International Journal of Systems Science, 1997, volume 28, number 11, pages 1077–1099

Automated learning of load-balancing strategies in multiprogrammed distributed systems

PANKAJ MEHRA[†] and BENJAMIN W. WAH[‡]

Dynamic load-balancing strategies for distributed systems seek to improve average completion time of independent tasks by migrating each incoming task to the site where it is expected to finish the fastest: usually the site having the smallest load index. SMALL is an offline learning system for developing configuration-specific load-balancing strategies; it learns new load indices as well as tunes the parameters of given migration policies. Using a dynamic workload generator, a number of typical systemwide load patterns are first recorded; the completion times of several benchmark jobs are then measured at each site, under each of the recorded load patterns. These measurements are used to train comparator neural networks simultaneously, one per site. The comparators collectively model a set of perfect load indices in that they seek to rank, at arrival time, the possible destinations for an incoming task by their (not yet known) respective completion times. The numerous parameters of the decentralized dynamic load-balancing policy are then tuned using a genetic algorithm. We present experimental results for a mix of scientific and interactive workloads on Sun workstations connected by Ethernet. The policies tuned by SMALL are shown to exploit idle resources intelligently and effectively.

1. Introduction

Dynamic load-balancing strategies seek to improve the performance of distributed systems by migrating workload from heavily-loaded to lightly-loaded sites. Our distributed system comprises a network of multiprogrammed workstations, each having private CPU, memory and possibly disks. Non-private resources—the network and disks having shared file systems—are transparently shared among the processes at different sites; private resources can be accessed by local processes only. We consider independent tasks that can arrive at any of the sites and can be scheduled for execution at any of the sites. The focus of this paper is an automated tuning of a generic decentralized dynamic scheduling policy representative of those used in currently available load-balancing systems (Baumgartner and Wah 1989, Litzkow *et al.* 1988, Zhou 1987a).

```
SENDER-SIDE RULES($s$)
    $Destinations = \{site: Load(site) - Reference(s) < \delta(s)\}$
    $Destination = Random(Destinations)$
    IF $Load(s) - Reference(s) > \theta_1(s)$ THEN Send($Destination$)
RECEIVER-SIDE RULES($r$)
    IF $Load(r) < \theta_2(r)$ THEN Receive
```

Figure 1. Parametrized load-balancing policy studied (δ, θ_1 and θ_2 are non-negative real numbers).

Figure 1 shows the policy we study in this paper. It is applied in a distributed fashion as follows. An incoming task at site s triggers its sender-side rules (SSRs). Using the outputs of *Load*()[§], the site-specific load-index functions, the SSRs determine *Destinations*, the set of sites whose load falls within a $\delta(s)$-neighbourhood of

Accepted 12 December 1996

[†] Tandem Labs, 19633 Vallco Pkwy., Loc. 1-27, Cupertino, CA 95014, U.S.A. e-mail: mehra_pankaj@tandem.com.

[‡] Coordinated Science Laboratory, University of Illinois, Urbana-Champaign, Urbana, IL 61801, U.S.A. e-mail: b-way@uiuc.edu.

[§] We do not consider the case of multiple workload indices per site, which would be useful if there were many different classes of jobs, and sites offer different processing capabilities for different classes. For the only class of jobs studied in this paper, the trade-offs afforded by multiple indices are automatically accommodated by our empirically determined multi-resource load indices.

Reference(*s*). *Reference*(*s*) is either 0 or *MinLoad* (denoting the smallest *Load* value). If *Destinations* is not the empty set, then one of its members (say, *r*) is picked at random; such randomization avoids instability in distributed scheduling. Depending upon the value of *Reference*(*s*), the SSRs compare either the absolute or the relative value of *Load*(*s*) against the threshold $\theta_1(s)$. (θ_1 prevents migration when the expected gains in completion time are not large enough to offset the costs of migration. If *Load*(*s*) is large enough, then the rule for sending succeeds, and the SSRs invoke the receiver-side rule (RSR) at site *r*; otherwise, the job is executed locally at site *s*. The RSR at site *r* compares *Load*(*r*) against threshold $\theta_2(r)$. (θ_2 helps counter bad decisions by senders who overestimate their own load.) The request from the SSRs at site *s* is turned down if *Load*(*r*) is too high; in this case, the job is executed locally at site *s*. Otherwise, the request is accepted, and site *s* is given permission to migrate its job to site *r*.

The performance of a task under this policy is defined as the speed-up in its completion time due to load balancing. The performance of the policy is defined as its expected performance on an incoming task under typical systemwide loading conditions, provided the policy is used to determine each task's site of execution.

The performance of a task is a function of its site of execution; differences in performance of sites may stem from either (*a*) configurational factors, such as processor speed, amount of memory, disk space, number of functional units, size of cache and RAM, and speed of network interface; or (*b*) load variations, i.e. differences in the number and type of tasks executing at different sites. While configurational differences are static, load-related differences are known to be highly variable and generally unpredictable in the long run.

Because of the highly dynamic load-related variation in the performance of different sites, long-running jobs need to be rescheduled as the relative performance of sites changes. We assume first that executing tasks will periodically come up for rescheduling and further that there is minimal site-specific state at the time of rescheduling so that the cost of migrating a rescheduled task is no different than the cost of migrating a fresh incoming task. Long-running jobs are therefore modelled by a sequence of independently scheduled tasks in our experiments (§4.2). As a simplifying assumption, the constraint on the site of arrival of rescheduled entities—that each successive task in the sequence must arrive at its predecessor's site of execution—is not enforced.

To perform well, a policy must migrate incoming tasks to remote sites provided there is a speed-up greater than 1—relative to local execution. In this light, the policy can be seen as an exercise in cost–benefit analysis. Our policy uses the site-specific *Load*() functions as indicators of each site's completion time and assesses the possibility of speed-up by comparing the loads of different sites. The calculation of *Load*() values adds to a site's CPU load; likewise, their propagation across sites adds to network traffic. Thus, there is a short but finite interval between successive calculations of the load index; this, and the propagation delays, introduce inaccuracies in the values of *Load*(). Moreover, the cost of migrating tasks is not evidenced from simple comparison of load indices but adds to remote completion time of tasks. Imperfect indices and migration costs thus offset the benefits of remote execution; our policy seeks to model their impact on performance using policy parameters $\delta, \theta_1, \theta_2$ and *Reference*.

The performance of our policy depends critically on the design of load–index functions as well as the setting of policy parameters. Given our performance objectives, our policy will perform well on an incoming task: (i) if the comparison of load indices across sites accurately predicts the benefits of migrating that task to a remote site; and (ii) if the policy parameters satisfactorily model the costs of task migration.

Both load indices and cost factors have configuration-dependent and load-dependent components. For instance, the time to migrate a task has a configuration-specific minimum cost as well as a load-dependent component. Since load patterns are installation-specific, indices and parameters need to be tuned to the characteristics of the configuration as well as the installation. Manually designing load-balancing policies meeting the above criteria is therefore a tedious data-intensive task that must be repeated for every configuration and installation.

The SMALL (Systematic Method for Automated Learning of Load-balancing strategies) system automates our design task. It includes an experimentation environment called DWG (Dynamic Workload Generator) which allows reliable measurement of task completion times under realistic and reproducible loading conditions. (See our earlier work, Mehra 1992, Mehra and Wah 1995a, b.) This paper focuses on the learning component of SMALL, which uses the load measurements and completion times provided by DWG to design new load indices and to set policy parameters rationally.

The rest of this paper is organized as follows. Section 2 describes the architecture of SMALL. Section 3 surveys related work. Section 4 summarizes the capabilities of DWG. Section 5 presents a neural-network-based approach to design of load indices. Section 6 describes systematic setting of policy parameters using a novel combination of gradient-based and genetic learning techniques. Empirical results of our experiments are the subject of §7, and §8 concludes this paper.

2. The SMALL system

Systematic improvement in load-balancing performance requires systematic exploration of the large space of alternative parameter settings. In the absence of much prior knowledge of the load-balancing domain, our approach relies on conducting experiments and using the observed performance of such experiments as feedback for further exploration. This empirical approach to policy learning (Mehra and Wah 1990) operates in a cycle of decision making, performance measurement, and modification. The inputs for each decision are a job, its site of arrival, and a window of measurements constituting the systemwide load pattern prevailing at the time of the job's arrival; the output of each decision is a destination for the incoming job. The speed-up (with respect to local execution) attained by executing the job at the selected destination is our measure of performance. Modifications are made to policy parameters so that future decisions (using the modified parameters) will yield a better performance in similar situations.

Similar empirical methods are employed for learning load indices. The experimental data are in the form of before–after pairs, where the first item in each pair is a window of periodic measurements constituting a site's load pattern and the second item is the measured completion time of a task that began execution at the end of the window. Using two different data pairs involving the same task but different sites, our learning system simultaneously modifies the load–index functions at both sites so that, in similar situations in the future, the outcome of comparing load–index functions would likely be the same as that of comparing measured completion times.

Learning of new policies and indices is essentially only immediately following a significant change in configuration or installation. At other times, there is no need to slow down load balancing by incurring the extra overhead of measurements and modifications needed by learning processes. Our learning system, SMALL, is therefore designed as a suite of pluggable components which, when plugged in, tune a system's policies and indices; and, when unplugged, incur no overhead. We call the time when SMALL is plugged in as its *learning phase*, the time when SMALL needs exclusive use of the system for gathering data as its data-collection phase, and the time when SMALL is unplugged as its *application phase*. Isolating these phases temporally and optimizing each one requires careful experimental design.

2.1. *Offline data collection and learning*

Data collection is said to occur offline (with respect to learning) when all experimentation and data gathering necessary for measuring the performance of decisions is carried out only once, ahead of time. It is said to occur online if experimentation and learning alternate. In a similar vein, learning is said to occur online if load-balancing decisions alternate with learning processes; offline, if the learning phase strictly precedes the application phase.

Offline data gathering avoids repeating identical experiments. It measures the completion time of each task, under each systemwide load pattern, at each site of the distributed system. In any distributed system, the number of possible destinations for an incoming task is fixed (equal to the number of sites in the system). The number of possible policies one might need to evaluate is, on the other hand, quite large (exponential in the number of distinct parameters). Multiple policies that schedule the incoming task of an experiment to the same destination can be evaluated using the data collected in just one offline experiment! Considering that each experiment may take several minutes, the savings in time achieved by our experimental design are substantial.

Offline data gathering makes even more sense when we consider the load–index learning problem where alternative destinations for the same task need to be compared under identical loading conditions. Since one occurrence of a load pattern in an online setting permits experimentation with only one destination, offline data collection is an inevitable requirement of load–index learning. In this, DWG plays a vital role by allowing us to recreate a systemwide loading pattern as often as needed.

The choice between offline and online learning is simpler. Since the former incurs no overhead during the actual operation of the load-balancing system, it is the preferred approach. The only problem is that the population of tasks or the load patterns used in the offline learning phase may not resemble those encountered during the application phase. In that case, using the policies and indices developed by the learning phase will cause unexpected behaviour in the application phase. Therefore, offline learning is valid only when the short-term probabilisitic characteristics of task and load patterns are stationary with time.

Online learning systems adapt continuously so they gracefully adapt to changing task characteristics and changing load patterns. But online learning brings with it several unsolved machine learning problems, the principal ones being the temporal credit-assignment (TCA) problem (Sutton 1988) and the exploration–convergence dilemma (ECD) (Barto *et al.* 1990). The TCA problem arises whenever multiple decisions contribute to a single performance measurement; this is exactly what would happen if policy learning were to proceed concurrently with the application phase. Several different scheduling decisions may be made between scheduling an incoming task to a remote site and measuring its completion time. Other sites may have affected the measured completion time by deciding to schedule tasks at or away from the remote site. The ECD problem concerns the weight assigned to new information in relation to information accumulated from past experiments. The offline learning method employed in SMALL makes certain simplifying assumptions (see below) to avoid TCA. It overcomes ECD by transferring that responsibility to a human system administrator who can schedule a new round of offline learning whenever he or she either changes the system configuration or observes significant change in an installation's resource-utilization patterns.

SMALL avoids TCA by evaluating each decision in isolation. In reality, the load patterns encountered during load balancing are a result of earlier scheduling decisions. In such a sequential learning scenario, the number of possible load patterns at the time of scheduling an incoming task is combinatorially explosive. We make the simplifying assumption that the finite set of load patterns collected offline and ahead of time is representative of the exponentially large set. We need to experiment with a wide variety of loading conditions in any case; therefore, this assumption does not pose any problems.

It follows that if offline learning is carried out whenever there is a change in configuration or utilization pattern, and if load-balancing experiments are carried out under a wide variety of realistic and reproducible loading conditions, then the policies and indices developed in the learning phase can be used by the application phase. In this case, both learning and data collection can be taken offline.

In order to make such offline learning an effective and scalable strategy, it is crucial that the bulk of the knowledge learned earlier on a distributed system be retained whenever the system grows. This means that load-balancing strategies learned on a smaller system must continue to be applicable when the system is expanded or upgraded and new and possibly faster computers added. Hence, growth should not involve bringing the entire system down and relearning existing strategies. This is indeed the case when workload characteristics on existing sites are unaffected by upgrades or the addition of new sites, which in turn is true of the systems available today because shared resources, such as networks and file servers, are designed for growth without loss of performance.

Finally, in the extremely infrequent case when upgrades of shared network resources (such as the physical network medium and file servers) can affect the workload characteristics of existing computers, it is possible to have new workload conditions occur that were not foreseen by earlier learning. In this case, it will suffice to apply SMALL to retrain the load-balancing policies of the entire system. This step can be prevented if workloads of a wide variety, whether prevalent or not, have been used in earlier learning of load-balancing strategies.

2.2. *The five phases of learning in SMALL*

Figure 2 outlines the operation of SMALL, and also indicates the relevant sections of this paper. Figure 3 shows our learning system. The three key components of SMALL include a workload generator (DWG), a trainable load–index function (comparator network), and a mechanism for tuning the parameters of given

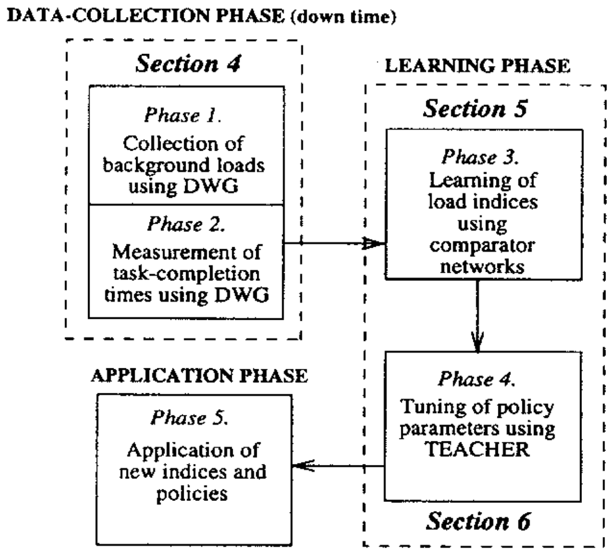
DATA-COLLECTION PHASE (down time)
Section 4
Phase 1. Collection of background loads using DWG
Phase 2. Measurement of task-completion times using DWG
LEARNING PHASE
Section 5
Phase 3. Learning of load indices using comparator networks
Phase 4. Tuning of policy parameters using TEACHER
Section 6
APPLICATION PHASE
Phase 5. Application of new indices and policies

Figure 2. The five phases of SMALL's operation.

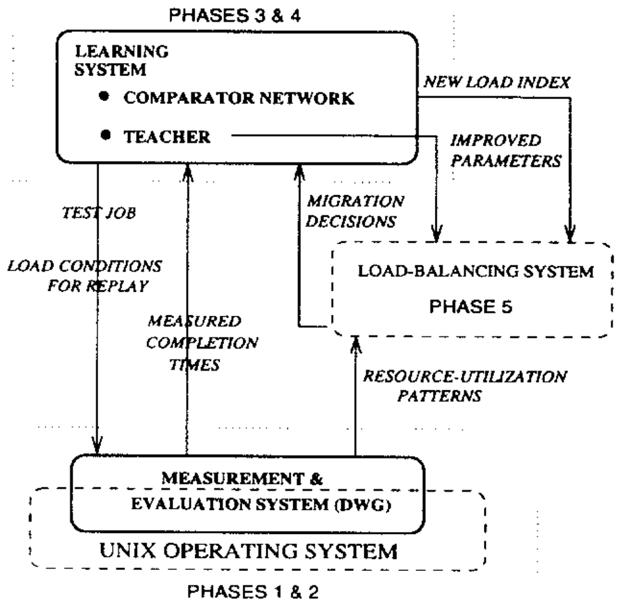

Figure 3. Overview of SMALL.

load-balancing policies (TEACHER). DWG resides partly inside the operating-system kernel and partly at the process level, and forms a layer below the given load-balancing system. The learning system sits on top of the load-balancing system, and can access and modify various policy parameters.

The first two phases of SMALL are for offline recording of realistic load patterns and for offline measurement of completion times for a representative set of tasks under these load patterns. As discussed in §2.1, this phase needs dedicated use of only the new computers to be upgraded or added to the distributed system; existing computers, whose behaviour under various workloads has been already recorded, do not have to be taken offline.

In Phase 1, we recorded both naturally occurring load patterns as well as patterns likely to occur at typical decision points during the application phase; the latter were generated by running a random selection of representative tasks on top of typical workloads and then recording the resulting load patterns. In Phase 2, we ran each of our representative tasks, once at each site, under each of the previously recorded workload patterns (replayed using DWG). DWG allowed us to initiate foreground test jobs at precise times and to measure accurately their completion times. As explained elsewhere (Mehra and Wah 1995b), DWG's design ensures highly accurate workload generation, even in the presence of foreground processes not present at the time of recording the workload.

Phases 3 and 4 are for offline learning of workload indices and tuning of policy parameters based on completion times of different tasks (measured in Phase 2) at each site and under each load pattern. Workload indices and policy parameters of existing sites in the system may need to be relearned. However, since learning in these two phases takes place offline, there is no need to take any of the sites offline. In fact, learning may take place at a computer server (a supercomputer, for instance) not even on the distributed system.

In Phase 3, our learning system defines load–index functions, one per site; these can be used for converting the multivariate resource-utilization patterns collected during the first phase into univariate load indices. In Phase 4, these load indices, along with the completion times measured during Phase 2, are used for tuning the parameters of given load-balancing policies. After this phase, the new parameter values found by the policy-learning system can be plugged into the load-balancing policy.

The results of learning are a set of load–index functions, one at each site, for computing a scalar quantity that ranks alternative sites for local incoming jobs. In Phase 5, the given load-balancing policies use the load indices, along with the tuned policy parameters, to determine the most appropriate site for each incoming job.

The load-balancing system (Fig. 3) implements the policy shown in Fig. 1, and converts the primitive resource-utilization measurements into the more meaningful load indices. It includes supports for communicating load indices among the sites, as well as for computing abstract performance metrics such as *MinLoad*, which denotes the minimum predicted load index. (See the paper by Baumgartner and Wah 1989 for an efficient implementation of such a load-balancing system.)

We assume the existence of a job-execution facility at each site, which can be invoked either locally or remotely. In the UNIX system, such functions are provided by remote shells and remote execution facilities. We also assume that the underlying operating system supports process initiation and detection of process termination.

3. Related work

Load balancing seeks to obtain significant speed-up over local execution. Since each incoming task has a finite set of possible destinations, one would like to use the workload information at each site to compare alternative destinations in terms of their expected speed-ups over local execution.

A common way to compare alternative destinations is manually to specify a formula for computing *Load*, the load index, as a function of the current and recent utilization levels of various resources. The design of a good load index and an effective load-balancing policy using these indices is difficult, as it is sensitive to machine-specific hardware characteristics and prevalent load patterns. In this section, we describe current approaches to these problems, discuss their deficiencies, and compare them with SMALL's approach.

3.1. *Existing methods for computing workload indices*

Many existing methods for computing workload measures are based on simplified queueing models of computer systems (Ferrari *et al.* 1983, Ferrari 1986, Zhou 1987b). They model resources as servers and use the lengths of associated queues as determinants of average delay/throughput. A further approximation leads to an architecture-independent and universal function known as UNIX-style load average (hereafter, load average), which is an exponentially smoothed average of the total number of processes (including the process in execution) waiting for the CPU response. Although this measure may be a reasonable performance indicator for purely computer-bound tasks, its design does not consider the other resources of a site, such as cache size,

Table 1. Typical performance-indicating variables available in UNIX-like operating systems

Performance metric	Type of information available
Number of context switches Number of system calls Number of device interrupts Number of pages swapped in/out	Total over 1 s period, Average over 5 s period, Total since boot
Number of processes swapped in/out	Total over 5 s period, total since boot
Number of processes ready to use the CPU Number of processes waiting for disk Number of processes waiting for free memory Number of swapped-out active processes Memory pages used by all the processes in core Memory pages used by active processes only	Instantaneous value, computed every 5 s
Number of free memory pages	Instantaneous value and 60 s average
Number of cache flushes	Total over 1 s, average over 5 s, and total since boot
Times spent in the different CPU states: Idle; in OS functions; in user programs; and, in low-priority user programs	Updated using instantaneous values sampled once every 20 ms
Amount of data transferred on each disk	Total, asynchronously updated at each transfer
Rate of data transfer	Calculated once every second or slower
Number of characters input/output from/to terminal devices	Total, asynchronously updated during terminal input/output operations
Number of packets input/output on each network interface	Total, asynchronously updated at the time of packet transfer
Number of collisions (for CSDMA interfaces)	Total, updated when collisions are detected)
Per-process statistics: time spent in user mode, time spent in system mode, resident-set size, numbers of messages received and sent, number of signals received, number of context switches, numbers of process swaps and page faults.	Timing statistics sampled once every 10–20 ms; the whole interval charged to the process in control of CPU. Other statistics updated asynchronously at the time an event happens.

memory space, speed of peripheral devices, and network traffic. Further, it does not provide a meaningful comparison of loading situations across sites, especially when the distributed system is configurationally heterogeneous. The disparity between the actual load and that predicted by the load average can be best illustrated when a fast and a slow workstation coexist in the same network. In this case, it may almost always be better to execute an incoming job at the fast workstation even though its load average may be higher.

A better workload measure is needed to account for multiple resources in a computer—such as memory, disk, and network—and for meaningful comparison of different sites of a configurationally heterogeneous system.

Table 1 lists a variety of performance metrics in a typical workstation-based operating system. These are obtained by software instrumentation in the operating system. However, several of these metrics are unsuitable for inclusion in a load index because the overhead associated with their measurement preclude frequent sampling; this category includes a variety of process-level metrics, which are usually sampled only once every 5 s. 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. Others, such as rate of data transfer, are fixed for a given site, and can be implicitly modelled by the 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 to allow frequent measurements; (ii) representative of load on all the resources of contention; and (iii) can be measured and controlled independently of each other. In the past, Zhou (1987a, b) has considered resource-queue lengths (the number of processes waiting for CPU, disk, and memory) in designing heuristic formulae to represent load indices. There are three reasons why using queue lengths alone is not adequate: (i) keeping track of all changes in queue lengths incurs too much overhead; (ii) using instantaneous queue lengths at sampled instants does not adequately represent cumulative workloads in between instants; and (iii) the utilization of some resources, such as memory occupancy, is not measured by queue lengths.

In our study, we use a combination of cumulative and instantaneous utilization on the four basic resources—CPU, memory, disc, and network—that form a useful set of performance metrics satisfying all three criteria. CPU and memory utilization are measured easily by instantaneous occupancy; we do not keep track of all changes in the states of CPU and memory due to the high overhead involved and the lack of a high-resolution real-time clock. Disk and network information is measured by cumulative traffic. The chosen metrics are shown in bold font in Table 1. We show in § 5 that a comparator neural network can be used to assimilate these four different inputs into a meaningful load index.

3.2. *Automated learning of load indices*

Ideally, given the loading conditions prevailing at the different sites, one would like to rank the sites by their respective completion times for an incoming job. A job's completion time depends on its requirements as well as on the characteristics of other jobs running simultaneously on the system. Two factors make it hard to predict completion times of jobs in a distributed multiprocessing system. (i) It is difficult to analyse a program when many of its execution paths are data-dependent and known only at run time; (even when all these paths are known, it is unclear how to translate such information into primitive resource requirements), (ii) Events in a distributed system are asynchronous (such as a user typing on a keyboard), and it is not possible to determine the best load-balancing decision without knowing future workload.

Although exact prediction of completion times is difficult, it is possible to characterize background workload by different workload indices for jobs of a few different classes, and to identify the class of each incoming job. For example, while the index for memory-intensive jobs may pay more attention to memory availabilty, the one for compute-bound may stress CPU availability. However, using these different load indices presupposes knowledge of the class to which each incoming job belongs; such knowledge might be available in domains where the same programs are used over and over again (Devarakonda and Iyer 1989). In our study, we assume no prior information about jobs (or all jobs belong to one class); we, therefore, restrict our attention to the case of one load-index function per site. Our learning method can be easily extended to the case with multiple job classes.

Notice that our goal in designing a load-index function is merely to compare alternative destinations for the same job. Therefore, we will attempt to determine only a relative (site-specific and configuration-specific but not job-specific) measure of completion time. It would, therefore, suffice to predict the relative completion times of a job at different sites. Since we need a point of reference, we choose an idle file server as such, and predict the *speedup* of an incoming job on a given machine relative to the reference idle file server, given only the loading conditions at the time of the job's arrival. Of course, this prediction will succeed only if current workload is a good indicator of future workloads. Therefore, we need to assume that resource-utilization patterns exhibit significant autocorrelations; such correlations may become insignificant as one predicts further into the future. In § 5, we list additional assumptions under which relative completion times can be predicted independently of tasks.

The prediction of a job's speed-up at a site, as described above, is still very difficult without complete knowledge of jobs and workload behaviour. Instead, we can predict, for each pair of sites, the difference between their speed-ups. That is, we are interested in finding a site that will maximize speed-up for the incoming job, without necessarily knowing the actual speed-up value for that site. Moreover, we judge the quality of load indices by the percentage of correct comparisons. Hence, we can stop refining the individual index functions when the percentage of correct comparisons exceeds an acceptance threshold. We need to ensure that the anti-symmetry of comparison—whenever A is better than B, B is worse than A—is preserved during learning. This issue of learning to predict differences of relative completion times is further motivated and resolved in § 5.

Learning programs, such as the one used to tackle the problem studied in this paper, may derive their power from two very different sources: domain knowledge and data. The index-learning task in load balancing is knowledge-lean because the exact time-variation of background workload, nor the exact relationship between load and completion time, is known ahead of time. Thus, empirical methods of strategy learning (Mehra and Wah 1990), which infer the missing relationships from experimental data, are the only means available to us for learning load indices in a knowledge-lean environment.

3.3. *Automated learning of load-balancing policies*

A substantial amount of work has been done in the past on load balancing. Table 2 surveys approaches to

Table 2. Previous research on intelligent load balancing

Approach	Example	Comments
Classifier systems	Midgard (Sannier and Goodman 1988)	No learning
Bayesian decision theory	BDT (Stankovic 1985)	Hand-coded utility
Learning automata	Probabilistic (Glorioso and Colon-Osorio 1980)	Static policy
Reinforcement learning	Stochastic Learning Automata (SLA) (Mirchandaney and Stankovic 1986)	Analytical performance; No multi-programming
Team theory	Utility-driven control (Pasquale 1988)	No learning
Planning	Expert system (Kumar *et al.* 1987)	No learning
Empirical learning	Gradient-based approach (Pulidas *et al.* 1988)	Analytical performance

intelligent load balancing. Only two of these approaches (SLA and the gradient-based approach) use learning.

Our approach differs from these approaches in two important ways. First, previous studies on learning load-balancing policies do not use measured performance as feedback. Instead, they use simulation models whose states can be evaluted independently and immediately. These simulation models are crude approximations of real distributed multiprocessing systems where events are asynchronous and could be caused by interacting sources in the system. Second, previously studied learning methods have only been demonstrated in simulated environments where arrival processes are Poisson and computer systems can be modelled by product-form queueing networks. The effectiveness of their approach is hard to verify for heterogeneous systems with auto-correlated workloads.

We choose to learn load-balancing policies based on actual completion times of jobs running concurrently with background workload on a real multiprocessing system. Moreover, we choose to evaluate the policies learned by our approach by comparing, for each job, its measured completion times on each of the machines in the system.

4. DWG: a tool for synthetic workload generation

This section reviews the goals and capabilities of DWG, a physical-level synthetic workload generator, that efficiently, accurately and repeatedly replays a wide variety of realistic and synthetic workload patterns. DWG was designed to measure and control the utilization of CPU, memory, disk, and network. It supports the following 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 behaviour generating the measured load, we have implemented most of DWG in the kernel (Mehra 1992, Mehra and Wah 1995a, b).

4.1. *Design considerations*

DWG was designed to efficiently capture the low-level resource-utilization information in a distributed multiprocessing system, and to replay the collected information as a synthetic workload accurately and with little overhead, so that load-balancing experiments can be carried out. We had two key considerations in designing DWG.

The first purpose of DWG is to measure, for each given foreground job, its completion time at each of the S sites in the system, under each of the background load patterns. For a given background load pattern, it is not possible to run the given job simultaneously at all sites because its execution at one site may affect the amount of shared resources (such as shared network and file system) available at another site. As a result, it is essential that we reproduce the same background load pattern S times so that the given foreground job can be executed at each of the sites.

Second, when a test job is executed at a site in the presence of background jobs, the proportion of computational resources allocated to the background jobs will be reduced. Test jobs and background jobs affect each other by competing for resources. For real workloads, such competition is resolved by a resource scheduler. While a background load pattern is being recorded, the process population generating that load has compete control over a site's resources. If a test job were introduced on top of such a workload, it would take away some resources from the background-process population, thus altering its resource-utilization pattern. Therefore, when test jobs are introduced on top of generated workloads, their impact on the workload generator needs to be carefully considered. Such an interaction creates a feedback from the experiment to the experimentation environment; it is an important characteristic of the workload-generation problem addressed by DWG.

The esssential component of DWG is, therefore, the one that handles the interaction between foreground jobs and replayed background load patterns. There are two alternative approaches for modelling this interaction.

(*a*) *Process-level view*. This amounts to representing asynchronous events, such as keyboard interrupts and context switches, at the process level. It entails (i) recording complete resource-utilization information for a large number of processes; (ii) modelling all the complex interactions between processes and interrupt-handling routines of the operating system; and (iii) precisely recording the timing of interrupts and context switches, and replaying them at the same fine grain. Obtaining and regenerating such workloads is impractical because it requires costly hardware instrumentation and high-resolution timers for driving the generator.

(*b*) *Physical-level view* (Ferrari 1984). This entails the design of a synthetic background-workload generator that models the effects of running a test job on a background workload. There are many synthetic workload generators at the physical level; however, they are unsuitable for our purposes because they ignore the feedback from foreground jobs to the workload generator. To generate the necessary feedback, we need to artificially model process-level interactions between a recorded background load and a foreground job, and use our model to adjust (dynamically) the amount of replayed load while some foreground job is active. One approach is to modify the process scheduler of the operating system that controls the amount of resources allocated to the foreground job in the presence of the background load. Another approach is to design a set of doctoring rules that modify the amount of resources allocated to the background local generator when a foreground job is active.

We have chosen the latter approach because it avoids modification of the process scheduler. 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, which calculate, at each clock interrupt, the amount of background workload to be generated for each resource. For the CPU, the amount of work is expressed as a fraction of the interval between successive interrupts. Memory load is expressed as the number of pages to be occupied until the next interrupt. Disk and network load are expressed as the number of disk transfers and the number of network packets. Parametrization and tuning of our doctoring rules, and statistical comparison of doctoring rules, are described elsewhere (Mehra 1992, Mehra and Wah 1995a, b).

4.3. *Collection of workload patterns*

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^{-1} Ethernet.

Although the testbed we used in collecting load patterns is small, we can apply a technique called *cloning* to study larger systems. In cloning, the load patterns for the same site at different times can be used to simulate a load pattern on multiple copies of that site. This method creates load patterns distributed in space from load patterns distributed in time. Since test jobs behave identically across different clones under the same loading pattern, and since workload characteristics of existing computers are assumed to be unchanged by the addition of new sites, completion-time measurements need not be repeated. By increasing the scale of systems, cloning creates greater opportunities for load imbalance and therefore, higher optimal speed-ups. Of course, cloning is limited to cases in which each clone must have hardware identical to some site already in the testbed.

We used ten test jobs: three to sort records of various sizes with various amounts of memory, two to uncompress files of varous sizes, and five Perfect Club benchmarks (Berry *et al.* 1989) (FLO52Q, TRFD, QCD, TRACK and DYFESM).

To study pre-emptive scheduling of jobs, we inserted checkpoints into each test job. Each checkpoint resembles a pre-emption point; consequently, we can treat a segment from the beginning of an instrumented job to each of its checkpoints as an independent job. Although this technique creates a large database of training patterns, it could be rendered ineffective by different segments on the same test job having similar resource requirements. This, however, does not affect the accuracy of our final result, as we are only interested in average speed-up as a function of job length. (Speed-ups 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. Our final database included the first checkpoint of each foreground job and, thereafter, the next checkpoint that took at least 50% 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 four sites and under 24 different background load patterns, for a total of 5324 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 ten test jobs described above. The remaining four were designed to create surprises for the load-balancing system. We started with a real, 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. (Note that surprises packed in the beginning are less likely to happen and are not studied.) We expect that, because of these load patterns, speed-ups 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. These surprises are meant to illustrate the loss of performance if long jobs are migrated to less loaded sites and never pre-empted from there even when the horizon of prediction of the load index elapses.

Figure 4 compares the load patterns from the controlled experiment (in which real foreground and background jobs were run) against the corresponding patterns from the generation experiment (in which a real foreground job was run on top of the corresponding background load generated by DWG) for one of our experiments. Visually, we can confirm that the generator reproduces the utilization patterns rather well using its tuned doctoring rules. Results of other experiments are similar to the one depicted here (Mehra 1992, Mehra and Wah, 1995, 1995b).

5. Automated learning of workload indices

5.1. *Design goals*

Our goal in load-index learning 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 the alternative destinations of an incoming job by these functions is consistent with the ranking induced on those same destinations by the measured completion times of that job. The following symbols are first defined.

S: Set of S sites in the network.

F: Set of F foreground test jobs. V 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 sites in the system, and that our unit of time equals 20 ms, which is the period between successive interrupts of the real-time clock on a Sun 3 workstation.)

B: Set of 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. ϕ 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 instant in the window, for the 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 $L_{b,\phi,s}$ 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 derived from recent behaviour of loads on different locally accessible resources at s for background load b before foreground job f is started at t.

$F_s^W(\hat{L}_{b,f,s}, f)$: Value at site s's load-index function. W denotes the weights of F_s and will be omitted whenever obvious.

$C_s^W(\hat{L}_{b,f,s}, f)$: Completion time of f at s under $\hat{L}_{b,f,s}$, where the role of superscript W is as explained above.

$W_s^W(\hat{L}_{b,f,s}, f)$: Time spent by f waiting for resources at s under $\hat{L}_{b,f,s}$, where the role of superscript W is as above.

The objective of load-index learning, then, is

$$F_s(\hat{L}_{b,f,s_1}, f) - F_s(\hat{L}_{b,f,s_2}, f) = \frac{C(\hat{L}_{b,f,s_1}, f) - C(\hat{L}_{b,f,s_2}, f)}{C(\hat{L}_{\phi,f,s_{\mathrm{ref}}}, f)}, \tag{1}$$

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 in (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 the load-index function but rather in other parameters of the load-balancing policy to be discussed in §6.

Let us examine whether the right-hand side of (1) depends on the job f being scheduled. Consider the expression $C(\hat{L}_{b,f,s}, f)/C(\hat{L}_{\phi,f,s}, f)$, which is commonly known as the stretch factor (Ferrari *et al.* 1983); its denominator is known as the service time of job f at site s. The service time depends only upon f and s, but not on the b. Further, the completion time $C(\hat{L}_{b,f,s}, f)$

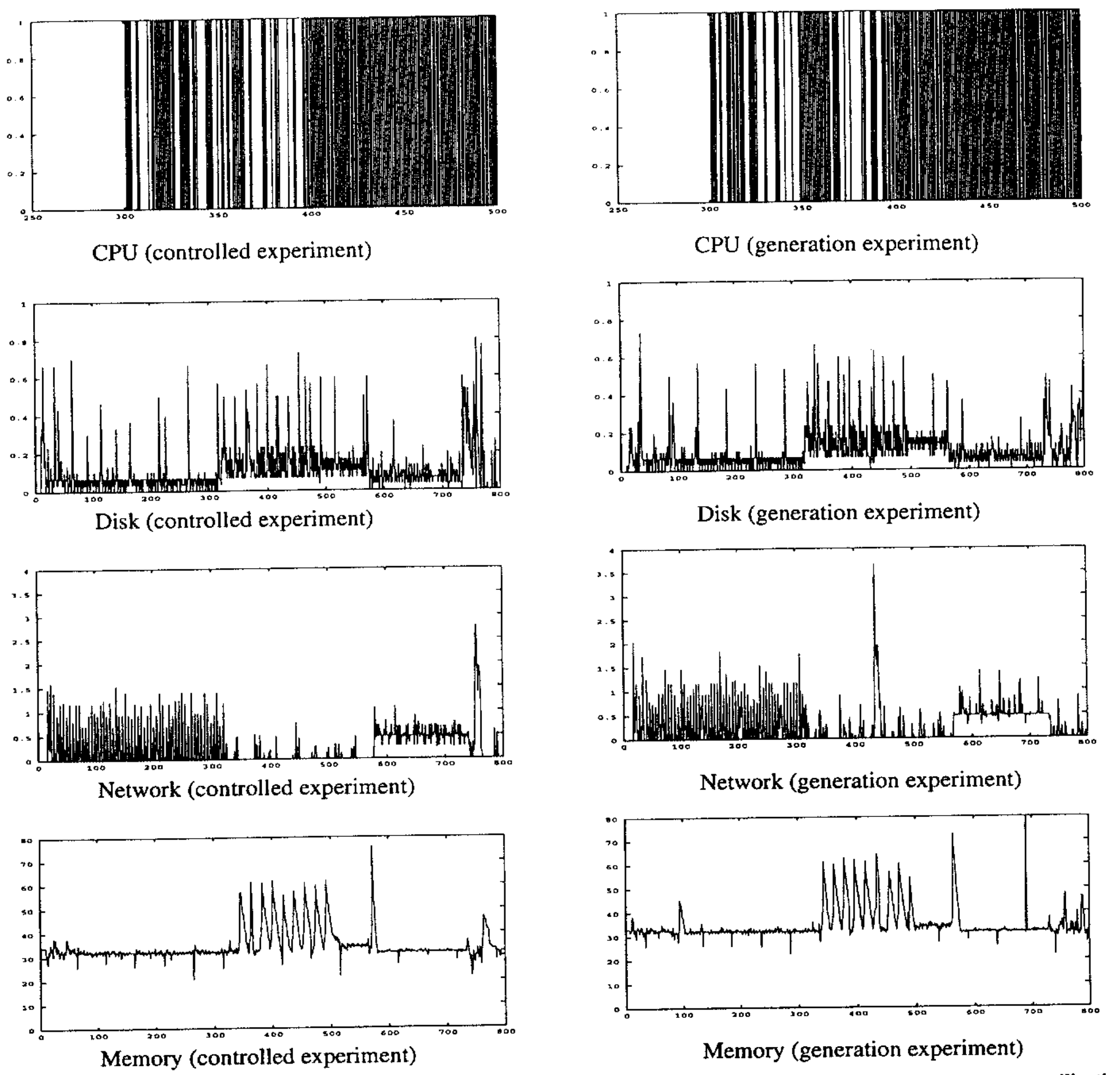

Figure 4. Comparison of true and generated resource-utilization patterns (Experiment 17). This figure shows the true utilization patterns of various resources to the left; and the generated ones, to the right. For CPU, the *Y*-value is either 0 or 1, indicating whether CPU was busy; the *X*-axis represents time (1 unit = 20 ms). For the remaining resources, sampled lots are shown, with 100 samples per minute. The *X*-axes in these cases show time (1 unit = 1/100 min). The *Y*-axes show: for disk, transfers per tick; for network, packets per tick; and, for memory, the number of free pages. The background job is the Perfect Club Benchmark TRFD, and the foreground job is the Unix sort command applied to a small file by a single field with limited memory space allocated.

can be expressed as a sum of service time and waiting time:

$$C(\hat{L}_{b,f,s}, f) = C(\hat{L}_{\phi,f,s}, f) + W(\hat{L}_{b,f,s}, f). \qquad (2)$$

Under round-robin scheduling policies at the process level, the waiting time of a process grows monotonically with its service time and the current load on the site's resources. Introducing an unknown site-specific and workload-specific function $G(\hat{L}_{b,f,s})$, we can rewrite (2) as:

$$C(\hat{L}_{b,f,s}, f) = C(\hat{L}_{\phi,f,s}, f)[1 + G(\hat{L}_{b,f,s})], \qquad (3)$$

where $G(\hat{L}_{b,f,s})$ increases monotonically with load for given f. Note that $G(\hat{L}_{b,f,s})$ can be considered as an average parameter that depends on the class of f but not on any attributes of a particular job. It will be learned implicitly in the weights of the neural network modelling the workload function.

Under the model of completion times described above, the stretch factor $C(\hat{L}_{b,f,s},f)/C(\hat{L}_{\phi,f,s},f)$ depends only upon load $\hat{L}_{b,f,s}$ and site s and on the class of jobs f, but remains constant for jobs in the same class. 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

$$C(\hat{L}_{\phi,f,s}f) = K(s,f)C(\hat{L}_{\phi,f,s_{\text{ref}}},f), \tag{4}$$

where $K(s,f)$ is the speed-up of site s for the class of jobs under consideration with respect to reference site s_{ref}. Similar to $G(\hat{L}_{b,f,s})$, $K(s,f)$ should be considered as an average parameter that may depend on the class of f but not on any attributes of a particular job. Again, we do not need to determine $K(s,f)$ explicitly as it will be learned implicitly in the weights of the neural network modelling the workload function. We can rewrite (1) as

$$\begin{aligned} F_s(\hat{L}_{b,f,s_1},f) - F_s(\hat{L}_{b,f,s_2},f) \\ = K(s_1,f)[1+G(\hat{L}_{b,f,s_1})] - K(s_2,f)[1+G(\hat{L}_{b,f,s_2}]. \end{aligned} \tag{5}$$

The right-hand side may depend on the class of job f but is independent of any attributes of the specific job f being scheduled. Therefore, the objective of learning a task-independent load-index function satisfying (1) is achievable, provided (i) waiting time grows linearly with service time; and (ii) different sites belong to 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.

To achieve our goal of obtaining accurate comparison in most situations, we would like to develop load-index functions F_s which will cause the left and right-hand side of (1) to have the same sign. That is, F_s will 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 tuned functions. Suppose that the set of raw training patterns is $\mathbf{R} = \{\tau(b,f,s_1,s_2) | b \in \mathbf{B}; f \in \mathbf{F}; s_1,s_2 \in \mathbf{S}; s_1 \neq s_2\}$. Each raw training pattern $\tau(b,f,s_1,s_2)$ is a 5-tuple

$$\{\hat{L}_{b,f,s_1}, C(\hat{L}_{b,f,s_1},f), L_{b,f,s_2}, C(\hat{L}_{b,f,s_2},f), C(\hat{L}\phi, f, s_{\text{ref}}, f)\}.$$

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

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

5.2. *Comparator network: architecture and learning algorithms*

Little is known about the problem of learning to compare functions. One exception is the work of Tesauro (Tesauro and Sejnowski 1989), 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 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 5 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 computed; based on these outputs, the desired outputs for each of the index functions is computed as follows:

$$\begin{aligned} O_L^D &= O_L - 2\eta(O_L - O_R - O^D); \\ O_R^D &= O_R + 2\eta(O_L - O_R - O^D). \end{aligned} \tag{7}$$

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 Fig. 5; 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. The filters are used to remove transients that prevent us from predicting the trend of workload in the near future. Since workload can be nonstationary, and it is

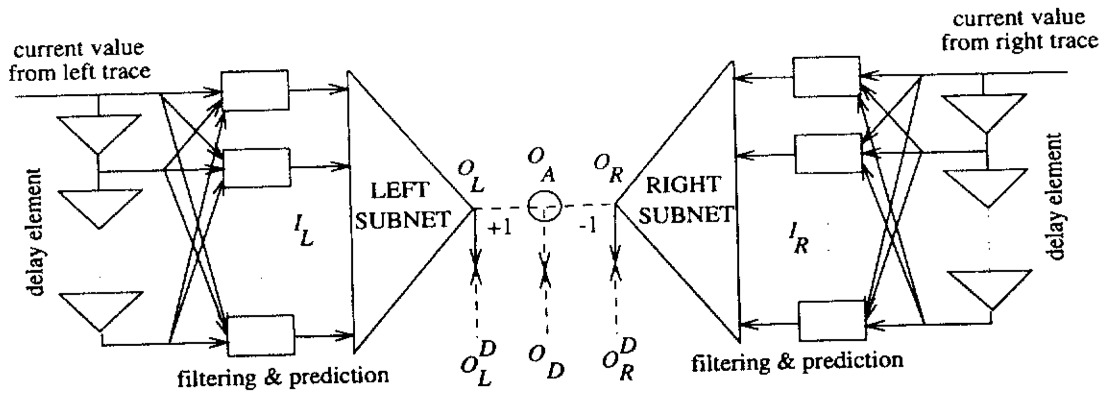

Figure 5. An episode in training a comparator network.

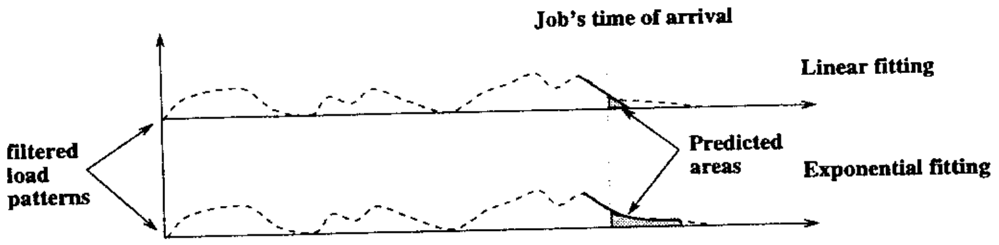

Figure 6. Trend extraction via curve fitting.

hard to decide the exact cut-off frequency for the filter, we use five different filters, with cut-off frequencies at, respectively, 1%, 5%, 10%, 25% and 50% of maximum frequency. This yields five 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. (This is another of those class-dependent parameters that are safe to use because they do not depend on any attribute of the job being scheduled.) Again, as it is hard to know *a priori* the best extrapolation technique, we use two different techniques: one using linear fitting and the other using exponential fitting. The area under each extrapolated curve (see Fig. 6) is used as an input to the load–index function. Since there are four different resources, five filters, and two 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 (Fig. 5).

Note that the difficult problem of using a neural network to predict relative speed-up is reduced to that of picking the appropriate combination of cut-off frequency for the filter and the extrapolation method.

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 of a link going from the ith to the jth unit of the neural 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 (Goddard *et al.* 1989).

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 $g(x) = 1/(1 + e^{-x})$.

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-proagation 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, $O^A = O_L - O_R$. Given our objectives stated in (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 (1). That is,

$$O^D = O_L^D - O_R^D = \frac{C(\hat{L}_{b,f,s_1}, f) - C(\hat{L}_{b,f,s_2}, f)}{C(\hat{L}_{\phi,f,s_{\text{ref}}}, f)}, \quad (8)$$

The value of the objective function shown in (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,

$$E_{\text{LMS}} = \sum_{\tau} (O^A - O^D)^2. \quad (9)$$

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:

$$\Delta O_L = -\eta \frac{\partial E_{\text{LMS}}}{\partial O_L}, \qquad \Delta O_R = -\eta \frac{\partial E_{\text{LMS}}}{\partial O_R}, \quad (10)$$

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 (7).

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.

5.3. *Empirical results*

This section presents our results on learning load indices for a system containing four sites. Each training pattern presented to the computer network contains 40 extrapolated values of areas under extrapolated resource-utilization patterns and two additional fields, $C(\hat{L}_{b,f,s}, f)$ and $C(\hat{L}_{\phi,f,s_{\text{ref}}}, f)$.

The training algorithm described in the last subsection was applied to $40 \times 40 \times 1$ 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% of the training patterns to the tests and the remaining 90% to the training set. The networks were trained using the training set, and their accuracy measured on the test set. (Partitioning data between training set and test set is the usual way to prevent biasing of the index-functions to the training set.) Training and testing were applied alternately; during each epoch of training, the networks were trained on 1000 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 (Fig. 5) could not be reversed during Stage 2.

If we were solely interested in the four-processor system that we measured, Stage 1 would have been unnecessary. However, we can simulate larger systems by 'cloning' sites as described in §4.2. Note that comparisons between clones are tantamount to comparisons between different loading situations for the same job at the same site.

While pairwise comparisons of loading conditions may form the basis of a comparator-network's operation, its eventual application involves comparison of multiple sites. In this context, the load indices output by comparator networks at different sites are compared in order to determine the least-loaded site. Since the objective of load balancing is to maximize speed-up over local execution, we can assess the quality of the new load-index functions by the speed-up attained if each incoming job were scheduled at the least-loaded site.

In Fig. 7, we show in the top four boxes the load indices (due to the background loading pattern only) of the various sites of our four-processor testbed. Test jobs were introduced on top of these background workloads 2252 time units into the experiment, each time at a different site. The completion times of all the jobs and their check-points were recorded. Two policies were compared: (i) *opt*, which always places the incoming

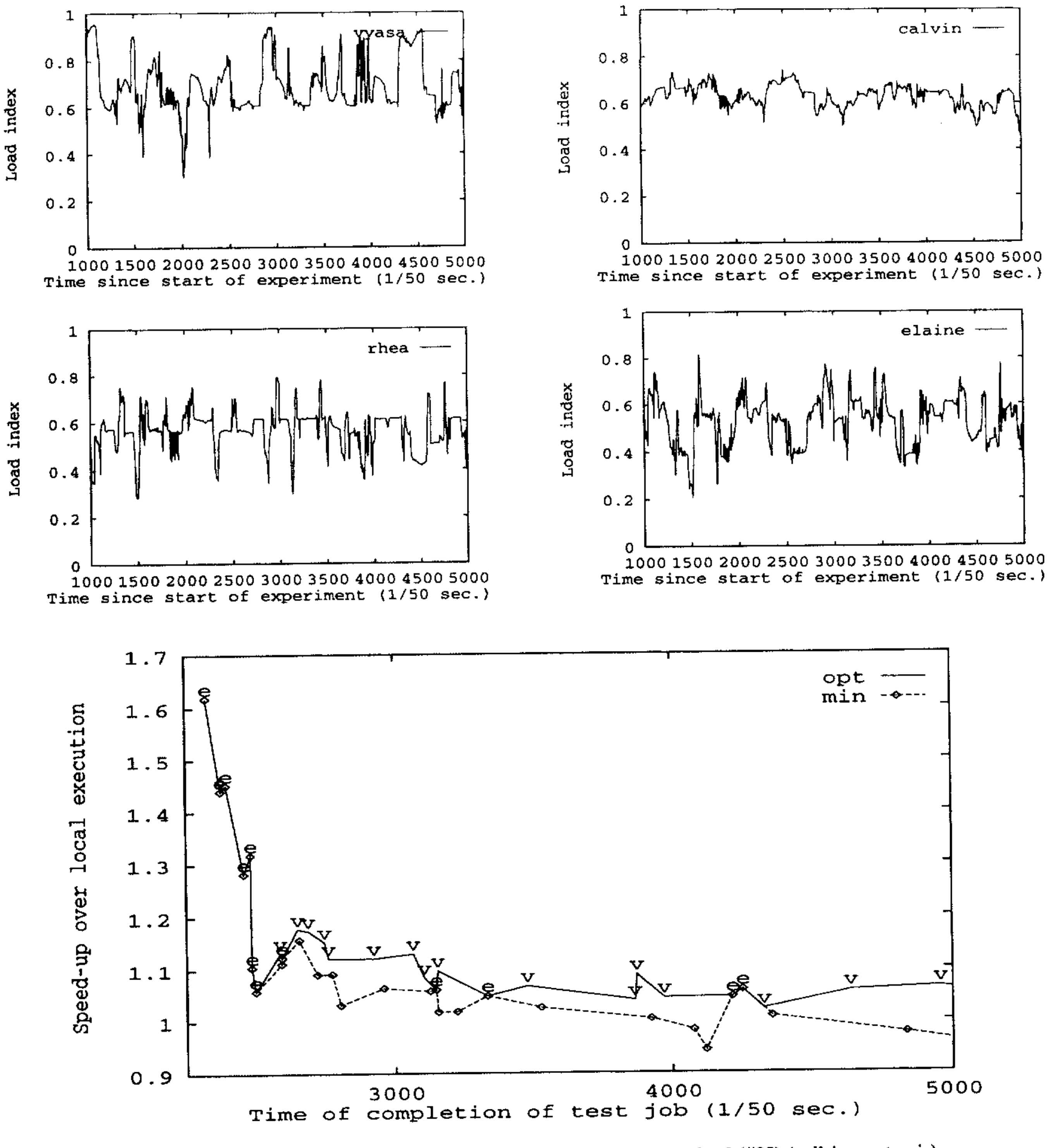

Figure 7. Average speed-up of all test-jobs under the worst-behaved load (#23) (policies: *opt*, *min*).

job at the site with the optimum (least) completion time; and (ii) *min*, which always schedules an incoming task 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. In the bottom part of Fig. 7, these speed-up values are plotted against the time at which a job completes. This plot shows that while the site having the least load index behaves as well as the optimal site for short jobs, its performance drops below optimal for long jobs. The top curve in this plot indicates the optimal site ('e' for elaine, and 'v' for 'vyasa') and the corresponding optimum speed-up for each test-job. 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 sub-optimal only when a site other than elaine is the optimal destination. Since the load pattern used in this experiment is one of the four we designed (artificially) to have poor locality, we can conclude that the effects of poor locality in background workloads (which become worse for longer jobs) cannot be counteracted by a smart load index. Instead, pre-emptive process migration should be used for long jobs, so that the near-optimal short-term performance of the load index can be exploited during rescheduling.

We postpone to § 8 the evaluation of the *min* policy. Also, *min* does not consider the effects of job-migration overhead, delay between load–index calculation, and the time the load index is used. These effects are considered in § 7.

6. Automated tuning of load-balancing policies

6.1. *The policy-learning problem*

Figure 1 shows the sender-side and receiver-side rules for the load-balancing policies considered in this paper. There is one set of rules per site. The rules are parametrized; the number of parameters depends upon the number of sites in the distributed system. Let N be the number of sites. At each site, its SSRs have three parameters: θ_1, δ and *Reference*, and its RSR has one parameter: θ_2. Altogether, there are $4N$ parameters for the whole system. In this paper, we use the term heuristic method (HM) (Ieumwananonthachai and Wah 1996, Wah 1992, Wah 1995) to denote a set of system-wide parameters.

Tuning of load-balancing policies is done offline, which means that the decisions made by a policy are evaluated not by actually sending test jobs to an operational load-balancing system, but by simulating the application of that policy on loading conditions and test jobs. Each loading situation is represented by the load indices of different sites at the time of the test job's arrival. Since the measurements were conducted by DWG during the data-collection phase, the completion time of each incoming test job is known ahead of time for all possible destinations. Further, by applying the load-index functions described in the last section, we can obtain, for each load pattern, a complete system-wide trace of load indices up to the job's time of arrival. Given the load indices, we can determine the destination of each incoming job by simulating the application of the load-balancing policy using the SSRs and RSRs. Since measured completion times are already known for both local execution and remote execution, the improvement in completion time due to migration can be determined immediately.

We organize the data collected ahead of time into test-cases, each carrying information about one incoming job, its time of arrival, the load indices of different sites at the time of arrival, and information about the measured completion time of that job at each of the d sites. We use *test* in this paper to denote the evaluation of a selected HM on one test-case. Since our experiments are performed offline, we assume that each test takes unit time.

The test-cases of an application constitute the test database, which represents a sample from the space of problem instances. (For the load-balancing domain, each instance describes the incoming job and the background loading pattern.) Associated with each HM is a population of performance values, one value per test-case. The space of HMs can be viewed as a population of populations. Our goal of optimizing average speed-up over local execution can be reformulated as one of choosing the HM whose population has the highest mean.

Given the large number of parameters, many of them real-valued, the space of possible HMs is very large. We must search this space intelligently and by performing only a limited number of tests. If we had unlimited time, we could test every HM on every test-case, and then choose the one with the highest mean. In limited time, however, only a limited number of HMs may be tested, and only a few tests can be performed per HM. Therefore, the policy-learning system must possess bounded rationality (Simon 1982): it must schedule tests intelligently. This scheduling problem is the crux of policy learning; the theory for this problem was developed by Wah *et al.* (1995), whose TEACHER (TEchniques for Automated Creation of HEuRistics) system (Ieumwananonthachai *et al.* 1992, Ieumwananonthachai and Wah 1995, 1996, Ieumwananonthachai 1996, Wah 1992, Wah *et al.* 1995) provides a domain-independent approach to rational scheduling of limited learning time between generation and testing.

6.2. *Architecture and operation of TEACHER*

Figure 8 shows the architecture of TEACHER (Ieumwananonthachai and Wah 1995, 1996, Wah 1992) as applied to learning load-balancing policies. Its key component is the scheduler, which decides whether to generate a new HM or continue testing the ones already in the pool. The advantage of generation is that if a large number of HMs are tested before selecting the HM with the largest sample mean, then the likelihood of missing the population with the largest mean is

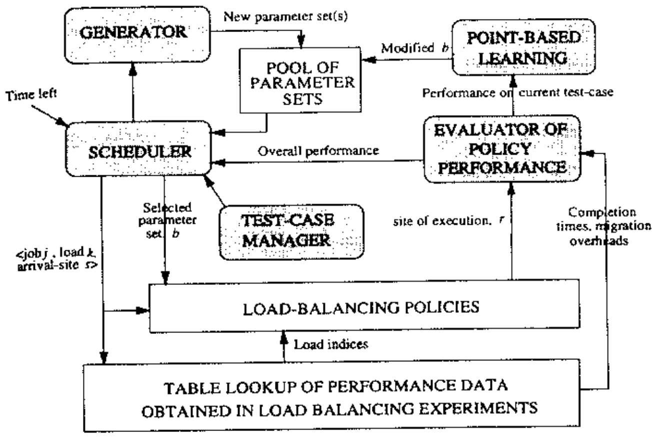
GENERATOR
New parameter set(s)
POOL OF PARAMETER SETS
Modified b
POINT-BASED LEARNING
Performance on current test-case
Time left
SCHEDULER
Overall performance
EVALUATOR OF POLICY PERFORMANCE
Selected parameter set, b
TEST-CASE MANAGER
site of execution, r
Completion times, migration overheads
<job j, load k, arrival-site s>
LOAD-BALANCING POLICIES
Load indices
TABLE LOOKUP OF PERFORMANCE DATA OBTAINED IN LOAD BALANCING EXPERIMENTS

Figure 8. Automated learning of load-balancing policies by TEACHER.

reduced. On the other hand, when tests are spread across several different HMs, each one can be tested on only a few problem instances; even though the estimated quality (sample mean) of an HM may be high, our confidence in that estimate will be low. The advantage of continuing to test existing HMs is that our confidence in the estimate improves with the number of tests. TEACHER divides up its total learning time into generations similar to those used in genetics-based heuristic-learning systems. At the beginning of each generation, it calls the generator to generate new HMs based on the past performance (if any) of the existing HMs. Then, until the end of the generation, it continues to test the new HMs (as well as some old HMs that have been retained from previous generations). At the end of the final generation, the HM with the highest sample mean is chosen.

The inputs of a load-balancing policy are described by a test-case (Fig. 8), which includes information about the job (j) that needs to be scheduled, the background-load pattern (k) being simulated, and the site (s) at which the job arrives. The different test-cases to be used are managed by a Test-case Manager.

Before commencing the experiment, the learning system sets all the policy parameters using the corresponding values from a selected parameter set, b. It then uses the information contained in the test-case to compute the load index of each site at the time of job j's arrival. Thus, given the values of policy parameters, as well as the values of primitive and abstract decision metrics, one can determine the policy's decision for the chosen test job. This decision involves the choice of a remote destination (r) where the incoming job j should be executed.

Given the information contained in the test-case, and knowing the decision (r) of the policy, the performance of the policy parametrized by b can be computed as follows. First, the system consults the completion time of job j at site r under load-pattern k. Next, if the chosen destination r is not the same as the site of arrival s, then an overhead of migration (to be discussed in the next section) is added to the completion time. The total represents the completion time of a job under remote execution, including overhead. From the data collected in the first phase, we also know the completion time under local execution. Using these numbers, the speed-up achieved by parameter set b for the given test case can be calculated.

The Evaluator of Policy Performance (Fig. 8) computes the overall performance of each parameter set on a population of test cases. Using information about the recent performance of different parameter sets, and knowing how much time is available for learning, the Scheduler determines whether to generate new parameter sets or to continue testing from the current pool.

The initial HMs used in TEACHER can either be generated randomly or provided by the designer. TEACHER uses three genetic operators to generate new HMs from existing ones: (i) *Mutation*: the Reference value of a randomly chosen site is changed from 0 to *MinLoad*, or vice versa; (ii) *Crossover-1*: two new HMs generated by interchanging the substrings containing the parameters of two HMs at a randomly chosen site; (iii) *Crossover-half*: similar to Crossover-1, except that the exchange involves $N/2$ sites.

Following the customary practice in genetic search, we make the more destructive operators less likely, and, the less destructive ones, more likely. In our experiments, we used the probabilities of 0·1, 0·85 and 0·05, respectively, for the three operators described above.

Our learning system not only develops new parameter sets by the population-based learning process described above, but also refines existing parameter sets after each learning trial. It does so using point-based learning, a parameter-modification procedure that reduces the likelihood of decisions causing low speed-ups, by suitably altering policy thresholds. Since we have only limited prior domain knowledge, refinement by point-based learning can only be applied to some of the parameters.

In applying point-based learning to learn load-balancing policies, our learning system uses the information on completion times and overheads for all possible destinations. Since this information is known ahead of time, our system knows whether remote execution was optimal and, if so, then at which site. It can thereby infer what destination should have been included in the set of possible destinations. If the optimal destination was not included in that set, the point-based learning algorithm raises the threshold δ by a small amount. Likewise, if a destination causing sub-optimal speed-up is selected, then the threshold δ is decremented

Adjustment of δ:

- If the optimal destination was not in the set of possible destinations, then let ε_1 be the amount by which δ should be raised at the site of arrival (s) in order to include the optimal site.
- If some sub-optimal destination having load index higher than that of the optimal site was included in the set of destinations, then let ε_2 be the amount by which δ should be lowered at the site of arrival (s) in order to exclude the sub-optimal site.
- If the optimal destination is not in the set of possible destinations, then set $\delta(s) \leftarrow \delta(s) + \eta \cdot \varepsilon_1$, else set $\delta(s) \leftarrow \delta(s) - \eta \cdot \varepsilon_2$.

Adjustment of $\theta_1(s)$:

- WHEN (AND send-optimal (NOT send-occurred)) $\theta_1(s) \leftarrow \theta_1(s) + \eta(\text{Load}(s) - \text{Reference}(s) - \theta_1(s))$.
- WHEN (AND send-occurred (NOT send-optimal)) $\theta_1(s) \leftarrow \theta_1(s) - \eta(\theta_1(s) - \text{Load}(s) + \text{Reference}(s))$.

Adjustment of $\theta_2(r)$ [r was the destination chosen by SSRs(s)]:

- WHEN (AND receive-occurred (NOT receive-optimal)) $\theta_2(r) \leftarrow \theta_2(r) + \eta(\text{Load}(r) - \theta_2(r))$.
- WHEN (AND receive-optimal (NOT receive-occurred)) $\theta_2(r) \leftarrow \theta_2(r) - \eta(\theta_2(r) - \text{Load}(r))$.

Figure 9. Rules for point-based learning.

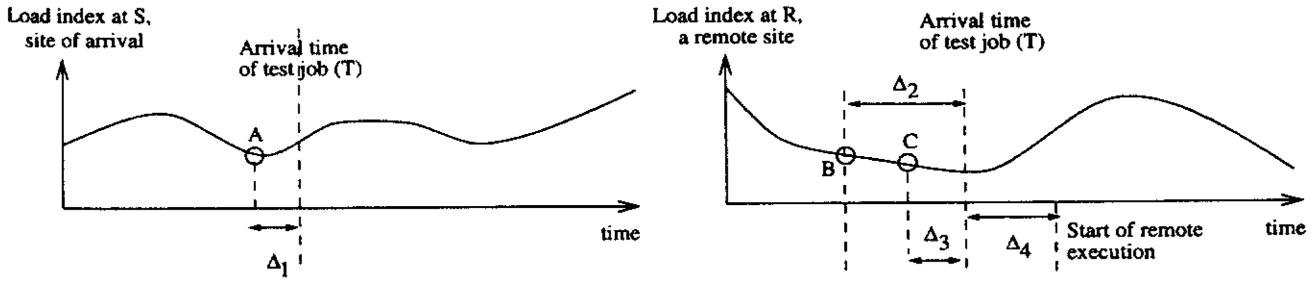
Load index at S, site of arrival
Arrival time of test job (T)
A
Δ_1
time
Load index at R, a remote site
Arrival time of test job (T)
Δ_2
C
B
Δ_3
Δ_4
Start of remote execution
time

Figure 10. Overheads and delays in load balancing. (A: time that site S received the load index of site R; C: time that site R received the load index of site S; B: time that load index at site R was computed.)

by a small amount provided such a decrement will not exclude the optimal destination as well.

When the destination chosen randomly by the first two SSRs is rejected, either by the RSRs or by the third SSR, then the behaviour of the point-based learning algorithm depends upon the performance of that destination. If the chosen destination would have given better speed-up than 1, then the threshold of the offending rule is adjusted. If the third SSR caused the rejection, then its θ_1 is lowered; and if the RSR caused the rejection, then θ_2 is raised.

When the destination selected by the SSRs and accepted by the RSRs yields sub-optimal speed-up, then the θ_1 threshold for the offending SSRs is raised, and the θ_2 for the offending RSRs is lowered.

Thus, all the thresholds governing the behaviour of the load-balancing policy can be adjusted using the performance information for just the current test-case. The rules of point-based learning are summarized in Fig. 9.

6.2. *Data preparation*

The sole purpose of the data-preparation phase of policy learning is to replace an actual problem solver by a table-look-up routine, as shown in Fig. 8. Given a job and a background load pattern, as well as the job's site and time of arrival, the look-up routine must supply the values of load indices at each of the sites, the completion time of that job at all possible destinations, as well as the respective migration overheads of each destination.

Figure 10 shows, left, a trace of the load–index function at the arrival site of an incoming job and, right, the corresponding trace at a remote site where the job is to be executed.The letters ‘A’ and ‘C’ indicate the most recent local load–index values; the letter ‘B’ indicates the most recent load–index value of the remote site available at the site of arrival. The delays Δ_1 and Δ_3 depend upon the interval of computation of load–index function. Such delays affect the currency of both the local and the remote load indices. The delay Δ_2 includes both the delay in computing the load index at the remote site and the delay in propagating that value to the site of arrival. Finally, the delay Δ_4 represents the overhead of migration.

It is not enough to add Δ_4 to the completion time of every test job, for in our (offline) data-collection phase, we did not run the test job at the remote site starting at $T + \Delta_4$; rather, we always started the test job at time T into the experiment. Thus, the job would have run under a slightly different load with an online experiment; the difference between online (actual) and offline (simulated) cases is proportional to Δ_4. If Δ_4 is large, the load could have been very different; if small, then less so. The effect of migration overhead is simulated by adding Δ_4 to the age of the remote load index.

Consider a typical load–index function. If the feed-forward comparator network computing it had 40 hidden units, then the computation of this function requires 1680 ($40 \times 40 + 40 + 40$) floating-point multiplication operations. Computing this function at every clock interrupt would incur too much overhead. Therefore, such a function can be computed only periodically. Let τ be the period between successive computations; then, the delays Δ_1 and Δ_3 are random variables distributed uniformly over the interval $[0, \tau]$.

One final factor that needs to be considered is Δ_2, the delay including the propagation time of load values to remote sites. Assuming that load values are sent over the network each time they are computed, the propagation delay can be modelled by half the round-trip time of a short packet. Round-trip times can be measured offline for a network. Propagation delays of load indices affect the values of only the remote load indices at any decision point.

Since the comparator network at each site gives us a complete trace of the load index at that site up to the decision point, the effects of delays due to computation and communication of load indices can be simulated by using older values of load indices, rather than their values at time T.

7. Empirical results

We collected workload and completion-time data on our testbed of four computers and performed offline learning on a Sun SparcStation 10, Model 20. The total down time was around 10 days; most of it was spent on manually recording the 24 background loads and measuring completion times of 10 test jobs under these loads at each of the four sites. We expect down times to be reduced to around 3 days if the process is fully automated. Moreover, down times apply only to new computers of different configurations added in the future; computers of the same configuration can be considered as clones, and their workload index and load balancing strategies do not have to be relearned. On the other hand, the training of comparator neural networks is fully automated; it requires no down time because it is performed offline (around 15 hours of CPU time).

Using the data for our four-processopr testbed, we created a total of nine clones (three each for the two diskless clients, two for the diskful client, and one for the file server). This gave us a total of 13 sites. Policies for this scaled-up system were learned using TEACHER under a variety of assumptions about overheads and delays.

As indicated in the previous section, three types of delays contribute to the age of a load index. After measuring packet turn-around times, we found it adequate to assume that load information from remote sites can be received within 20 ms ($P = 20$) of being broadcast. We studied four different migration overheads (M): (i) small, 100 ms; (ii) medium, 500 ms; (iii) moderately large, 2000 ms; and (iv) large, 5000 ms. In addition, we considered three different intervals (I) of load-index computation: (i) small, 100 ms; (ii) medium, 500 ms; and (iii) large, 1000 ms. Table 3 shows the average speed-up for each combination of M and I. In generating data in Table 3, we have added a random delay uniformly distributed between 0 and I to the age of each load index. Only for the remote load indices do we add the factor M and P.

Table 3. Performance of policies learned under different assumptions for our four-processor testbed and nine clones. M: process migration overhead; *I*: time interval between load–index computation)

M (ms)	I (ms)	Average speed-up	Std. dev. speed-up	M (ms)	I (ms)	Average speed-up	Std. dev. speed-up
100	100	1·737 43	0·226 114	500	100	1·501 56	0·179 138
	500	1·499 16	0·219 608		500	1·530 47	0·152 211
	1000	1·509 08	0·203 219		1000	1·600 4	0·173 11
2000	100	1·239 7	0·069 449 8	5000	100	1·251 81	0·128 558
	500	1·159 47	0·112 387		500	1·191 49	0·140 72
	1000	1·178 21	0·124 843		1000	1·310 23	0·134 078

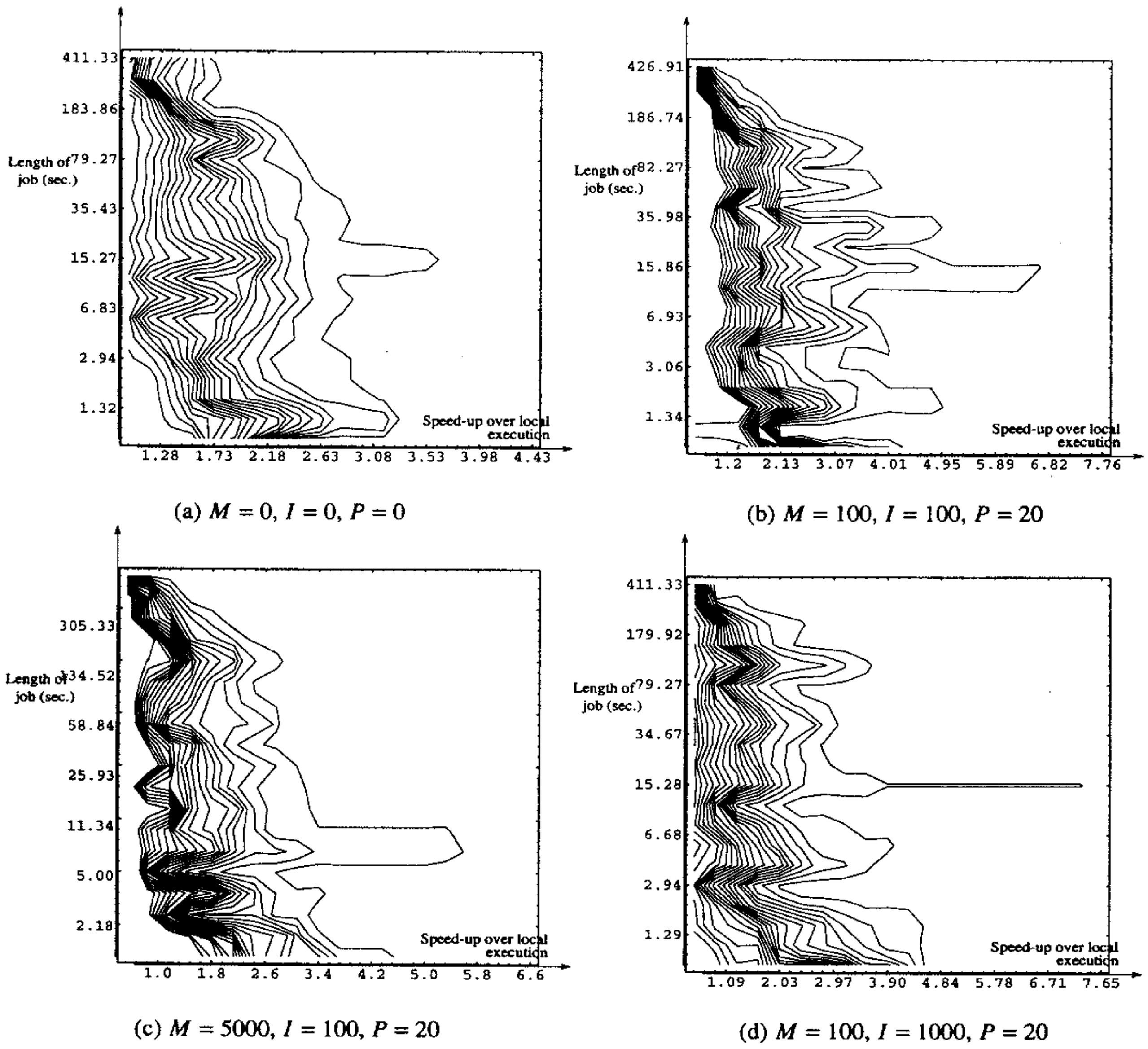

Figure 11. Four contour plots of cumulative probability of speed-up for our four-processor testbed and nine clones. (M: process migration overhead; I: time interval between load–index computation; P: delay in propagating load index from one site to another.)

TEACHER was allowed 30 minutes learning time for each experiment on a SparcStation 10, Model 20. Twelve experiments were performed, one for each parameter set described in Table 3. The best policy found by TEACHER after 30 minutes was fully evaluated on all the test-cases. The average speed-up of that policy and the corresponding standard deviation are shown in Table 3.

The best policy is found when both M and I are small. When M is large, the workload at the destination site may be very different from the time when the load balancing decision was made at the source site. As a result, the load balancing decision made earlier may not be the best. For a similar reason, when I is large, the load balancing decision may be made based on possibly out-of-date workload information, and the site chosen for migration may not be the best. Our results in Table 3 show that performance appears to decrease with increasing M, but no such effect is visible for I.

In order to study the relationship between the length of a job and the speed-up achieved for that job, we evaluated each policy on all 10 test-jobs under all 24 load patterns. We then created data-pairs $\langle l_i, e_i \rangle$, where the first item of each pair is the length of the job, and the second, the corresponding speed-up over local execution. Figure 11 shows four contour plots, depicting the performance of the *min* policy discussed in §5.4 and other policies with various overheads.

Recall the *min* policy always picks the site having the minimum of load indices predicted by comparator networks at different sites. In this policy, we assume that $\Delta_1, \Delta_2, \Delta_3$ and Δ_4 are all zeros. Figure 11(a) shows a contour plot of the probability of achieving certain speed-ups over local execution (X-axis, ranging from 0·93 to 4·92) for a checkpoint at a specific time in seconds from the start of a job (Y-axis, ranging from 0·66 to 439·94 s). Since there are many more short jobs than there are long ones, we use logarithmic scaling for the Y-axis. Nineteen contours, each connecting X-Y points having equal cumulative probability of speed-up, divide up the space into 20 regions of 5% probability. While speed-ups higher than 1·5 occur frequently for short jobs, they almost never occur for jobs that take more than 3 min to complete.

Figure 11(b) shows a contour plot of the best policy learned when $M = 100$, $I = 100$ and $P = 290$. This figure shows that the best speed-up values are achieved for jobs that are neither too long nor too short. The quality of our indices drops with the length of the job; therefore, long jobs cannot be speeded up by one-time placement; they need pre-emptive process migration. The speed-up of short jobs is adversely affected by the magnitude of migration overheads relative to the gains in completion time achievable using remote execution.

Figure 11(c) shows the behaviour of policies learned under high migration overheads but a small interval of load-index computation. Relative to Figure 11(b), high speed-up values are less probable when migration overhead is significant. Figure 11(d) shows the case when migration overheads are small but the interval of load index computation is long. High speed-up values, close to 3·0, for medium-length jobs are not as likely as in Figure 11(b), but they are much more likely than in the case of high migration overheads.

We note that the best policy found by pure point-based learning (starting with one randomly generated parameter set) had a speed-up of only 1·37 for the case $M = 100$, $I = 100$. Considering that our system finds a policy with a speed-up of 1·73, we can appreciate the utility of integrating learning. It is important to note that these speed-ups were obtained not by parallel processing but by job migration. Moreover, the speed-ups are for each independent job rather than the overall speed-up for a batch of (possibly mutually dependent) jobs.

8. Conclusions

We have demonstrated in this paper automated learning of meaningful load index functions and load-balancing policies from real workload data. Using data collected on a real distributed system, we trained neural networks to effectively compare alternative loading conditions for an incoming job. We have applied TEACHER, a heuristics learning system, to tune the parameters of our load-balancing policies at each site. TEACHER generates parameter sets of load-balancing policies using genetic operators, and modifies them systematically using point-based learning. The key functions supported in TEACHER are selective breeding of well-performing parameter sets at the end of each generation, and rational allocation of tests among parameter sets during each generation.

We collected workload and completion-time data on a network of four configurationally-distinct Sun 3 workstations, and performed offline learning on a Sun SparcStation 10, Model 20. The total down-time was around 10 days; most of it was spent on manually recording the 24 background loads and measuring completion times of 10 test jobs under these loads at each of the four sites. We expect down-times to be reduced to around 3 days if the process is fully automated. Moreover, down-times apply only to new computers of different configurations added in the future; computers of the same configuration can be considered as clones, and their workload index and load balancing strategies do not have to be relearned. On the other hand, the training of comparator neural networks is fully automated; it requires no down-time because it is performed offline (around 15 hours of CPU time).

Although the number of sites in our testbed is small (due to the fact that the system had to be offline frequently throughout the four years of this research), the method we have developed is not limited to such a small scale. In this paper, we have demonstrated that by cloning the sites—using workload data from one site at different times to create more sites—we can learn effective load-balancing policies for a larger (13-site) system. Of course, cloning is limited to the case where new sites added to a system have hardware identical to the already-existing sites.

Our comparator network architecture discovers new load indices that can be meaningfully compared across sites; these functions induce a ranking on loaded sites that is consistent with their true ranking in a statistically significant fashion. We further conclude that (i) our system discovers policies with significant speed-ups over local execution under a variety of assumptions about overheads and delays; (ii) the quality of policies learned is adversely affected by both large migration overheads and long intervals of load index computation; and (iii) migration overhead is the dominant factor affecting the quality of policies learned.

As important as the machine learning problems we solved in this paper are those we chose not to solve by making certain simplifying assumptions and modifying our experimental design to satisfy those assumptions. The problems we addressed include the development of a neural network configuration capable of learning to compare functions of time series (see Wah *et al.* 1994 for related work) and an integraton of population-based and point-based learning techniques. The problems we did not solve—temporal credit assignment and exploration–convergence dilemma—continue to defy general solutions although good solutions are now known for Markovian decision problems (not applicable to load balancing).

By carrying out an in-depth investigation into the use of machine learning techniques for the development of self-tuning load-balancing software, we have conclusively demonstrated that not only can we develop good load-balancing policies but also that we can do so systematically, automatically and efficiently.

Acknowledgments

Research supported by National Aeronautics and Space Administration Grant NCC 2-481 and NAG 1-613, National Science Foundation Grants MIP 92-18715 and MIP 96-32316, and a grant from Sumitomo Electric Industries, Yokohama, Japan.

Preliminary results related to this paper were presented at the 1993 International Conference on Parallel Processing (Mehra and Wah 1993).

References

Barto, A. G., Sutton, R. S., and Watkins, C. J. C. H., 1990, Learning and Sequential Decision Making. In *Learning and Computational Neuroscience: Foundations of Adaptive Networks*, edited by M. Gabriel and J. Moore (Cambridge, Massachusetts, U.S.A.), pp. 539–602.

Baumgartner, K., and Wah, B. W., 1989, GAMMON: a load balancing strategy for a local computer system with a multiaccess network. *IEEE Transactions on Computers*, **38**(8), 1098–1109.

Berry, M., *et al.*, 1989, The Perfect Club benchmarks: effective performance evaluation of supercomputers, *International Journal of Supercomputing Applications*, **3**(3), 5–40.

Devarakonda, M., and Iyer, R. K., 1989, Predictability of process usage: a measurement-based study of UNIX. *Transactions on Software Engineering*, **15**(12), 1579–1586.

Ferrari, D., Serazzi, G., and Zeigner, A., 1983, *Measurement and Tuning of Computer Systems* (Englewood Cliffs, NJ: Prentice-Hall).

Ferrari, D., 1984, On the foundations of artificial workload design. *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pp. 8–14; 1986, A study of load indices for load balancing schemes. In *Workload Characterization of Computer Systems and Computer Networks*, edited by G. Serazzi (Amsterdam, Netherlands: Elsevier Science).

Gloriosi, R. M., and Colon-Osorio, F., 1980, Engineering intelligent systems. *Engineering Intelligent Systems* (Burlington, Massachusetts, U.S.A.: Digital Press).

Goddard, N. H., Lynne, K. J., Mintz, T., and Bukys, L., 1989, Rochester Connectionist Simulator Technical Report, University of Rochester.

Ieumwananonthachai, A., 1996, Automated design of knowledge-lean heuristics: learning, resource scheduling, and generalization. Ph.D. Thesis, Department of Electrical and Computer Engineering, University of Illinois, Urbana, Illinois, U.S.A.

Ieumwananonthachai, A., Aizawa, A., Schwartz, S. R., Wah, B. W., and Yan, J. C., 1992, Intelligent process mapping through systematic improvement of heuristics. *Journal of Parallel and Distributed Computing*, **15**, 118–142.

Ieumwananonthachai, A., and Wah, B. W., 1995, TEACHER—An automated system for learning knowledge-lean heuristics, Technical Report CRHC-95-08, Center for Reliable and High Performance Computing, Coordinated Science Laboratory, University of Illinois, Urbana, Illinois, U.S.A.; 1996, Statistical generalization of performance-related heuristics for knowledge-lean applications, *International Journal of Artificial Intellience Tools*, **5**, 61–79.

Kumar, A., Singhal, M., and Liu, M. T., 1987, A model for distributed decision making: an expert system for load balancing in distributed systems. *Proceedings of COMPSAC*, pp. 507–513.

Litzkow, M. K., Livny, M., and Mutka, M. W., 1988, Condor—a hunter of idle workstations,. *Proceedings of the 8th International Conference Distributed Computer Systems*, pp. 104–111.

Mehra, P., 1992, Automated learning of load balancing strategies for a distributed computer system. Ph.D. Thesis, Department of Computer Science, University of Illinois, Urbana, Illinois, U.S.A.

Mehra, P., and Wah, B. W., Architecture for strategy learning. *Computer Architectures for Artificial Intelligence Applications*, edited by B. Wah and C. Ramamoorthy (New York: Wiley), pp. 395–468; 1993, Automated learning of workload measures for load balancing on a distributed system. *Proceedings of the International Conference on Parallel Processing*, pp. III-263–III-270, CRC Press; 1995a, *Load Balancing: An Automated Learning Approach* (Singapore: World Scientific Publishing); 1995b, Synthetic workload generation for load balancing experiments. *IEEE Parallel and Distributed Technology: Systems and Applications*, **3** (3), 4–19.

Mirchandaney, R., and Stankovic, J. A., 1986, Using stochastic learning automata for job scheduling in distributed processing systems. *Journal of Parallel and Distributed Computing*, 527–552.

Pasquale, J. C., 1988, Intelligent decentralized control in large distributed computer systems. Ph.D. Thesis, University of California.

Pulidas, S., Towsley, D., and Stankovic, J. A., 1988, Imbedding gradient estimators in load balancing algorithms. *Proceedings of the 8th International Conference Distributed Computer Systems*, pp. 482–490.

Sannier, II, A. V., and Goodman, E. D., 1988, Midgard: a genetic approach to adaptive load balancing for distributed systems. *Machine Learning*, 174–180.

Simon, H., 1982, *The Sciences of the Artificial* (Cambridge, Massachusetts, U.S.A.: MIT Press).

Stankovic, J. A., 1985, An application of Bayesian decision theory to decentralized control of job scheduling. *Transactions on Computers*, **34**, 117–130.

Sutton, R. S., 1988, Learning to predict by the methods of temporal differences. *Machine Learning*, **3**, 9–44.

Tesauro, G., and Sejnowski, T. J., 1989, A parallel network that learns to play backgammon. *Artificial Intelligence*, **39**, 357–390.

Wah, B. W., 1992, Population-based learning: a new method for learning from examples under resource constraints. *IEEE Transactions on Knowledge and Data Engineering*, **4**(5), 454–474.

Wah, B. W., Mehra, P., and Teng, C.-C., 1994, Comparator neural network for dynamic prediction. *Proceedings of the International Symposium in Neural Networks*, Tainan, Taiwan, pp. 571–580.

Wah, B. W., Ieumwananonthachi, A., Chu, L. C., and Aizawa, A., 1985, Genetics-based learning of new heuristics: rational scheduling of experiments and generalization. *IEEE Transactions of Knowledge and Data Engineering*, **7**, 763–785.

WAH, B. W., IEUMWANANONTHACHA, A., YAO, SHU, and YU, TING, 1995, Statistical generalization: theory and applications (Plenary Address). *Proceedings of the International Conference on Computer Design*, pp. 4–10.

ZHOU, S., 1987a, Performance studies of dynamic load balancing in distributed systems. Technical Report of UCB/CSD 87/376, Ph.D. Dissertation, Computer Science Division, University of California; 1987b, An experimental assessment of resource queue lengths as load indices. *Proceedings of the Winter USENIX Conference*, pp. 73–82.