AUTOMATED LEARNING OF WORKLOAD MEASURES FOR LOAD BALANCING ON A DISTRIBUTED SYSTEM

Pankaj Mehra and Benjamin W. Wah[†]
Coordinated Science Laboratory
University of Illinois, Urbana-Champaign
1308 West Main Street
Urbana, IL 61801
{wah,mehra}@manip.crhc.uiuc.edu

ABSTRACT

Load-balancing systems use workload indices to dynamically schedule jobs. We present a novel method of automatically learning such indices. Our approach uses comparator neural networks, one per site, which learn to predict the relative speedup of an incoming job using only the resource-utilization patterns observed prior to the job's arrival. Our load indices combine information from the key resources of contention: CPU, disk, network, and memory. Our learning algorithm overcomes the lack of job-specific information by learning to compare the relative speedups of different sites with respect to the same job, rather than attempting to predict absolute speedups. We present conditions under which such learning is viable. Our results show that indices learnt using comparator networks correctly pick the best destination in most cases when incoming jobs are short; accuracy degrades as execution time increases.

1. INTRODUCTION

This papers addresses the computation of workload measures in distributed systems. Such measures are used by dynamic load-balancing policies to determine the site most suitable for executing an incoming task.

Our model of distributed systems comprises a network of interconnected sites. Each site has private resources such as CPU and memory; some have disks; and all share the network resource. These resources are *architecturally homogeneous*: they can service requests from programs at any of the sites. Resources such as network and secondary storage can be shared transparently among the processes at different sites; other resources, such as CPU and memory, can only be accessed by local processes. We do allow for *configurational heterogeneity*, i.e., sites having different processor speeds, memory space, or disk space.

We assume that the "background workload" at each site varies outside the control of the load-balancing software. Such load may be caused either by operating-system functions or by tasks that, for some reason, cannot be migrated. We assume no prior knowledge of the tasks to be scheduled, except that they are drawn from a large population having stationary (but unknown) mean and variance of relative resource requirements. Local execution of an incoming task is the base case, relative to which all load-balancing decisions are evaluated. Since we assume independent tasks, our objective is to maximize the average speed-up, computed on a task-by-task basis, over local execution.

The unpredictable long-term variation of background workload rules out static strategies, and, coupled with our ignorance of task lengths, necessitates preemptive strategies which can "undo" the effects of poor initial placements. The distributed arrival pattern of tasks and the lack of constraining dependences between them entail that we prefer the less expensive and more robust decentralized strategies over the centralized ones. Therefore, our focus in this paper is on dynamic, decentralized, and preemptive load-balancing strategies.

Load-balancing strategies have two components: *workload measures*, which indicate each site's load; and *decision policies*, which determine both the conditions under which tasks are migrated and the destinations of incoming tasks. In this paper, we address automated learning of load measures; learning of policy parameters is described elsewhere [8].

Figure 1 shows the parameterized policy considered in this paper. The sender-side rules (SSRs) are evaluated at s, the site of arrival of a task. *Reference* can be either *0* or *MinLoad*; the other parameters — δ, θ_1, and θ_2 — take non-negative real values. A remote destination, r, is picked randomly from *Destinations*, a set of sites whose load indices fall within a small neighborhood of *Reference*. If *Destinations* is the empty set, or if the last SSR fails, then the task is executed locally at s; otherwise, site r is requested to receive the task. Upon receiving that

```
SENDER-SIDE RULES(s)
    Destinations = {site: Load(site)-Reference(s) < δ(s)}
    Destination = Random(Possible_destinations)
    IF Load(s) - Reference(s) > θ1(s) THEN Send
RECEIVER-SIDE RULES(r)
    IF Load(r) < θ2(r) THEN Receive
```

Figure 1. The load-balancing policy studied

[†]Contact author: B. W. Wah. Research supported by National Aeronautics and Space Administration Grant NCC 2-481 and a grant from Sumitomo Electric Industries, Yokohama, Japan.

request, site r applies its receiver-side rule (RSR). If the RSR succeeds, the task is migrated; otherwise, the task is executed locally at s.

Our policy is a generalization of several well-known decentralized dynamic load-balancing policies [1,7,12]. It uses primitive workload measures (*Load*, denoting the load at each site) as well as an abstract measure (*MinLoad*, denoting the smallest *Load* value). Our focus in this paper is on methods for calculating *Load*.

A system supporting our load-balancing policy must support (i) measurement of low-level resource-utilization information; (ii) communication of *Load* values among sites; and (iii) determination of *MinLoad*.

The rest of this paper is organized as follows. In Section 2, we critique existing methods for computing load indices and motivate automated learning techniques. In Section 3, we describe how DWG, a workload generator, is useful for collecting the data necessary for learning. In Section 4, we describe our algorithm for learning load indices using neural networks. Section 5 contains empirical results on the quality of load indices learnt using our method. Section 6 concludes this paper.

2. LOAD INDICES

Load-balancing systems seek to maximize speed-up over local execution. This requires them to rank alternative destinations for each incoming task by their *expected* speed-ups over local execution. Usually, alternative destinations are ranked by their *Load* values, which are computed using a manually-specified formula as functions of current and recent utilization levels of various resources. Good load indices are difficult to design as they are sensitive to installation-specific characteristics of hardware devices as well as to the prevalent load patterns. This section reviews current approaches to load-index design, discusses their deficiencies, and presents our approach based on automated learning.

2.1. *Existing Methods*

Many existing methods for computing load indices use simplified queuing models of computer systems [3,4,12]. Almost all implemented systems use a function known as *UNIX-style load average* (hereafter, load average), which is an exponentially smoothed average of the total number of processes competing for CPU. Load average meaningfully compares loading situations across configurationally identical sites, but fails when the distributed system is heterogeneous. Assuming preemptive round-robin scheduling of processes within each site, let us consider the comparison between a site having a load average of 3 and another that is 5 times slower and has a load average of 0. An incoming task is likely to require 20% lesser time to complete at the fast site than the at the slow site, despite the former's high load average! A more fundamental problem with the traditional load-average function is that it completely ignores resources other than the CPU. Therefore, while it may be reasonable predict the performance of purely compute-bound tasks, its utility is questionable for tasks that use the other resources of contention: memory, disk, and network.

Typical workstation operating systems provide a number of different performance metrics. (For example, over 20 performance metrics are available in the SunOS kernel.) Although measuring the utilization levels of resources other than CPU may not require any hardware modifications, several of these metrics are unsuitable for inclusion in a load index because the overhead of estimating their values precludes frequent sampling. These include process-level metrics, which are sampled only once every 5 seconds. Even if we eliminate these, we are still left with a fairly large set of mutually dependent variables; for example, disk traffic is affected by the number of page swaps and process swaps. Other metrics, such as rate of data transfer, are fixed quantities for a given configuration, and affect only the (fixed) coefficients of a load index.

Ideally, workloads for load balancing should be characterized by a small set of performance metrics satisfying the following criteria: (i) low overhead of measurement, which implies that measurements can be performed frequently, yielding up-to-date information about load; (ii) ability to represent the loads on all the resources of contention; and (iii) measurable and controllable independently of each other. In the past, Zhou [12] has considered resource-queue lengths (the number of processes waiting for CPU, disk, and memory) in designing load indices. Although his approach of using multiple measurable metrics is similar to ours, he did not consider a systematic method for learning good workload indices. Further, he did not consider the design of a suitable workload generator for verifying his load index. In this paper we use the instantaneous utilization levels on the four basic resources — CPU, memory, disk, and network — that form a useful set of performance metrics satisfying all three criteria. We further present a synthetic workload generator that can be used for collecting workload patterns and for regenerating the workload collected in the presence of foreground jobs.

2.2. *Automated Learning of Load Indices*

Ideally, we would like to rank alternative destinations for an incoming job by their respective completion times. However, completion times can only be measured for completed jobs, whereas decisions needs to be made before jobs start. Therefore, we need to somehow *predict* completion times using only the information available before a job begins execution. Without knowing the resource requirements of jobs, we cannot predict absolute task-completion times.

Notice, however, that we only need to *compare* alternative destinations for the *same job*. Therefore, we only need to determine a relative (site-specific and configuration-specific) measure of completion time. It would, therefore, suffice to predict the *relative* completion times of a job at different sites. As a point of reference, we can use the completion time of the job on an idle file server [6]. Every site needs to predict the completion time of an incoming job relative to its completion time on the chosen idle file server, given only the loading conditions at job-arrival time. (Such prediction is only feasible for autocorrelated resource-utilization patterns [9].)

Predicted relative completion times can be used as load indices. If completion times of different sites for the same job vary widely, then the accuracy of prediction can be low; otherwise, if the difference between completion times are small relative to the reference, then more accurate prediction is needed. At the outset, it is unclear how much accuracy will suffice. Since the accuracy of convergent iterative learning procedures improves with training, we need to reformulate our problem so that we will know when adequate accuracy is attained.

Instead of predicting, for each site, the relative completion time of a task at that site, we now predict, *for each pair of sites*, the difference between their relative completion times. We express accuracy as the *percentage of correct comparisons*. We can stop refining our predictions when sufficient accuracy is achieved. We also need to ensure that the anti-symmetry of comparison — whenever A is better than B, B is worse than A — is preserved during learning.

3. DATA COLLECTION

No matter how many load indices we need to compare, the number of possible destinations for an incoming job is finite and equals the number of sites. Therefore, we can collect all the data necessary for learning load indices *off-line* and ahead of time. Such data take the form of before-after pairs: the first item of each pair is a trace of utilization of different resources; the second is the measured completion time of a job introduced at the end of the trace period. We can first measure realistic system-wide workloads, and then replay them repeatedly, each time introducing a job at precisely the same time into the experiment but at a different site of the network. If we repeat such experiments with several representative jobs, under several realistic loading conditions, then we can create a large database of realistic decision points.

Our experimentation environment must allow us to both *generate realistic loading conditions* and *repeat them as often as desired*. Representative test jobs must be run under such *synthetic workloads*, and the utilization of all resources recorded periodically. Both utilization levels and completion times must be measured with low overhead and high precision. With these in mind, we developed DWG [8, 10], a synthetic workload generator that helps us build our database of decision points.

DWG can measure and control the utilization levels of four key resources: CPU, memory, disk, and network. It supports a variety of data-collection operations: (i) precise measurement of resource-utilization patterns; (ii) precise generation of recorded patterns; (iii) initiation of foreground test jobs at precise times; and (iv) measurement of job-completion times. To accurately reproduce the behavior of the process population generating the measured load, we have implemented most of DWG inside the kernel [8] so that it has complete control over the utilization levels of local resources. DWG gives up a fraction of its resources allocated to the background load in response to the arrival of a foreground job, and reclaims these resources when the job terminates. The precise amount to give up is controlled by "doctoring" rules. Parameterization and tuning of these rules is described elsewhere [8].

We collected load patterns on a configurationally heterogeneous system consisting of (i) a diskless Sun 3/50 with 4 Mbytes of RAM; (ii) a diskful Sun 3/50 with 4 Mbytes; (iii) a diskful Sun 3/260 with 8 Mbytes; and (iv) a diskless Sun 3/60 with 24 Mbytes. The four workstations were connected by a single 10 Mb/s Ethernet.

We used a total of ten test jobs: three for sorting records of various sizes with various amounts of initial memory, two for uncompressing files of different sizes, and five from the Perfect Club suite of benchmarks (FLO52Q, TRFD, QCD, TRACK, and DYFESM) [2].

To study preemptive scheduling of jobs, we inserted checkpoints into each test job. Each checkpoint resembles a preemption point: the given job can be preempted and resumed at any of its checkpoints; consequently, we can treat the segments from the beginning of an instrumented job to each of its checkpoints as independent jobs. Although this technique creates a large database of training patterns, they could be less effective since different segments of the same test job could have similar resource requirements. This, however, does not affect the accuracy of our final result, as we are not interested in just the overall average speedup, but rather average speedups as a function of job length. (Speedups for different checkpoints of the same job are, therefore, never compared in the final result.)

Each job was instrumented to produce approximately 200 checkpoints during its execution. The Perfect Club benchmarks were modified so that they could complete within five minutes on an idle Sun 3/50 workstation. In order to avoid checkpoints too close to each other, our final database included the first checkpoint of each foreground job and, thereafter, the next checkpoint that took at least 5% longer than the one included before it. The final checkpoint of each job was always included. We ended up with a total of only 58 jobs, with about five checkpoints per test job; each of these jobs was run at each of the 4 sites and under 24 different background load patterns, for a total of 5,324 decision points.

Of the 24 background load patterns used in our experiments, 20 were created by running job-files created randomly from the pool of 10 test jobs described above. The remaining 4 load patterns were designed to create surprises for the load-balancing system. We started with an actual, heavy, system-wide load and patched together pieces of it and the idle load pattern, such that the resulting patterns would frequently contain loading conditions just the opposite of those prior to the job's arrival. Since the load-balancing system can only access the loading conditions before a job's arrival, its decisions would falter under these 'surprising' loads. Moreover, since real workloads often have unpredictable changes in workload, whose likelihood increases with the length of the load pattern, we packed more surprises near the end of the load pattern. We expect that, because of these load patterns, the performance (speed-up over local execution) attained by scheduling a job at the site with the least load index will tend to drop with the length of the job, approaching the performance of random choice for long jobs.

4. COMPARATOR NEURAL NETWORKS FOR LEARNING LOAD INDICES

We now describe our approach to off-line learning of load index functions. First, we formally state our learning task: learning to compare functions of multivariate time series. Then, we present the comparator-network architecture and its associated learning algorithms.

4.1. Design Goals

Different destinations (loading conditions) yield different completion times for an incoming job; our objective is to find the one with the minimum completion time. Comparison of job-completion times requires information about the utilization levels of local as well as remote resources. Raw utilization levels of resources are highly dynamic vector quantities, whose communication would incur too much overhead. It is desirable that comparison among sites be achieved solely via periodic and infrequent communication of scalar load indices, whose computation does not involve utilization levels of remote resources.

Completion times of jobs are unknown at their time of arrival. Given our lack of information about resource-intensities of different jobs, and unavailability of models relating resource-utilization patterns to job-completion times, we can predict completion times of jobs using functions statistically estimated from data. The data for estimation can be easily obtained using DWG; they contain information about (i) the utilization patterns of different resources prior to the arrival of a test job; and (ii) completion times of that job at each of the alternative destinations.

Our goal in load-index learning, therefore, is to estimate functions F_s, one per site, such that (i) the functions do not assume any prior information about the job being scheduled; (ii) they depend only on (smoothed, or otherwise processed) local resource-utilization values; and (iii) the ranking induced on alternative destinations by these functions is consistent with the ranking induced on those same destinations by the true completion times. Before the objective of learning is stated, we define the following symbols.

S: Set of S sites in the network.

F: Set of F foreground test jobs. 0 is the case with no foreground job.

T: Window size in time units during which workload is to be generated for the distributed computer system. (Note that the window of time covers all computers in the system, and that our unit of time equals 20 milliseconds, which is the period between successive interrupts of a real-time clock.)

B: Set containing B background load patterns, where a background load pattern is defined as a collection of background jobs, each characterized by a prescribed site and time of arrival. 0 is the case with no background load.

$l_{b,f(t),s}$: A vector containing the utilization levels at site s for CPU, memory, disk, and network at each time unit in the window for background load pattern b and foreground job f started at time t. $l_{b,f(t),s}$ is a matrix of 4 rows (representing the four resources) and T columns. Note that $l_{b,0,s}$ represents the measured utilization when no foreground job is run, and that, whenever it is obvious, we will use f instead of $f(t)$.

$\hat{l}_{b,f(t),s}$: Vector of values derived from recent behavior of loads on different locally accessible resources at site s for background load pattern b before foreground job f is started at time t. $\hat{l}_{b,f(t),s}$, therefore, depends on t, the time that job f is started, but not on the behavior of job f. In subsequent discussions, we use f instead of $f(t)$ when t is constant.

$F_s^W\left[\hat{l}_{b,f,s}, f\right]$: Value of site s's load-index function, where W denotes the current values of the *weights* (or parameters) of F_s. We will omit the superscript W from our equations whenever it is obvious.

The objective of load-index learning, then, is

$$F_s\left[\hat{l}_{b,f,s_1}, f\right] - F_s\left[\hat{l}_{b,f,s_2}, f\right] = \frac{C\left[\hat{l}_{b,f,s_1}, f\right] - C\left[\hat{l}_{b,f,s_2}, f\right]}{C\left[\hat{l}_{0,f,s_{ref}}, f\right]} \qquad (1)$$

for all foreground jobs f, for all background load patterns $\hat{l}_{b,f,s}$, and for all pairs (s_1, s_2) of sites. Note that $C\left[\hat{l}_{0,f,s_{ref}}, f\right]$ represents the completion time of job f at the idle reference site. Further, note that in Eq. (1), we assume that load indices are computed at time t for both sites s_1 and s_2. In practice, the above assumption is not correct, as it is difficult to synchronize the computation of indices at different sites. Moreover, there are delays in obtaining load indices from remote sites. These errors are not considered in the design of load-index function but rather in other parameters of the load balancing policy [8].

Let us examine whether the right-hand side of Eq. (1) depends on the job f being scheduled. Consider the expression $C\left[\hat{l}_{b,f,s}, f\right] / C\left[\hat{l}_{0,f,s}, f\right]$, which is commonly known as *stretch factor* [3]; its denominator is known as the *service time* of job f at site s. The latter depends only upon f and s, but not on the b. Further, the completion time $C\left[\hat{l}_{b,f,s}, f\right]$ can be expressed as a sum of service time and waiting time:

$$C\left[\hat{l}_{b,f,s}, f\right] = C\left[\hat{l}_{0,f,s}, f\right] + W\left[\hat{l}_{b,f,s}, f\right], \qquad (2)$$

where $W\left[\hat{l}_{b,f,s}, f\right]$ is the time spent by job f waiting for resources. Under round-robin scheduling policies at the process level, waiting time grows linearly with service time. The rate of growth depends upon the load: when the load is high, even short jobs can take a long time to complete. Therefore, we can rewrite Eq. (2) as:

$$C\left[\hat{l}_{b,f,s}, f\right] = C\left[\hat{l}_{0,f,s}, f\right]\left[1 + G\left[\hat{l}_{b,f,s}\right]\right], \qquad (3)$$

where $G\left[\hat{l}_{b,f,s}\right]$ is a site-specific function that increases monotonically with load.

Under the model of completion times described above, the stretch factor $C\left[\hat{l}_{b,f,s}, f\right] / C\left[\hat{l}_{0,f,s}, f\right]$ depends only upon the load $\hat{l}_{b,f,s}$ and site s, but not on the job f. If we further assume that the different sites of our distributed system come from the same architecture family and differ only in their raw speeds, then

$$C\left[\hat{l}_{0,f,s}, f\right] = K(s)\, C\left[\hat{l}_{0,f,s_{ref}}, f\right], \tag{4}$$

where $K(s)$ is some site-specific constant. Now, we can rewrite Eq. (1) as

$$F_s\left[\hat{l}_{b,f,s_1}, f\right] - F_s\left[\hat{l}_{b,f,s_2}, f\right] \tag{5}$$
$$= K(s_1)\left[1 + G\left[\hat{l}_{b,f,s_1}\right]\right] - K(s_2)\left[1 + G\left[\hat{l}_{b,f,s_2}\right]\right],$$

The right hand side does not depend upon the job f being scheduled. Therefore, under the assumptions described above, the objectives described by Eq. (1) are achievable without prior knowledge of the job being scheduled.

In short, the two assumptions required for the just-proven independence between the characteristics of a job f and the right-hand side of Eq. (1) are: (i) waiting time grows linearly with service time; and (ii) different sites come from the same architecture family. The first of these assumptions is a reasonable requirement of any fair scheduling policy, and is approximately true of UNIX-related operating systems that employ variants of round-robin scheduling. The second assumption is generally true of workstation-based computing environments, because even workstations from different vendors often employ the same microprocessor architecture.

How well can the objective described by Eq. (1) be satisfied depends upon several factors. Let $\mathbf{W}$ be the space of all parameter values for functions F_s, and let $\mathbf{F} = \{F^W: W \in \mathbf{W}\}$. The load-index function we shall learn must belong to the family of functions $\mathbf{F}$. Even when the best function from this family is used, the error between the left and right sides of Eq. (1) may not be zero for all data points; such errors contribute to the *approximation error*, which is defined independently of the training algorithm used for adjusting W. A given training algorithm may not be capable of finding the optimal weights W in reasonable amount of time. The error incurred due to suboptimality of weights W is called *estimation error*; it depends upon the particular training algorithm used. We need an approach that reduces both the approximation error as well as the estimation error.

To achieve our goal of obtaining *accurate comparison in most situations*, we would like both sides of Eq. (1) to have the same sign. We, therefore, seek to learn load-index functions that, when compared across sites, correctly predict (for most training patterns) the site with the lower relative completion time.

We partitioned our database of raw training patterns into a *raw training set* and a *raw test set*. We used the raw training set to tune the weights of the load-index functions, and the raw test set to evaluate the trained functions. Suppose that the set of raw training patterns is $\mathbf{R} = \{\tau(b, f, s_1, s_2) \mid b \in \mathbf{B}; f \in \mathbf{F}; s_1, s_2 \in \mathbf{S}; s_1 \neq s_2\}$. Each raw training pattern $\tau(b, f, s_1, s_2)$ is a 5-tuple $\left\{\hat{l}_{b,f,s_1}, C\left[\hat{l}_{b,f,s_1}, f\right], \hat{l}_{b,f,s_2}, C\left[\hat{l}_{b,f,s_2}, f\right], C\left[\hat{l}_{0,f,s_{ref}}, f\right]\right\}$.

The objective function for load-index learning can be formally defined as follows.

$$\text{Minimize } \sum_{\tau \in \mathbf{R}} E(\tau), \text{ where} \tag{6}$$
$$E(\tau(b, f, s_1, s_2)) = \begin{cases} 0 & \text{if } \left[F_s\left[\hat{l}_{b,f,s_1}, f\right] - F_s\left[\hat{l}_{b,f,s_2}, f\right]\right] \\ & \times \left[C\left[\hat{l}_{b,f,s_1}, f\right] - C\left[\hat{l}_{b,f,s_2}, f\right]\right] > 0 \\ 1 & \text{otherwise} \end{cases}$$

4.2. *The Comparator Network: Architecture and Learning Algorithms*

Little is known about the problem of learning to compare functions. One exception is the work of Tesauro [11], who invented the comparator-neural-network architecture for learning to compare alternative moves for the game of backgammon. His approach does not directly carry over to the problem of comparing functions of time series. Our approach, described in this section, was motivated by Tesauro's work; however, in adopting his work to the index-learning problem, we have made significant departures from both his network configurations and training algorithms.

Figure 2 shows a schematic of our comparator neural network. It shows the details of the training algorithm, and the flow of information during a typical learning trial. Each learning trial involves one training pattern from the training set: first, resource-utilization information from a pair of randomly selected training patterns is presented at the inputs (to the left); then, the *actual outputs* of the two index functions are com-

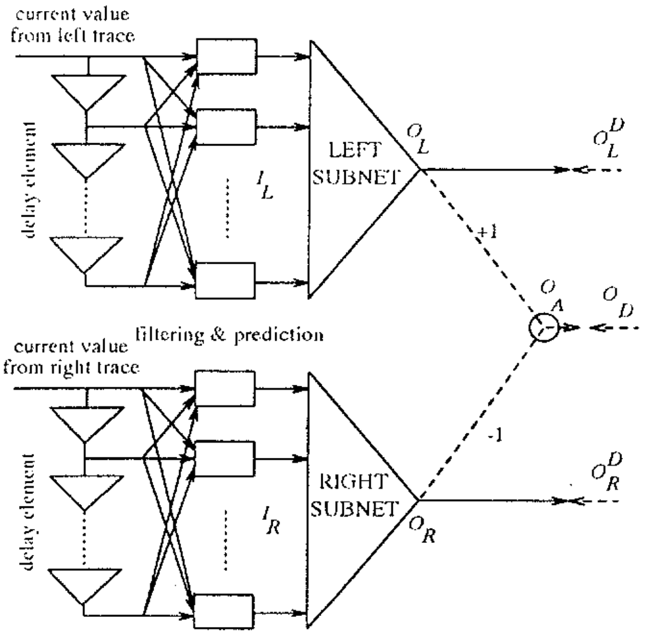

Figure 2. An episode in training a comparator network

puted; based on these outputs, the *desired outputs* for each of the index functions is computed as follows.

$$O_L^D = O_L - 2\eta(O_L - O_R - O^D) \tag{7a}$$

$$O_R^D = O_R - 2\eta(O_L - O_R - O^D) \tag{7b}$$

Finally, the two index functions are modified so that future presentations of similar inputs will generate outputs closer to their respective desired outputs.

Raw utilization patterns enter from the left of Figure 2; the delay elements create a window of recent values; the traces from each window are smoothed using low-pass filtering and an estimate of future resource utilization determined by extrapolating the smoothed trace. We use 5 different filters, with cutoff frequencies at, respectively, 1%, 5%, 10%, 25%, and 50% of maximum frequency. This yields 5 filtered traces per resource. Each of the filtered traces is projected a fixed interval into the future; the interval of projection equals the average completion time of test jobs at the reference site. Two different extrapolation techniques are used: one using linear fitting and the other using exponential fitting. The area under each extrapolated curve (see Figure 3) is used as an input to the load-index function. Since there are 4 different resources, 5 filters, and 2 extrapolation methods, our indices are functions of 40 variables. Vectors of these variables constitute the inputs I_L and I_R of the load-index functions (Figure 2).

We implemented load-index functions using feed-forward neural networks. Each network comprises three layers of units: an input layer, a hidden layer, and an output layer. The output layer has only one output unit. Links between units are uni-directional, and can connect either a unit in the input layer to a hidden unit or the output unit, or a unit in the hidden layer to the output unit. Associated with each link is a *weight* of that link. The weight on a link going from the i'th unit to the j'th unit of the network is denoted $w_{j,i}$. Every unit in the input layer is connected to every unit of the hidden and output layers, and every unit in the hidden layer is connected to every unit in the output layer. The set of parameters W for the load-index function at a site consists of all the $w_{j,i}$ values, where i and j are units in the feed-forward network for that site.

Given the actual and desired outputs for a feed-forward neural network, the "back-propagation algorithm" can determine the appropriate modifications to the weights of that network. We use the 'vanilla' back-propagation algorithm available in a public-domain simulation package [5].

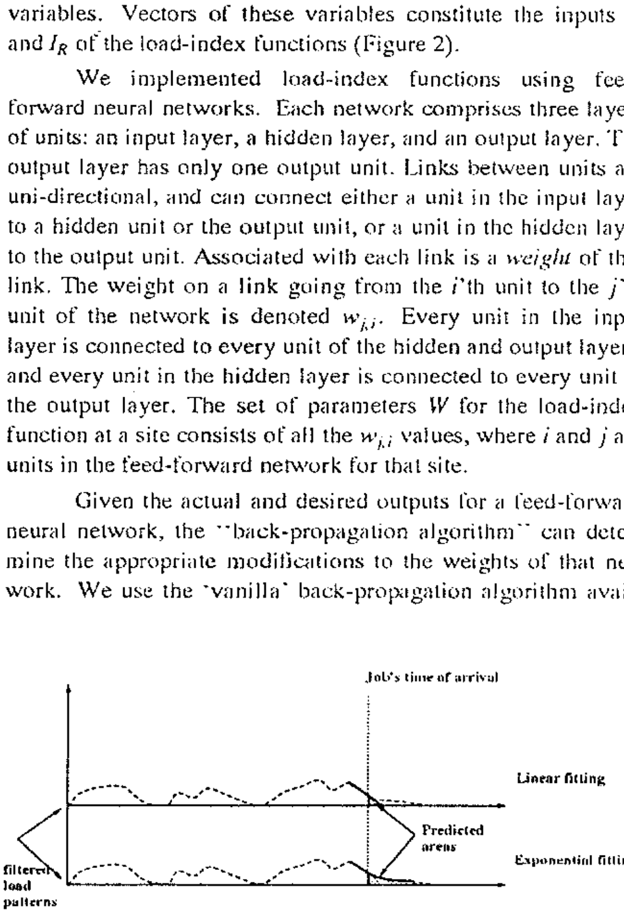

Figure 3. Trend extraction via curve fitting

The outputs of input-layer units are set at the beginning of each learning trial using filtered and extrapolated resource-utilization values, which are in turn derived from information contained in the training pattern chosen for that trial. The outputs of hidden-layer and output-layer units depend upon their *net inputs*. The net input of unit i (in the hidden or output layer) is given by $\sum_j w_{j,i} o_j$, where o_j denotes the output of unit j. The output of each unit of a feed-forward network is given by the *sigmoidal function* of its net input:

$$f(x) = \frac{1}{1 + e^{-x}}. \tag{8}$$

Let us denote by O_L the output of the left subnet; and, correspondingly, O_R, of the right subnet. In order to use the back-propagation learning procedure for training the subnets, we need to determine their desired outputs for every input. Let O_L^D denote the desired output of the left subnet; and, correspondingly, O_R^D, of the right subnet. Further, let us denote by O^A the actual output of the comparator network; that is,

$$O^A = O_L - O_R. \tag{9}$$

Given our objectives stated in Eq. (1), O^A corresponds to the left-hand side of that equation. Therefore, the desired output of the comparator network (denoted O^D) is given by the right-hand side of Eq. (1). That is,

$$O^D = O_L^D - O_R^D = \frac{C\left[\hat{l}_{b,f,s_1}, f\right] - C\left[\hat{l}_{b,f,s_2}, f\right]}{C\left[\hat{l}_{0,f,s_{ref}}, f\right]}. \tag{10}$$

The value of the objective function shown in Eq. (6) will be reduced if both O^A and O^D have the same sign. That can be achieved by driving their values closer together. Let us denote by E_{LMS} the sum (over all raw training patterns in the training set) of squared errors between the actual and the desired outputs of the comparator. That is,

$$E_{LMS} = \sum_{\tau} (O^A - O^D)^2. \tag{11}$$

We can minimize E_{LMS} by performing gradient descent; that is, by adjusting the outputs of the left and right subnets along their respective partial derivatives of error:

$$\Delta O_L = -\eta \frac{\partial E_{LMS}}{\partial O_L}, \qquad \Delta O_R = -\eta \frac{\partial E_{LMS}}{\partial O_R}, \tag{12}$$

where η is known as the *learning rate*. Hence, our training algorithm sets the desired outputs for the left and right subnets as shown in Eq's. (7a) and (7b).

One final detail needs to be worked out: ensuring the anti-symmetry of comparison. We resolve this problem by biasing the order of presentation of training patterns to the comparator. Raw training patterns are presented in pairs, one after another. If the first training pattern in the pair is $\tau_1(b, f, s_1, s_2)$, then the second training pattern must be $\tau_2(b, f, s_2, s_1)$. Thus, whenever index-functions are forced to predict that one completion time will be larger than another, they must (in the very next learning trial) predict that the latter will be *smaller* than the former.

5. EMPIRICAL RESULTS

This section presents our results on learning load indices for a system containing four sites. Each training pattern presented to the comparator network contains 40 projected resource-utilization values and 2 additional fields, $C\left[\hat{l}_{b,f,s}, f\right]$ and $C\left[\hat{l}_{0,f,s_{ref}}, f\right]$, where f identifies the foreground job, b identifies the background load pattern, and s the site at which f was executed.

5.1. *Evaluation Methods*

The training algorithm described in the last section was applied to 40×40×1 networks (*i.e.*, networks containing 40 hidden units each). In determining the number of hidden units, we used the popular rule of thumb that a network must contain approximately half as many weights as there are training patterns. The learning parameter η of the back-propagation algorithm was set to 0.001, and the momentum parameter set to 0.99.

We began by randomly assigning 10 percent of the jobs to the test set and the remaining 90 percent to the training set. The networks were trained using the training set, and their accuracy measured on the test set. Training and testing were applied alternately; during each *epoch* of training, the networks were trained on 1,000 randomly chosen comparisons from the training set, and tested on 100 randomly chosen comparisons from the test set. Each comparison involved two loading situations for the same job. Training was done in two stages. Stage 1 started with two identical networks (with random initial weights) for each site; these were trained to compare different loading conditions for the same job at the same site. In Stage 2, just the left networks from each site were further trained to compare different sites for the same job under the same system-wide load pattern. Unlike in Stage 1, the roles of the left and the right load-index functions (Figure 2) could not be reversed during Stage 2.

If we rank all available test-cases (all the different sites and loading patterns) of a job by their load indices, then we would like this ranking to have high correlation with the *true ranking* of these test-cases. The true ranking is, of course, the one induced by the measured completion times, $C\left[\hat{l}_{b,f,s}, f\right]$. One way to measure the correlation between two different rankings of the same data is to compute the *rank-correlation coefficient*, defined as:

$$r = 1 - \frac{6 \cdot \sum_i d_i^2}{n(n^2-1)}, \tag{13}$$

where d_i is the difference between the two different ranks of the i'th test-case, and n is the number of test-cases ranked. The value of r ranges between -1 and $+1$, where -1 indicates strong disagreement between rankings, 0 indicates no correlation, and $+1$ indicates strong agreement. This coefficient can be computed for each of the 58 jobs in the training set.

If there is *no relationship* between the two rankings, then the sampling distribution of r can be approximated with a normal distribution N(0, σ), where σ, the standard deviation, equals $(n-1)^{-0.5}$. If we let $z = r/\sigma$, then z will be distributed according to the two-sided standard normal distribution N(0,1). Using this fact, we can set up the following *null hypothesis*:

[H0] There is no relationship between the true ranking and the ranking induced by the load-index functions learnt using comparator networks.

We would like to *reject* H0 with high confidence. Given a confidence level α, we can look up the table of standard normal distribution to find an A such that $\Pr(|z| \leq A) \geq (1-\alpha)/2$. In particular, A takes on the values 2.58 for corresponding α value of 0.01 (99% confidence). Knowing A and n, we can determine *significance levels* for r. If, for instance, the absolute value of r exceeds its 99% significance level, then H0 can be rejected with 99% confidence. Our experimental results show that in all cases the null hypothesis can be rejected with 99% confidence [8].

5.2. *Sensitivity of Load Index to Job Size*

Load indices output by comparator networks at different sites are compared in order to determine the least-loaded site. Given that load balancing seeks maximum speed-up over local execution, we can evaluate the new load-index functions by the speed-up attained if each incoming job is scheduled at the least-loaded site. Figure 4 shows the results for our 4-processor system. Test jobs were introduced on a synthetic workload 2252 time units into the experiment, each time at a different site. The completion times of all the jobs and their checkpoints were recorded. Two policies were compared: (i) *opt*, which places each incoming job at the site with the optimum (least) completion time; (ii) min, which schedules incoming jobs at the site with the smallest load index. Assuming no overhead of remote execution, we calculated, for each test-case, the speed-up over local execution achieved using these policies. Figure 4 plots these speed-ups against the time of completion; it shows that while the site with the least load index behaves as well as the optimal site for short jobs, its performance is suboptimal for

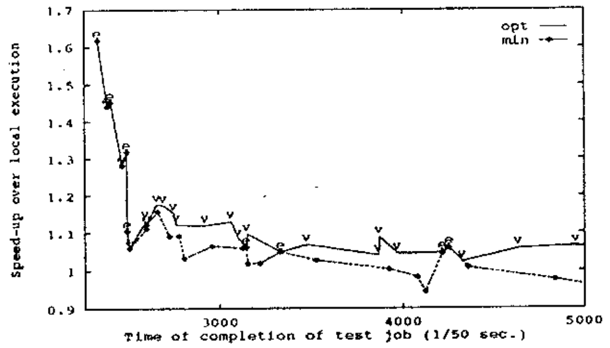

Figure 4. Performance under the worst-behaved load
Performance is computed for all test jobs under workload pattern 23; policies applied are *opt* and *min*.

long jobs. The letters on individual curves indicate the optimal site ('e' for elaine, and 'v' for 'vyasa'). The policy min runs all the jobs at elaine, the site with the least load index at the time of arrival; therefore, its performance is suboptimal only when a site other than elaine is optimal.

We evaluated min on all 10 jobs and all 24 load patterns. To relate job length with achieved speed-up, we created data-pairs $<l_i,e_i>$, where l_i is the length of the job and e_i, the corresponding speed-up. Figure 5 shows a contour plot of the probability of achieving certain speed-ups for jobs of certain length. The X-values represent speed-ups over local execution, and the Y-values, length of the job in seconds. Y-values range from 0.66 seconds to 439.94 seconds and, X-values, from 0.924524 and 4.92187. We used logarithmic scaling for the Y-axis. Nineteen contours, each connecting X-Y points having equal cumulative probability of speed-up, are shown; they divide up the space into twenty regions of equal (5%) probability. While speed-ups higher than 1.5 (more than 50% improvement over local execution) occur quite frequently for short jobs, they rarely occur for jobs more than 3 minutes long.

6. FINAL REMARKS

We have demonstrated automated learning of meaningful load-index functions from workload data. Using data collected by a synthetic workload generator, we trained neural networks to effectively compare alternative loading conditions for an incoming job. We collected workload and completion-time data on a network of 4 configurationally-distinct Sun 3 workstations, and performed off-line learning on a Sun Sparcstation 10, Model 20. The total down time was around 10 days; most of it was spent recording the 24 background loads and measuring completion times of 10 test jobs under these loads at each of the 4 sites. The training of comparator neural networks is fully automated; it requires no down time because it is performed off line (around 15 hours of CPU time).

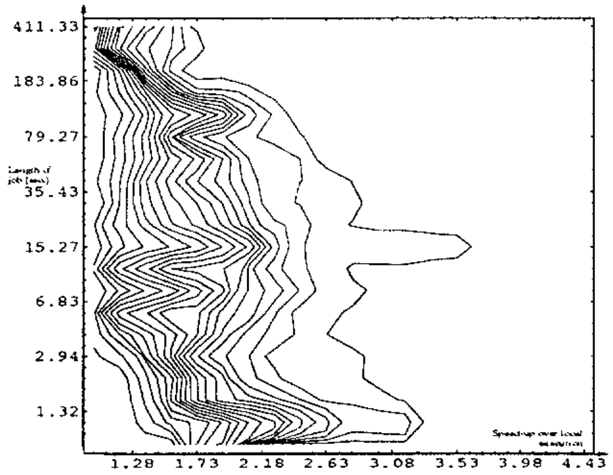

Figure 5. Policy: min; no overheads or delays.

REFERENCES

[1] 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.

[2] M. Berry et al., "The Perfect Club Benchmarks: Effective Performance Evaluation of Supercomputers," *International Journal of Supercomputing Applications*, vol. 3, no. 3, pp. 5-40, 1989.

[3] D. Ferrari, G. Serazzi, and A. Zeigner, *Measurement and Tuning of Computer Systems*, Prentice-Hall, Englewood Cliffs, NJ, 1983.

[4] D. Ferrari, "A Study of Load Indices for Load Balancing Schemes," pp. 91-99 in *Workload Characterization of Computer Systems and Computer Networks*, ed. G. Serazzi, Elsevier Science, Amsterdam, Netherlands, 1986.

[5] N. H. Goddard, K. J. Lynne, T. Mintz, and L. Bukys, Rochester Connectionist Simulator, Tech. Rep., Univ. of Rochester, Rochester, NY, Oct. 1989.

[6] 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.

[7] M. L. Litzkow, M. Livny, and M. W. Mutka, "Condor - A Hunter of Idle Workstations," *Proc. 8th Int'l. Conf. Distributed Computer Systems*, pp. 104-111, IEEE, 1988.

[8] P. Mehra, *Automated Learning of Load Balancing Strategies for a Distributed Computer System*, Ph.D. Thesis, Dept. of Computer Science, Univ. of Illinois, Urbana, IL 61801, Dec. 1992.

[9] P. Mehra and B. W. Wah, "Adaptive Load-Balancing Strategies for Distributed Systems," *Proc. 2nd Int'l Conf. on Systems Integration*, pp. 666-675, IEEE Computer Society, Morristown, NJ, June 1992.

[10] P. Mehra and B. W. Wah, "Physical-Level Synthetic Workload Generation for Load-Balancing Experiments," *Proc. First Symposium on High Performance Distributed Computing*, pp. 208-217, IEEE, Syracuse, NY, Sept. 1992.

[11] 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.

[12] S. Zhou, *Performance Studies of Dynamic Load Balancing in Distributed Systems*, Tech. Rep. UCB/CSD 87/376 (Ph.D. Dissertation), Computer Science Division, Univ. of California, Berkeley, CA, 1987.