice Hall, Englewood Cliffs, NJ, 1989.

[15] J. C. Pasquale, Intelligent Decentralized Control in Large Distributed Computer Systems(Ph.D. Thesis), Univ. of California, Berkeley, CA, April 1988.

[16] S. Pulidas, D. Towsley, and J. A. Stankovic, "Imbedding Gradient Estimators in Load Balancing Algorithms," *Proc. 8th Int'l. Conf. Distributed Computer Systems*, pp. 482-490, IEEE, 1988.

[17] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning Internal Representations By Error Propagation," pp. 318-362 in *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, ed. D. E. Rumelhart, J. L McClelland and the PDP Research Group, MIT Press, Cambridge, MA, 1986.

[18] A.V. Sannier II and E.D. Goodman, "Midgard: A Genetic Approach to Adaptive Load Balancing for Distributed Systems," *Machine Learning*, pp. 174-180, Kluwer Academic Publishers, Boston, MA, 1988.

[19] J. A. Stankovic, "An Application of Bayesian Decision Theory to Decentralized Control of Job Scheduling," *Trans. on Computers*, vol. C-34, pp. 117-130, IEEE, Feb. 1985.

[20] R. S. Sutton, "Learning to Predict by the Methods of Temporal Differences," *Machine Learning*, vol. 3, pp. 9-44, Kluwer Academic Publishers, August 1988.

[21] R. S. Sutton, "Integrated Architectures for Learning, Planning, and Reacting Based on Approximating Dynamic Programming," *Proc. 7th Int'l. Conf. Machine Learning*, pp. 216-224, Morgan Kaufmann, Palo Alto, CA, 1990.

[22] G. Tesauro, "Connectionist Learning of Expert Backgammon Evaluations," *Machine Learning*, pp. 200-206, Kluwer Academic Publishers, Boston, MA, 1988.

[23] G. Tesauro and T. J. Sejnowski, "A Parallel Network that Learns to Play Backgammon," *Artificial Intelligence*, vol. 39, pp. 357-390, Elsevier Science Pub., New York, 1989.

[24] A. Waibel, T. Hanazawa, G. Hinton, K. Shikano, and K. Lang, "Phoneme Recognition Using Time-Delay Neural Networks," *Trans. on Acoustics Speech and Signal Processing*, vol. 37, no. 3, pp. 328-339, IEEE, March 1989.

[25] S. Zhou, Performance Studies of Dynamic Load Balancing in Distributed Systems (Technical Report No. UCB/CSD 87/376, Ph.D. dissertation), Computer Science Division, Univ. California, Berkeley, CA, 1987.

Appendix: Autocorrelation plots for load metrics

('.' marks two standard errors)

NAME OF VARIABLE = CPU

AUTOCORRELATIONS

LAG	CORRELATION	-1 9 8 7 6 5 4 3 2 1 0 1 2 3 4 5 6 7 8 9 1	STD
0	1.00000	|**********|	0
1	0.77239	.|**********	0.02
2	0.65880	.|**********	0.0296188
3	0.59750	.|**********	0.0349927
4	0.55318	.|**********	0.0388599
5	0.51085	.|**********	0.0418915
6	0.48336	.|**********	0.0443133
7	0.45939	.|**********	0.0463743
8	0.45980	.|**********	0.0481602
9	0.44758	.|**********	0.0498853
10	0.41551	.|**********	0.0514665

PARTIAL AUTOCORRELATIONS

LAG	CORRELATION	-1 9 8 7 6 5 4 3 2 1 0 1 2 3 4 5 6 7 8 9 1
1	0.77239	.|**********
2	0.15422	.|***
3	0.12188	.|**
4	0.07938	.|**
5	0.04088	.|*
6	0.05552	.|*
7	0.03916	.|*
8	0.08979	.|**
9	0.03318	.|*
10	-0.01774	.|.

NAME OF VARIABLE = DSK

AUTOCORRELATIONS

LAG	CORRELATION	-1 9 8 7 6 5 4 3 2 1 0 1 2 3 4 5 6 7 8 9 1	STD
0	1.00000	|**********|	0
1	0.72949	.|**********	0.02
2	0.60479	.|**********	0.0287354
3	0.50963	.|**********	0.0334416
4	0.46926	.|**********	0.036416
5	0.44313	.|**********	0.0387594
6	0.41982	.|**********	0.0407355
7	0.42274	.|**********	0.0424309
8	0.41649	.|**********	0.0440834
9	0.40167	.|**********	0.0456303
10	0.37333	.|**********	0.0470233

PARTIAL AUTOCORRELATIONS

LAG	CORRELATION	-1 9 8 7 6 5 4 3 2 1 0 1 2 3 4 5 6 7 8 9 1
1	0.72949	.|**********
2	0.15526	.|***
3	0.05187	.|*
4	0.10302	.|**
5	0.07825	.|**
6	0.05139	.|*
7	0.09721	.|**
8	0.05656	.|*
9	0.02924	.|*
10	0.00282	.|.