CHAPTER I.

LOAD BALANCING AS A STRATEGY-LEARNING TASK

The ineluctable core of intelligence is always in that next thing which hasn't yet been programmed.

- Douglas Hofstadter

A. Load Balancing in Distributed Computer Systems

Workstations interconnected by local-area networks (LANs) are the most popular examples of distributed systems. Recent advances in RISC technology, as well as expected improvements due to superscalar architectures in the near future, are major factors behind their growing popularity. Modern workstation-based distributed computing environments feature microprocessors rated at 10 to 200 MIPS, interconnected by high-speed LANs rated at 10^7 to 10^{10} baud. The abstract model of distributed systems considered in this thesis (Figure 1) reflects this trend in distributed computing.

All of the sites in our model have private memory and processing capacity; some have secondary storage; and all share the communication resource. These resources are *architecturally homogeneous* [205], whereby they can service requests issued by programs running at any of the sites. Some of the resources, such as network and secondary storage, can be shared transparently; others, such as processing power and virtual memory, can be accessed by local processes only. We do allow for *configurational heterogeneity* [205], whereby different sites may have different processor speeds, memory space, or disk space.

Users at different sites initiate tasks (or jobs) in a distributed fashion. We restrict our attention to a simple model of tasks: our tasks are *independent* (i.e., there are no synchronization or precedence constraints) and can be processed at any of the several sites. A task under execution is called a process.

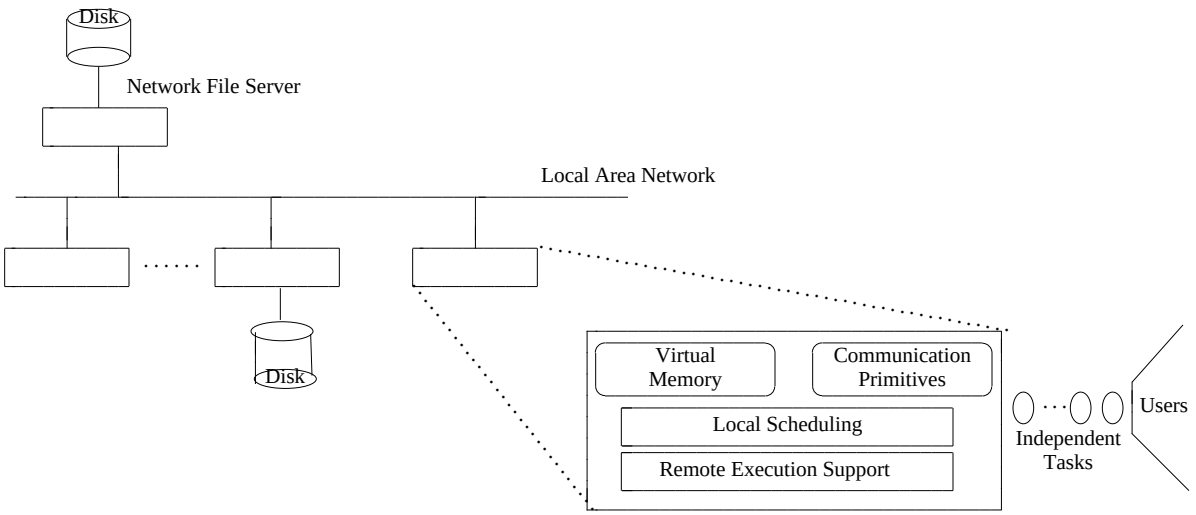


Figure 1. Our model of distributed systems.

Our model assumes code compatibility among sites, as well as the availability of an operating system responsible for local scheduling of processes and management of virtual memory. We further assume that user files are available at all sites, and the basic mechanisms for remote execution are provided by the operating system. At the user level, we assume that only independent tasks amenable to migration are considered for global scheduling.

Processes request and consume resources. The *workload* of a site consists of the combined demands on its resources from all of the local processes. The absolute and relative utilizations of various resources at each site, and of various sites across the network, are highly dynamic quantities. The dynamic nature of load causes frequent imbalances: certain resources local to a site may be overloaded even as similar resources at a remote site are underutilized or idle. With increases in the speeds of individual processors, and with growth in the scale of typical systems, there are concomittant increases in both the magnitude and the frequency of load imbalances. The desire to exploit such imbalances for improved performance has fueled much recent work on *dynamic load balancing* [18], which migrates tasks (and thereby, workload) from the heavily-loaded sites to the idle or lightly-loaded ones.

Just as distributed-systems hardware has benefited from technological advances in microprocessors and networking, so has distributed-systems software come of age because of advances in networked file systems and user-transparent remote-execution facilities [10, 49, 187]. Irrespective of how efficiently it is implemented, remote execution incurs overheads due to process migration; naive use of remote-execution facilities may, therefore, cause degradation rather than improvement in performance!

Several researchers have studied intelligent load-balancing strategies, which determine whether to execute an incoming task remotely and, if so, at which site. Such strategies come in a variety of flavors (Figure 2). *Centralized* strategies limit the information-gathering and decision-making responsibilities to a central site; the *decentralized* ones distribute such resposibilities among the sites. Centralized strategies are suitable for scheduling interdependent tasks; and decentralized strategies, for scheduling independent tasks. Also, decentralized strategies are the preferred alternative when tasks can be initiated at any site of the distributed system. *Static* strategies determine ahead of time where to execute each incoming task, without considering the run-time loading conditions. They are suitable for scheduling tasks having predictable resource requirements on machines having predictable load variations because they incur no

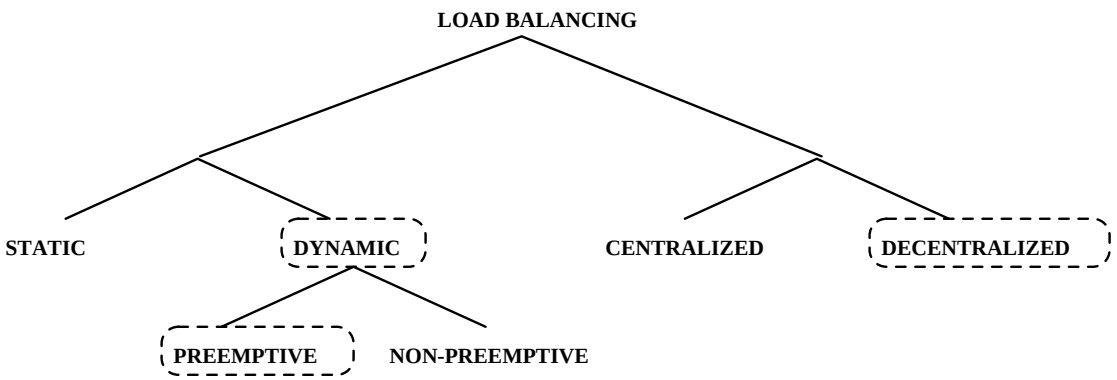


Figure 2. Classification of load-balancing strategies

Load-balancing strategies are classified on the basis of (i) the type of information used in scheduling tasks; (ii) whether tasks once placed can be rescheduled; and (iii) whether information-gathering and decision-making resposibilities are restricted to a single site. The strategies of interest to us are shown inside dashed boxes.

information-gathering overheads at run time. *Dynamic* strategies make scheduling decisions at run time, using *load indices* to distinguish between heavily- and lightly-loaded sites. These are more suitable when little is known about either the tasks being scheduled or the loading conditions prevalent during task execution. Dynamic strategies may be further classified as either *preemptive* or *non-preemptive*; the former permit migration of tasks that have already begun execution, but the latter do not. If load conditions were to change drastically during task execution, a preemptive strategy would allow one to rectify bad placements, but a non-preemptive one would not.

The primary function of a load-balancing strategy is to recommend decisions that improve performance. The choice of a performance objective for load balancing depends upon the application environment. For example, if there are dependences between tasks, then the objective should be to minimize the total time to complete a set of interdependent tasks. Otherwise, if tasks are independent, then the objective should be to improve either the average- or the worst-case completion time. Often, the performance of a load-balancing strategy is evaluated with respect to a *base strategy*; examples include *speed-up* over local execution of all tasks, and speed-up over the execution of each task at a randomly determined site. Speed-ups are preferred as performance objectives because they can be used for comparing a strategy's performance across several different sets of tasks. Besides, they offer an easy characterization of poor strategies — those that yield speed-up values below unity. Speed-up may be computed either separately for each task or collectively for entire batches. The former approach is used when evaluating the performance of load-balancing strategies on a stream of independent tasks; the latter, when evaluating on a batch of depedent tasks. Either the average or the worst-case speed-up may be maximized. In typical computing environments, the objective is to improve the average speed-up. Improvements in worst-case speed-up are desirable only in real-time systems, which require bounds on worst-case completion times.

B. Scope of this Thesis

As Figure 1 illustrates, our model assumes that tasks are independent and may originate at any site of the distributed system. Further, we assume the existence of a ‘‘background workload,’’ which varies outside the control of the load-balancing software. Such load may be caused either by site-specific operating-system functions or by tasks that cannot migrate due to the limitations of remote-execution software. We assume no prior knowledge of the behavior of the tasks to be scheduled, except that they are drawn from a large population having stationary (but unknown) mean and variance of relative resource requirements. We treat local execution of every task as the base case, relative to which all other strategies will be evaluated. Since our model assumes independent tasks, our objective is to maximize the average speed-up, computed on a task-by-task basis, over local execution.

The existence of a background workload, whose long-term variation cannot be predicted, rules out static strategies. Further, the lack of knowledge about task length, along with the aforementioned difficulty of long-term prediction, necessitates preemptive strategies, which can ‘‘undo’’ the effects of poor initial placements. Finally, the distributed arrival pattern of tasks, as well as the lack of constraining dependences between them, entails that we prefer the less expensive and more robust decentralized strategies over the centralized ones. Therefore, the focus of this thesis is on dynamic, decentralized, preemptive load-balancing strategies (Figure 2).

Usually, load-balancing policies are programmed manually. Manually programmed policies are either overly conservative (to work reasonably well on all installations), or not portable (because the optimal setting of policy parameters is installation-dependent). Both these conditions lead to unsatisfactory performance when the configuration of the system changes (due to failures or upgrades) or when new application programs are run. The thrust of this thesis is on the development of an *automated* method for improving the performance of parameterized load-balancing strategies. We explore machine learning as a vehicle for such automation.

Load-balancing strategies have two components [3]: *load metrics*, which characterize the level of loading for each site; and *decision policies*, which determine both the conditions under which a task should be migrated and the destination of each task. A load-balancing policy provides rules for using available load metrics to make global scheduling decisions — such as where to execute a task, and whether to accept foreign tasks — as well as decisions about priorities of tasks and sites.

Figure 3 shows the parameterized load-balancing policy targeted for improvements in this thesis. This policy is a generalization of several decentralized dynamic load-balancing policies proposed by researchers and implementors [17, 108, 204]. Even in a small system having, say, just 4 sites, the above policy has 16 systemwide parameters. Each site's policy has 4 parameters — the values of `Reference`, δ, θ_1, and θ_2. The primitive decision metrics used by this policy are the values of `Load`, one per site. Abstract (or derived) decision metrics are exemplified by `MinLoad`, which denotes the least among all the `Load` values.

```
SENDER-SIDE RULES (s)
     Possible-destinations = { site: Load(site) - Reference(s) < δ(s) }
     Destination = Random(Possible-destinations)
     IF Load(s) - Reference(s) > θ1(s) THEN Send

RECEIVER-SIDE RULES (r)
     IF Load(r) < θ2(r) THEN Receive
```

Figure 3. The load-balancing policy considered in this thesis

The sender-side rules are applied by the load-balancing software at the site of arrival (s) of a task. `Reference` can be either `0` or `MinLoad`; the other parameters — δ, θ_1, and θ_2 — take non-negative floating-point values. A remote destination (r) is chosen randomly from `Destinations`, a set of sites whose load index falls within a small neighborhood of `Reference`. If `Destinations` is the empty set, or if the rule for sending fails, then the task is executed locally at s, its site of arrival; otherwise, the chosen site (r) is requested to receive the task. Upon receiving that request, the remote site applies its receiver-side rule. If the rule succeeds, the request is accepted, and the task is migrated; otherwise, the task is executed locally at s, its site of arrival.

C. State of the Art

Traditional approaches to the design of load-balancing strategies require a human designer to specify a formula for computing `Load`, the load index, as a function of the current and recent utilization levels of various resources. Moreover, they require manual setting of all the policy parameters. Not only are the parameters numerous, but also they are sensitive to installation-specific characteristics of hardware devices as well as to the prevalent load patterns.

It is also a common practice in the load-balancing community to use abstract queueing models of computer systems for analytically deriving load-index functions [58, 60, 203]. Almost universally, implemented systems use a function known as *UNIX-style load average* (hereafter, load average), which is an exponentially smoothed average of the total number of processes (including the process in execution) waiting for CPU. The number of processes (n) is sampled once every 5 seconds. Three different load averages are updated each time n is sampled: Avg_1, the 1-minute moving average which covers 12 sampling intervals; Avg_5, the 5-minute moving average which covers 60 intervals; and Avg_{15}, the 15-minute moving average which covers 180 intervals. We illustrate their computation with the formula for Avg_1:

$$Avg_1(t) = e^{\frac{-1}{12}} \cdot Avg_1(t-1) + (1 - e^{\frac{-1}{12}}) \cdot n. \tag{1a}$$

The other two averages are similarly computed by replacing the constant 12 in the exponents with constants 60 and 180, respectively. These load averages may be used for comparing different loading situations, either at the same site or across different sites of an (architecturally and configurationally) homogeneous distributed system.

Table I lists a variety of performance metrics available in a typical workstation-based computing environment. This table shows that measuring the utilization levels of resources other than the CPU — memory, disk, and the network — may not require any hardware modifications. However, several of these statistics are unsuitable for inclusion in a load index because the overhead associated with estimating their values precludes frequent sampling; such statistics include a variety of process-level metrics, which

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

Performance Metric	Type of information available
Number of context switches	Total over 1-second period, Average over 5-second period, Total since boot
Number of system calls	
Number of device interrupts	
Number of pages swapped in/out	
Number of processes swapped in/out	Total over 5-second period, total since boot
Number of processes ready to use the CPU	Instantaneous value, computed every 5 seconds
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	
Number of free memory pages	Instantaneous value and 60-second average
Number of cache flushes	Total over 1 second, average over 5 seconds, 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 milliseconds
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 CSMA 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 milliseconds; the whole interval charged to the process in control of CPU. Other statistics updated asynchronously at the time an event happens.

are sampled only once every 5 seconds. Even if we eliminate these, we are still left with a fairly large set of mutually dependent variables; for example, disk traffic is affected by the number of page swaps and process swaps. Others, such as rate of data transfer, are fixed quantities for a given configuration, and affect only the (fixed) co-efficients of a load index. Ideally, workloads for load balancing should be characterized by a small set of performance metrics satisfying the following criteria: (i) low overhead of measurement, which implies that measurements can be performed frequently, yielding up-to-date information about load; (ii) represent the loads on all the resources of contention; and (iii) can be measured and controlled independently of each other. In the past, Zhou [203, 204] has considered resource-queue lengths (the number of processes waiting for CPU, disk, and memory) in designing load indices. However, these metrics violate the first criterion because their computation involves a search through the operating system's process table. The instantaneous utilization levels on the four basic resources — CPU, memory, disk, and network — constitute a useful set of performance metrics satisfying all three criteria. These metrics are shown in bold font in Table I; indices derived from these metrics have not been studied in the past.

1) Problems with the traditional load-index function

Our model allows configurational heterogeneity, whereunder load average cannot provide meaningful comparison of loading situations across sites. For instance, consider the comparison between a fast site having a load average of 3 and a slow site having a load average of 0. Further, assume that a preemptive round-robin scheduling policy [11] is used for local process-level scheduling, and that the fast site is 5 times faster than the slow site. It is likely that an incoming task will require 20% lesser time to complete at the fast site than the time it would require at the slow site, despite the former site's higher load average!

A more fundamental problem with the traditional load-average function is that it completely ignores resources other than the CPU. Therefore, while load average may be a reasonable indicator of

performance for purely compute-bound tasks, its utility is questionable for tasks that use other resources of contention, such as memory, disk, and network.

The bottom line in any attempt at load balancing should be to obtain a significant speed-up over local execution. Since each incoming task has a finite set of possible destinations, one would like to use the load index at each site to compare alternative destinations in terms of their expected speed-ups over local execution. This is not feasible with available load-index functions.

2) Problems with the traditional method of tuning policy parameters

A widely accepted rule of thumb in the load-balancing community is that dynamic policies should be simple in form, easy to implement, and efficiently applicable [51]. Complex policies incur substantially larger overheads and their benefits are questionable [52]. However, as shown above, even simple policies can have numerous parameters. At design time, little is known about the characteristics of the hardware and software configurations under which a policy will eventually be used; therefore, the tuning of policy parameters is often relegated to the end users. The users rarely have enough insight to analytically optimize policy parameters; they, therefore, resort to trial and error as a means to optimization. The manual effort spent in such trial-and-error learning is tedious, costly, slow, and often unsystematic. Manual tuning of policy parameters is impractical especially since it may be required each time the system is expanded or upgraded.

Since speed-up can be measured only after actually performing an experiment, systematic improvements in policy performance require a systematic trial-and-error approach to parameter tuning. Considering the large number of possible parameter settings, an exhaustive (unguided) search through the parameter space will be prohibitively time-consuming. Even a guided manual search will require enormous human effort. Therefore, *automated* mechanisms are needed for developing a useful load-index function,

as well as for systematically tuning policy parameters.

D. Automated Learning of Load-Balancing Strategies

Automated refinement of metrics and policies based on observed performance may overcome some of the limitations of manually designed load-balancing strategies. Such refinement is naturally viewed as *strategy learning*, gradual improvement in performance of strategies based on experience. There has been much research on strategy learning, and many general tools and techniques are now available for automating this process [114].

This view of *load balancing as a strategy-learning task* immediately suggests how we might tackle the problem at hand. Since load-balancing strategies comprise indices and policies, the strategy-learning task can be naturally decomposed into learning of indices and policies. Learning requires data. As we show below, the collection of such data requires an environment for controlled experimentation. Thus, we need to address three subproblems: automated learning of load indices, automated tuning of policy parameters, and the design of a controlled-experimentation environment for data collection. The rest of this section briefly examines each of these subproblems.

1) Automated learning of load indices

Ideally, given the loading conditions prevailing at the different sites, one would like to rank the sites by how long an incoming job would take to complete on each of them. However, completion times can only be known after the job completes, whereas a decision about the job's destination needs to be made before it starts. Therefore, we need to somehow *predict* how long a job will take using only the information available before a job's execution.

It should be noted that different classes of jobs may sometimes require different load indices. For example, while the index for memory-intensive jobs may pay more attention to memory availability, 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 [44]. However, in our model, we assume no prior information about jobs; we, therefore, restrict our attention to the case of one load-index function per site.

The completion time of a job is not predictable in an absolute sense without prior knowledge of the job's requirements on different resources. Obtaining such knowledge requires extensive analysis of programs, which is complicated by the data-dependent nature of execution traces. Even when such information can be obtained, it is not clear how it can be translated into primitive resource requirements on a distributed system. For example, an I/O-intensive job, when run on a client, generates extreme traffic on the disk as well the network; the same job, on the file server, generates only disk traffic. Therefore, without any knowledge of the resource requirements of tasks, we cannot predict absolute task-completion times.

Notice that our goal in designing an index-function for load balancing is merely to *compare* alternative destinations for the *same job*. Therefore, we will attempt to determine only a relative (site-specific and configuration-specific) measure of completion time. It would, therefore, suffice to predict the *relative* completion times of a job at different sites. We now need a point of reference. Following the example of Hwang, *et al.* [85], we choose an idle file server as a point of reference. That is, every site needs to predict the completion time of an incoming task relative to its completion time on the chosen 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 [116]. In Chapter IV, we identify additional assumptions under which relative completion times can be predicted independently of tasks.

Our objective in learning load indices is to choose with high probability a site that will minimize the completion time of an incoming job (given either the length of the job or the preemption interval). If

different machines have widely different completion times for the same job, then the accuracy of prediction can be low. On the other hand, if the difference between a task's completion times on different sites is small (relative to its completion time on the idle file server), then the accuracy of prediction needs to be high. At the outset of learning, it is difficult to predict how much accuracy will suffice. We know that the accuracy of any convergent, iterative learning procedure improves with training; we, therefore, need to reformulate our problem so that we will know when the accuracy attained is adequate.

Instead of attempting to predict, for each site, the relative completion time of a task at that site, we now predict, *for each pair of sites*, the difference between their relative completion times. Moreover, we judge the quality of learning by the *percentage of correct comparisons*. Then, we can stop refining the individual index functions when the percentage of correct comparisons crosses an acceptance threshold. We need to make sure 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 Chapter IV.

2) Automated tuning of parametrized load-balancing policies

Our goal in tuning the parameters of a given load-balancing policy is to discover a parameter set that achieves optimal or near-optimal average speed-up over local execution. If there were no overhead of migrating tasks between sites, then a simple load-balancing policy — one that would always schedule a job at the site with minimum load — would suffice for centralized load-balancing schemes. However, when used in conjunction with a decentralized load-balancing policy, this policy may cause every site to dump its load on the least-loaded site. If the overhead of migration were much higher than the typical gains in completion time achieved by executing a job at a remote site with lesser load, then another simple policy — one that would always schedule each task locally — would suffice. However, typical overheads of migration fall somewhere between these extremes [29].

Once the migration overheads are taken into account, neither of the two simple policies outlined above will yield optimal performance. Instead, it is expected that the quality of a load-balancing policy will depend on the setting of its parameters. Dynamic load-balancing policies use thresholds and other parameters in deciding when to migrate local jobs to remote sites and when to accept remote jobs from other sites. While it is possible to dynamically adjust thresholds and other parameters in response to changing workloads [92], our approach is to use policy parameters primarily to accommodate the uncertainty in predicting workloads and in estimating migration overheads. We, therefore, assume that policy parameters are (tunable) constants that depend upon the relative costs and benefits of migration, but do not depend upon the loading conditions. Since the relative costs and benefits of migration can only be known incrementally and a posteriori, the tuning process is a trial-and-error process that needs to be automated. Moreover, if we only have limited time for learning, the tuning process needs to be rational in its use of learning time.

There are two general classes of learning techniques: point-based and population-based [1, 185]. *Point-based methods* systematically modify the parameters of policies using prior knowledge of relationships between policy parameters and policy performance. They maintain one incumbent policy and modify it to the point of diminishing returns. Often, the behavior of such methods can be viewed as gradient ascent to the nearest local maximum of performance in the parameter space. The quality of strategies learned using point-based learning is sensitive to the choice of the initial parameter set, as well as to the quality of the local optimum nearest to the initial set. *Population-based methods* maintain a population of parameter sets, devoting more resources to the more promising sets. They employ a generate-and-test paradigm, in which new parameter sets are generated either randomly or through selected breeding of the top few current parameter sets. *Integrated learning* methods combine the advantages of both by maintaining multiple parameter sets, and modifying the active ones using point-based learning.

Certain parameters of load-balancing strategies, especially the thresholds (Figure 3), are amenable to point-based learning. If the incumbent policy recommends migrating a job to a remote site, and if that

migration is subsequently found to perform worse than local execution, then the threshold of sender-side rules at the source needs to be raised, and the threshold of receiver-side rules at the destination needs to be lowered. Similar point-based heuristics are applicable under other conditions. However, certain other parameters, such as `Reference` (Figure 3), are more amenable to the generate-and-test paradigm used by population-based learning methods. Therefore, it is desirable that we use an integrated learning method for tuning the parameter sets of load-balancing policies. This learning problem is further motivated and resolved in Chapter VI.

3) Collecting data for learning indices and policies

Learning programs, such as the ones used to tackle the two previous problems, may derive their power from two very different sources: domain knowledge and data. The strategy-learning task in load balancing is *knowledge-lean* because neither 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 [114], which infer the missing relationships from experimental data, are the only means available to us for the automated learning of load indices and load-balancing policies in a knowledge-lean environment.

Data for load-index learning take the form of before-after pairs, where the first item is a window of load levels on local and shared resources, and the second item is the measured completion time of a task introduced at the end of that window. The associated data-collection task is merely to run each job out of a sample of representative jobs under a wide variety of *realistic* loading conditions. In each such experiment, the utilization levels on various resources must be recorded for a certain interval of time, followed by the initiation of a test job. When the job ends, its completion time must be measured. Both the measurement of load and the measurement of completion time must be achieved with low overhead and high precision, in order to yield credibility to the data.

Data for policy learning require comparative evaluation of different policies. For our purposes, each policy is evaluated by its average speed-up over local execution, for a given set of jobs and under given initial loading conditions. Moreover, the policies targeted for improvement in this thesis exhibit two significant characteristics: (i) since the policy at each site can access neither the pool of jobs at other sites nor any information about future job arrivals, it schedules jobs one at a time in a distributed fashion; and (ii) since it is not known what the effect of past scheduling decisions will be on the loads of local and remote resources, policies make scheduling decisions using only the current and recent loading conditions, but *not* past decisions. Further, the average speed-up of a load-balancing policy over local execution can be computed using only the speed-ups of individual test jobs. Therefore, alternative parameter sets of load-balancing policies can be evaluated separately on each job (or on an interval between two checkpoints of a job), provided that the loading conditions during evaluation resemble those resulting from past decisions.

Figure 4 shows two ways of introducing a test job under the loading conditions required of a typical load-balancing experiment. The traditional approach, shown to the left, is to use a distributed *stream* of jobs (shown as ellipses in the figure) [17, 47, 100, 204]. Suppose that there are j jobs in the stream, each having m checkpoints (where it can be preempted and rescheduled), and that there are N possible sites. Different parameter sets may lead to different scheduling decisions for each job (or checkpoint); with respect to this stream of jobs, there can be N^{jm} possible outcomes. Policy-learning experiments require that we test a large number of different parameter sets; meeting this requirement can be prohibitively expensive in the traditional set-up, which may entail evaluating a substantial fraction of the exponential number of outcomes. Even if some parameter sets differ only in their outcomes with respect to a single test job (Figure 4), the entire experiment needs to be repeated over and over again. Thus, on account of its intractability and wasteful repetition, the traditional method can only be used for evaluating only small numbers of different parameter sets for policy learning.

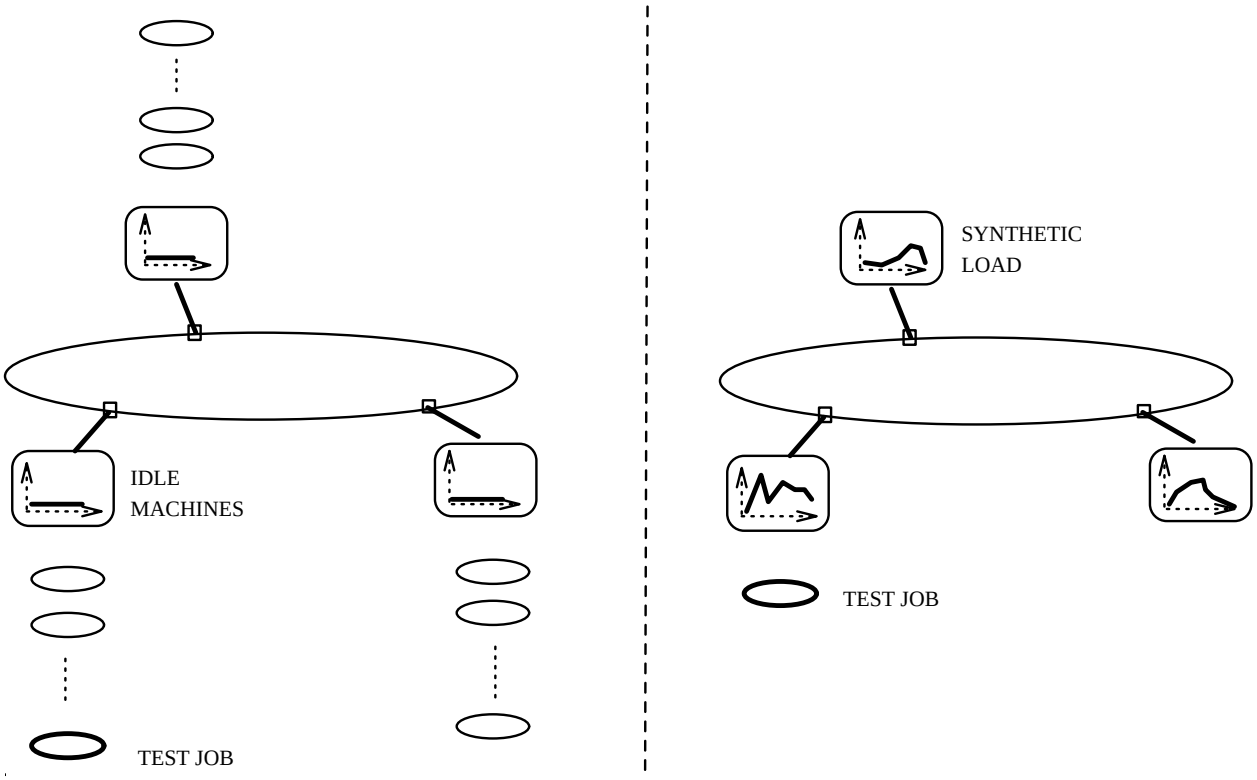


Figure 4. Comparison between experimentation environments

The experimental design shown to the right (Figure 4) represents a way of keeping load-balancing experiments tractable, while still allowing the evaluation of numerous alternative parameter sets for policy learning under realistic loading conditions. It exploits the aforementioned fact that the speed-up of a parameter set on a stream of jobs can be expressed as a linear function of the speed-ups of individual jobs. In this set-up, each experiment involves just one job; the combined effect of all other jobs is captured in a ‘‘background workload.’’ (This is reasonable since independent tasks interact with each other only by competing for resources.)

For scheduling a single test job, there can only be *N* outcomes. In order to evaluate all possible parameter sets with respect to a given test job and background workload, we need only perform *N* experiments, each time executing the test job at a different site. This requires that we regenerate the same background workload *N* times. Workload generation is also required for evaluating parameter sets under

realistic loading conditions: resource-utilization patterns of actual system-wide workloads will need to be recorded and replayed.

During policy learning, point-based learning is employed for performance-driven modification of thresholds at the sites making scheduling decisions for incoming test jobs of an experiment. If there are multiple decisions, as in the traditional set-up, the feedback from performance measurements must be distributed first among the different decisions, and then among the policy parameters leading to each decision. This is called the *credit-assignment problem* [125], i.e., how the feedback (credit or blame for observed policy performance) is distributed among the numerous policy parameters. The two stages of credit assignment — known, respectively, as the temporal and structural credit-assignment problems [173] — have been extensively studied in recent years. While good solutions are known for structural credit assignment [14, 140], temporal credit assignment has so far defied effective general solutions. Since the experimental design shown to the right (Figure 4) evaluates each decision individually, it altogether avoids the temporal credit-assignment problem!

The burden in our set-up is on the experimentation environment. Not only must it allow us to *generate realistic loading conditions*, but also it should allow us to *repeat them as often as desired*. If these desiderata were met, then we could perform multiple experiments, each with the same test job but at a different site. Without knowing anything about the policies to be evaluated, we can collect all the data necessary for comparing those policies *off-line* and ahead of time. All we need to do is to first measure an actual system-wide workload, and then replay it repeatedly, each time introducing a job at precisely the same time into the experiment but at a different site of the network. If we repeat such experiments with several different jobs, under several different loading conditions, then we can create a sample large enough for a realistic comparison of alternative policies. Further details of our experimental set-up can be found in Chapter III.

E. Contributions of this Thesis

This thesis examines several key aspects of dynamic load balancing: from measurement and generation of computer workloads, to automated refinement of load-balancing policies via machine learning. This section summarizes our key results, and their anticipated impact on the state of the art.

1) Dynamic workload generation

We present DWG (Dynamic Workload Generator), a program for generating realistic and reproducible background workloads with high accuracy and high resolution. DWG's most innovative feature is its ability to mimic the behavior of a background workload in the presence of a foreground job. The critical functions for measurement and generation of resource-utilization levels are implemented in each site's operating-system kernel, ensuring low overhead. Actual resource-usage patterns of a distributed system can be captured and replayed with high fidelity, test jobs introduced at precise instants, and their completion time measured accurately. Such experiments can be repeated, running the job at a different site each time but under the same background workload. Since a policy's performance is completely determined by where it schedules an incoming job, alternative policies can be compared *under identical loading conditions*. Thus, DWG allows us to perform reproducible load-balancing experiments, a facility hitherto unavailable to experimenters in this area.

2) Learning consistent and comparable performance indices

DWG keeps track of the utilization levels of the key resources at each site: computational, primary memory, secondary storage, and communication. It provides a precise account of the loading conditions prevalent just before a job begins execution. DWG also measures the completion time of that job when it finishes execution. These ‘‘before’’ and ‘‘after’’ data suffice for learning to compare alternative destinations for incoming tasks. The problem is one of *learning to compare functions of multivariate time series*.

We have adopted the neural-network architecture used for learning evaluation functions in Tesauro's renowned backgammon-playing program [179, 180], and added various smoothing and extrapolation capabilities to his method of learning to compare multivariate functions. We also present an innovative learning algorithm that obviates the 'linking of weights' [113] required by Tesauro's original architecture. This modification allows us to use off-the-shelf neural-network-simulation packages [69]. Extensive statistical tests on the load-index functions learned using our *comparator neural network* reveal high positive correlations (at 99% level of significance) between the true ranking of sites and the one induced by the new index functions. Thus, if load indices were computed right before each decision point, and if they could be communicated instantly across sites, then we could (with high confidence) select the destination having the the least completion time for each incoming job.

The comparator neural network transforms the multi-dimensional and highly dynamic measurements provided by DWG into smooth one-dimensional load indices that can be efficiently communicated over the network, and compared across sites in a meaningful fashion, unlike the traditional load average.

3) Automated tuning of policy parameters

Wah, *et al.* [186] have developed a domain-independent population-based learning system, called TEACHER (TEchniques for Automated Creation of HEuRistics), which also accommodates point-based learning [185]. TEACHER rationally schedules limited learning time between *generating* new parameter sets and *testing* the promising ones in the current population. We have developed a point-based learning procedure for adjusting the thresholds of the load-balancing policy shown in Figure 3. We have also developed the domain-specific generation and testing routines required by TEACHER. Generation of new parameter sets is performed by applying *genetic operators* [82], such as mutation and crossover.

Mutation is performed by randomly perturbing the `Reference` parameters of some sites from 0 to `MinLoad`, or vice versa. Crossover is performed by taking two policies with good recent performance, and interchanging their parameters for some randomly chosen sites.

We have found that the choice of parameter set is sensitive to the overheads involved in migrating tasks and communicating load indices from site to site. Further, we report experiments with different intervals of computation for the load-index function; these affect the average age of load-index values. Since the quality of a load index degrades with age, the choice of policy parameters is also sensitive to the interval of load-index computation. In all these cases, the integrated (population- and point-based) learning system described above is able to quickly determine the appropriate parameter set with high confidence, given data about the completion times of test jobs, and information about various overheads and delays. Thus, what used to be an unsystematic, manual, and tedious process of discovering new parameter sets by trial and error has been replaced by a systematic, automated, and efficient process of population-based learning. Likewise, what used to be the ad hoc process of setting thresholds of policies based on human experience has now been replaced by an automated performance-driven process of point-based learning.

F. Thesis Outline

Chapter II presents an overview of SMALL, a system for automated learning of load-balancing strategies. Subsequent chapters describe the three major components of SMALL: DWG, the dynamic-workload-generation package, in Chapter III; the comparator-neural-network architecture for learning load indices, described in Chapter IV; and our interface with TEACHER, a system for population-based learning of parameterized load-balancing policies, described in Chapter VI. Each of these three chapters also includes extensive experimental results from our implementation on a network of SUN 3 workstations. Chapter V reviews past work on policy learning, and presents a comprehensive taxonomy of issues and approaches in strategy learning. Chapter VII summarizes our results and defines directions for further research.

CHAPTER II.

SMALL: A SYSTEM FOR LEARNING LOAD-BALANCING STRATEGIES

Small is beautiful.

E. F. Schumacher

In this chapter, we briefly describe our overall approach to the three key issues in learning strategies for load balancing: workload generation, learning load indices, and tuning policy parameters. Our philosophy in building a system for automated learning of load-balancing strategies was to first develop general solutions to each of these subproblems and then integrate them into a working system. This way, lessons learnt and tools developed could be applied to other applications having strategy-learning tasks similar to the one in load balancing. Such spinoffs of our research are discussed in Chapter VII. The rest of this chapter is organized as follows. Section A describes the overall architecture of SMALL, our learning system. Section B describes the various phases of operation of SMALL: data collection, load-index learning, policy learning, and application.

A. Architecture of SMALL

Figure 5 shows a layered view of 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 load-balancing policies (TEACHER). The functions of DWG, the lowermost layer of SMALL, are: (i) precise measurement of resource-utilization information; (ii) precise generation of recorded loads; (iii) initiation of foreground test jobs at precise times; and (iv) measurement of job-completion time. After the raw measurements supplied by DWG have been preprocessed using filtering and extrapolation, they are used by the local load-index function for computing a load index, a scalar quantity that can be used for comparing alternative destinations for local incoming

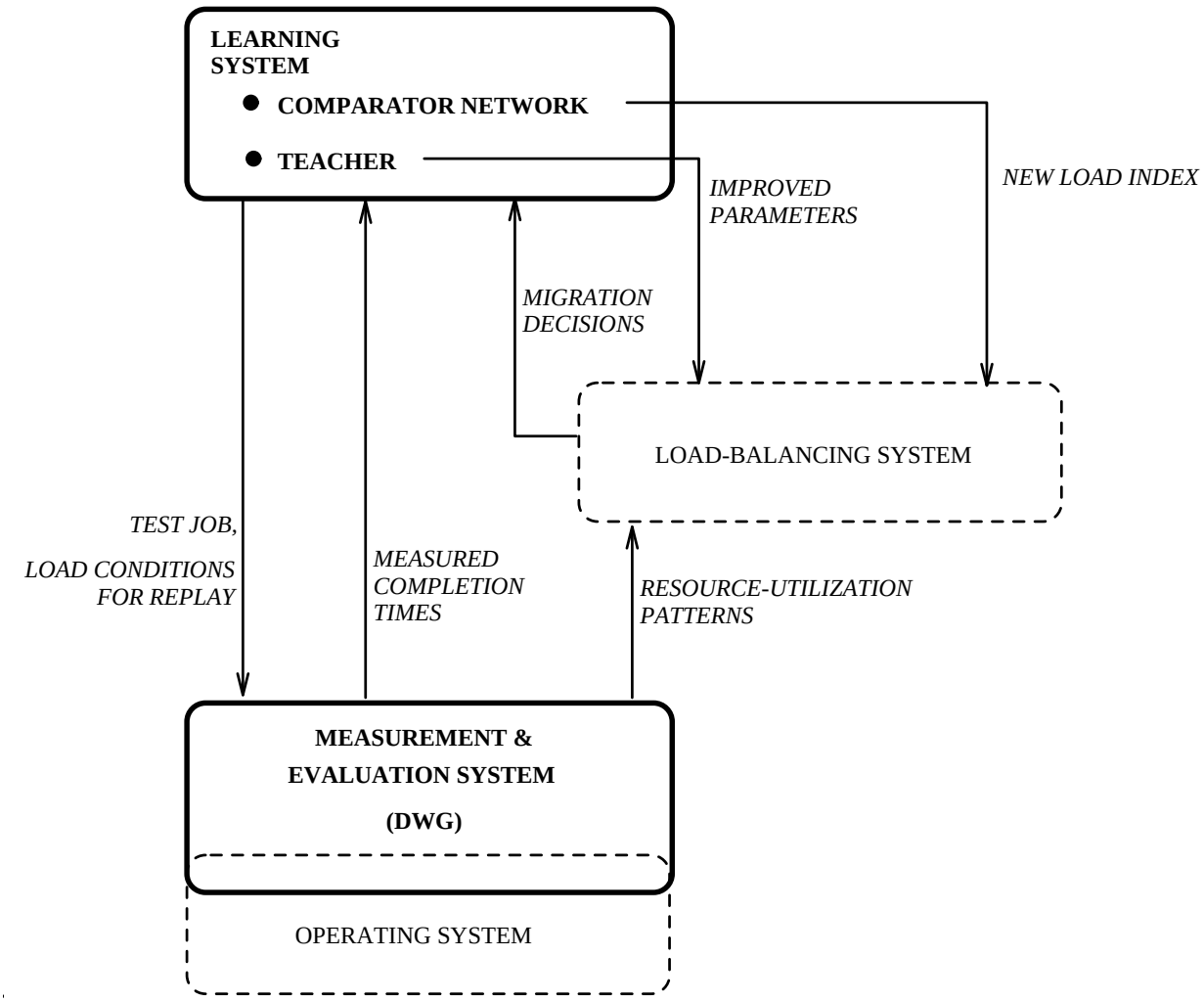


Figure 5. Overview of SMALL

jobs. The given load-balancing policies use the load indices, along with other policy parameters, in order to determine the most appropriate destination for each incoming job.

Both the load-index function and the parameters of load-balancing policies can be modified based on the completion-time measurements provided by DWG. Such modifications are carried out by the learning system, which has two components: one to learn a new load-index function for each site, and another to tune the parameters of a system-wide policy.

The load-balancing system implements the policy shown in Figure 3, and converts the primitive measurements provided by DWG's measurement facilities into the more meaningful load indices. This system includes support for communicating the load indices among the sites, as well as for computing abstract performance metrics such as `MinLoad`, which denotes the minimum predicted load index.

We assume the existence of a job-execution facility at each site, which can be invoked either locally or remotely. In the UNIX operating system [11], such functions are provided by *shells*. We also assume that the underlying operating system provides support for initiating processes and for detecting their termination.

Architecturally, therefore, SMALL is implemented in two layers: DWG and the learning system. (These components of SMALL are shown highlighted in Figure 5.) 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 is built on top of the load-balancing system, and has access to its inputs and outputs; it can also access and modify various policy parameters.

B. Operation of SMALL

SMALL's operation can be broken into phases of data collection, learning, and application. The data-collection phase can be subdivided into collection of background workloads and measurement of task-completion times under those workloads. The learning phase can be subdivided into the learning of indices and the tuning of policy parameters. Figure 6 shows a schematic view of the overall operation of SMALL.

Our methodology is based on *off-line learning*, in which data collected ahead of time are used for modifying a load-balancing strategy under controlled conditions. The way our experiment is organized, only the data-collection phase requires dedicated use of the distributed system under consideration; therefore, the duration of this phase can be considered 'down time' in system managers' parlance.

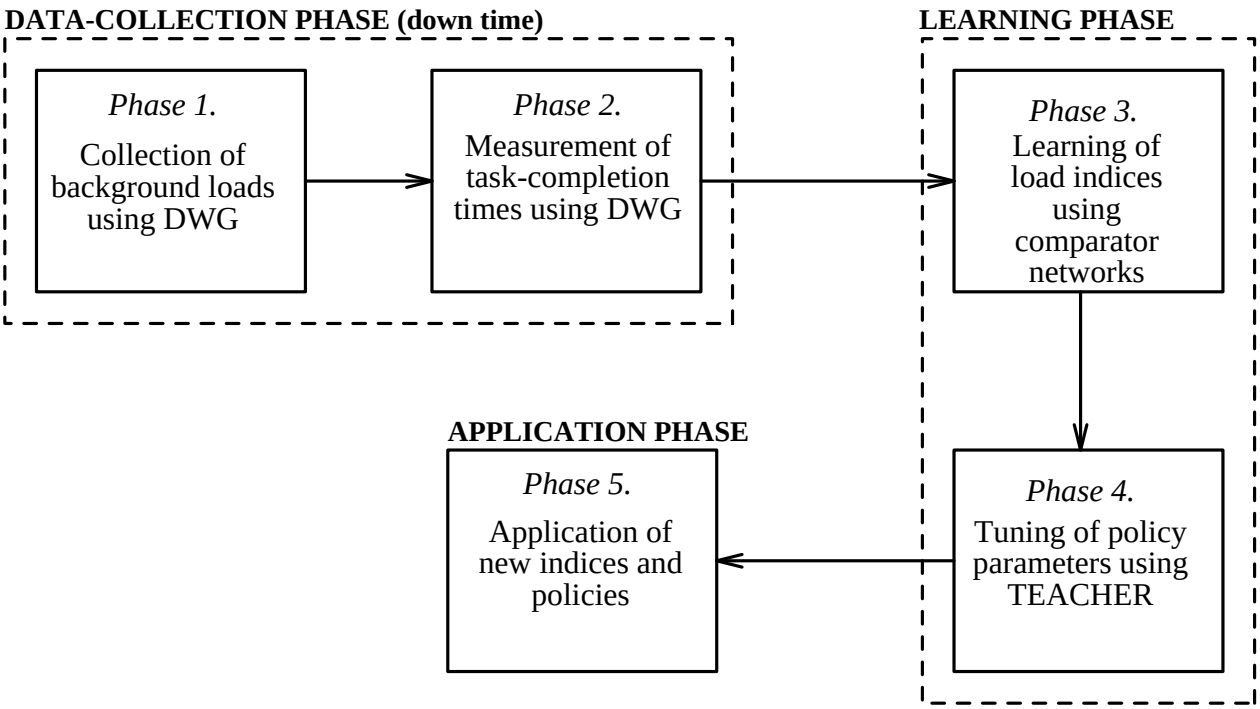


Figure 6. The five phases of SMALL's operation

Experiments for learning indices and policies depend only upon the data collected during the first phase; these experiments can be performed at a suitable site either on or off the distributed system whose policies are being tuned. The first learning phase generates load-index functions, one per site; these can be used for converting the multivariate and highly dynamic resource-utilization information collected during the first phase into univariate and slowly-varying load indices. In the second learning phase, these load indices, along with the completion times measured during the second phase of data collection, 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. When the systems come back up for the application phase, the effects of earlier phases are visible only in the new load-index functions and modified policy parameters. *Off-line learning incurs no run-time overhead!*

Off-line learning is the right approach when the loading conditions during the learning phase resemble those that occur during the application phase. This is the case when the short-term characteristics of load patterns are stationary (do not vary with time). Non-stationary environments may require on-line learning, and SMALL's methodology will not be applicable. Just because SMALL operates off line, that does not mean it is a one-time learning system. On the contrary, it can be used for retraining load-balancing strategies when new sites are added to the system, and when faster devices or newer operating systems are installed at existing sites.

The rest of this section describes the various phases of SMALL's operation in greater detail.

1) Data collection

In the first phase of data collection, background load patterns are recorded for future replay. SMALL supports two kinds of experiments: (i) recording load patterns that occur naturally on a typical system; and (ii) recording load patterns that occur at typical decision points during the application phase. Figure 7 shows the schematic of an experimental set-up for collecting such patterns. The components of DWG are shown highlighted in this figure.

The experiment begins with the measurement of a system-wide log on an idle system (labelled P(0) in the figure). Off line, this log is instrumented by certain trap-insertion routines (labelled T in the figure). These routines make use of global jobfiles (the jobfile for the k'th load pattern is labelled F(k)), which contain information about the arrival times and execution sites of various jobs used in generating the background workload patterns. The global jobfiles are partitioned into local jobfiles (the jobfile for site A is labelled f(k,A)), which are then given to the job-manager processes (labelled J) at each site. Likewise, local instrumented logs (the log for site A is labelled p(0,A)) are given to the generator processes (labelled G), which implement a buffered transfer of data from these logs into the kernel. At precisely the starting times of a site's jobs, its kernel notifies the local job manager, which initiates test jobs using the local job-execution facilities. The resulting system-wide load patterns (labelled p(k,A) for site A) are

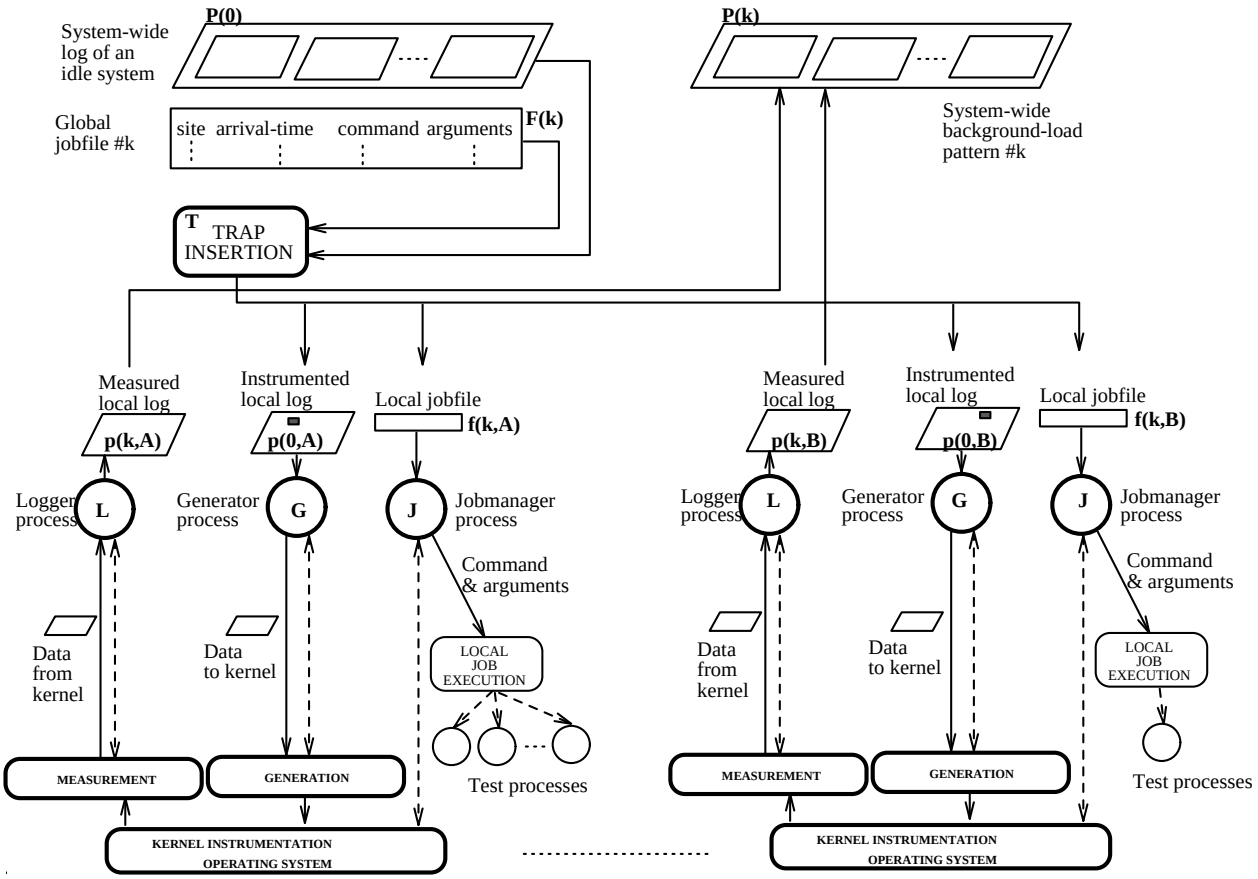


Figure 7. Operation of SMALL: Collection of background-load patterns

recorded by the measurement functions of DWG. Periodically, these records are removed from the kernel by a logger process (labelled L), which writes them out into a file. These files constitute a system-wide background-load pattern (the k'th background workload pattern is labelled P(k)).

At the end of the first phase of data collection, there is one system-wide background-load pattern per global jobfile. The second phase of data collection (Figure 8) is similar to the first, except in three aspects: (i) the global jobfile (labelled F(j,s)) contains only one test job, j, to be run at some site s; (ii) instead of using the load pattern of an idle system as a background load, this phase generates background workload using the load patterns collected during the first phase (labelled P(k) in the figure); and (iii) when the test job terminates, the logger records not only the resulting load pattern but also the completion

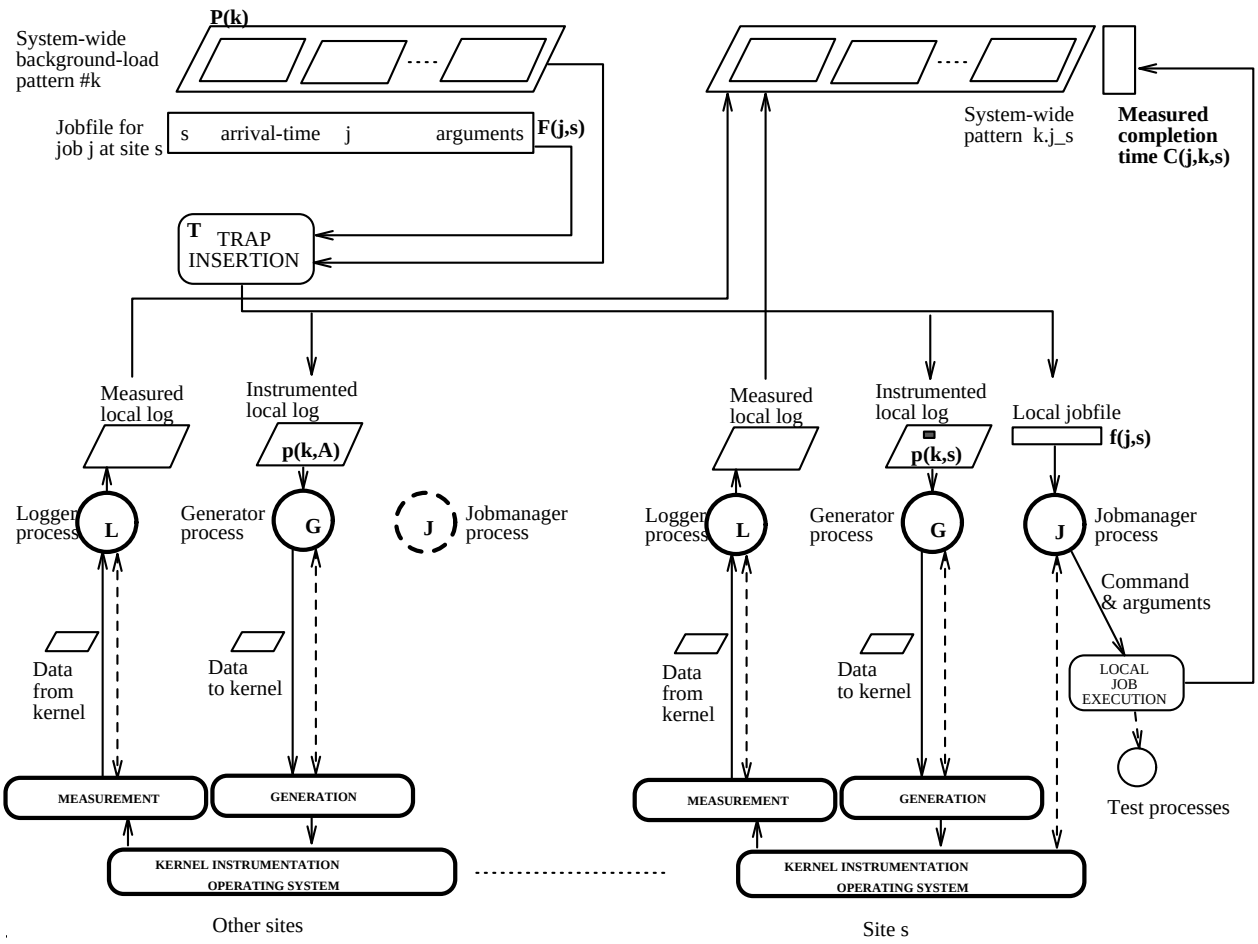


Figure 8. Operation of SMALL: Measurement of completion times

---

time of the test job (labelled C(j,k,s)).

At the end of the data-collection phase, we have two kinds of records: (i) log patterns, labelled k.j_s in Figure 8, describing the actual load conditions prevailing when job j is executed at site s under background-load pattern k; and (ii) the completion time, labelled C(j,k,s) in Figure 8, of job j when it is executed at site s under load pattern k.

2) Off-line learning of load indices

Figure 9 shows a schematic of SMALL during its index-learning phase. Notice that, at any given instant, the training algorithm simultaneously considers the load-index functions at two different sites. The reasons for solving a ‘‘learning to compare’’ problem rather than a ‘‘learning to predict’’ problem were alluded to in Chapter I: essentially, the former approach prevents overtraining by stopping when the

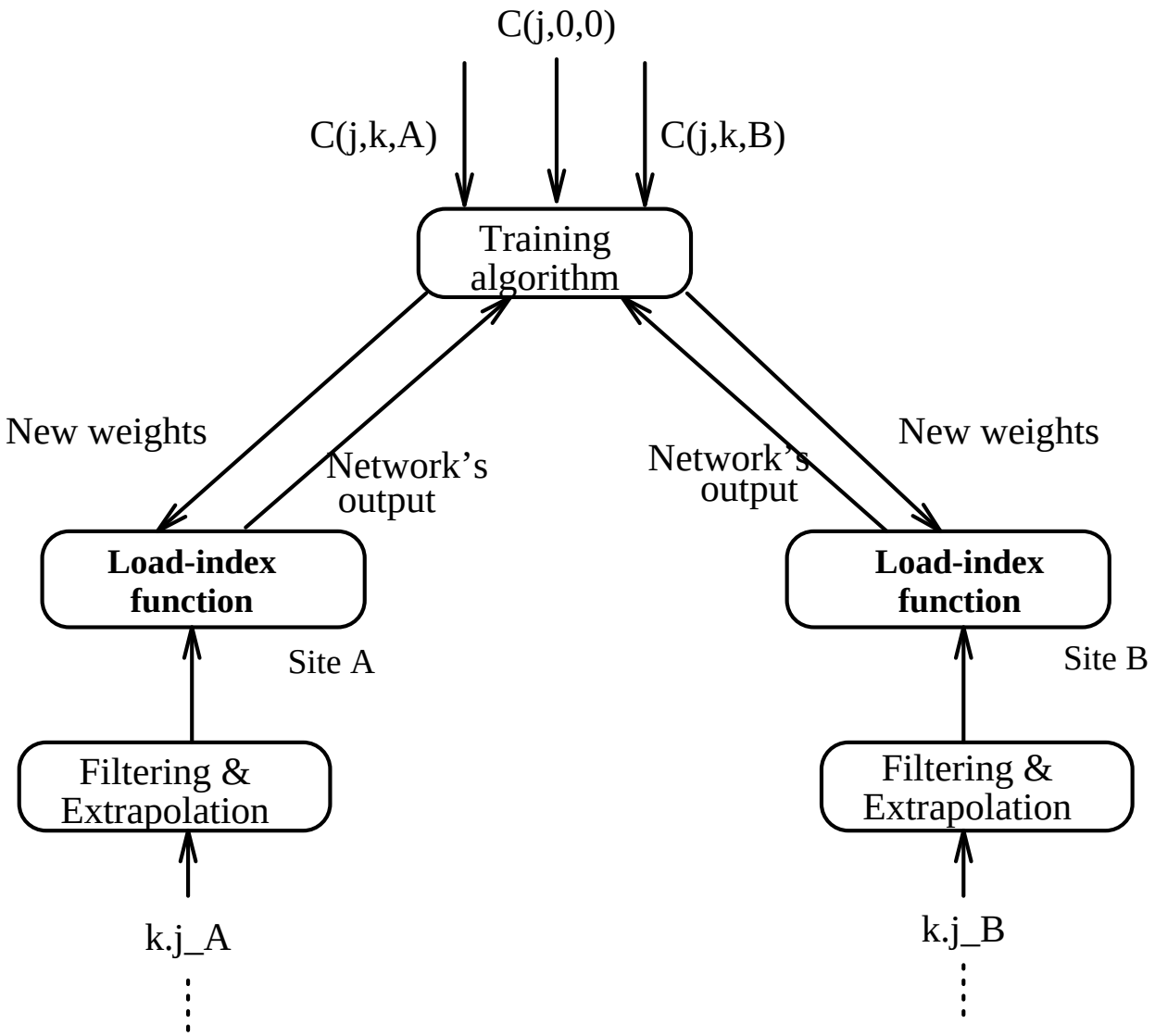


Figure 9. Operation of SMALL: Load-index learning

accuracy of load-index functions is adequate for comparing alternative destinations for the same job.

Information flows for a typical learning trial are shown in Figure 9. Each trial involves a randomly selected job (j), a randomly selected load pattern (k), and a randomly selected pair of sites (A and B). First, the raw utilization patterns recorded earlier by DWG are smoothed using lowpass filters [78]. Smoothing helps in detection of trend; the smaller the cutoff frequency of the lowpass filter, the greater is the degree of smoothing, and the longer is the interval to which the trend can be extrapolated. Trends are extracted from filtered patterns by means of linear and exponential extrapolation. The former fits a straight line to a recent window of smoothed values, and extrapolates using that line; the latter fits an exponentially growing/decaying function. The extracted trend information is fed into a parameterized load-index function. The outputs of load-index functions are then compared. Based on this comparison, the training algorithm suggests new parameters (weights) for the two functions involved. The training algorithm uses the completion times C(j,k,A) and C(j,k,B), measured by DWG during the data-collection phase; its goal is to drive the difference between the outputs of the two load-index functions toward the *true* relative difference, $\frac{C(j,k,A)-C(j,k,B)}{C(j,0,0)}$, where C(j,0,0) is the completion time of job j on a chosen idle file server. Our configuration, shown in Figure 9, simultaneous trains multiple comparable functions; we refer to this as *the comparator configuration*.

Details of the learning experiment can be found in Chapter IV. Suffices it to say that after repeated trials with different jobs, sites, and load patterns, the error of comparison between different sites begins to converge. If this error is acceptably small, then the current set of load-index functions is selected. Otherwise, one needs to repeat this phase with a different set of random initial parameters for the load-index functions.

3) Off-line tuning of policy parameters

By off-line tuning, we mean 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. Policy decisions are computed by applying the sender-side and receiver-side rules shown in Figure 3. Evaluation of decisions requires information about completion times of various test jobs under different loading conditions. All the policy-learning experiments are planned ahead of time so that the completion times necessary for evaluating policy decisions can be measured (ahead of time) using DWG; such measurements are carried out during the second phase of data collection described above.

Given the large number of parameters and the possibly many values each parameter can take, an uncontrolled search for a good parameter set for the given load-balancing policies could take an exorbitant amount of time. Figure 10 shows the schematic of our policy-learning system, including information flows between it and the scheduling system. Each policy has the form shown in Figure 3. The next section describes how the policies are instantiated when a new job arrives. Our only concern in this section is with the inputs and outputs of the load-balancing policy.

The inputs of a load-balancing policy are described by a *test-case* (Figure 10), 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 during this second learning phase 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, p. 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

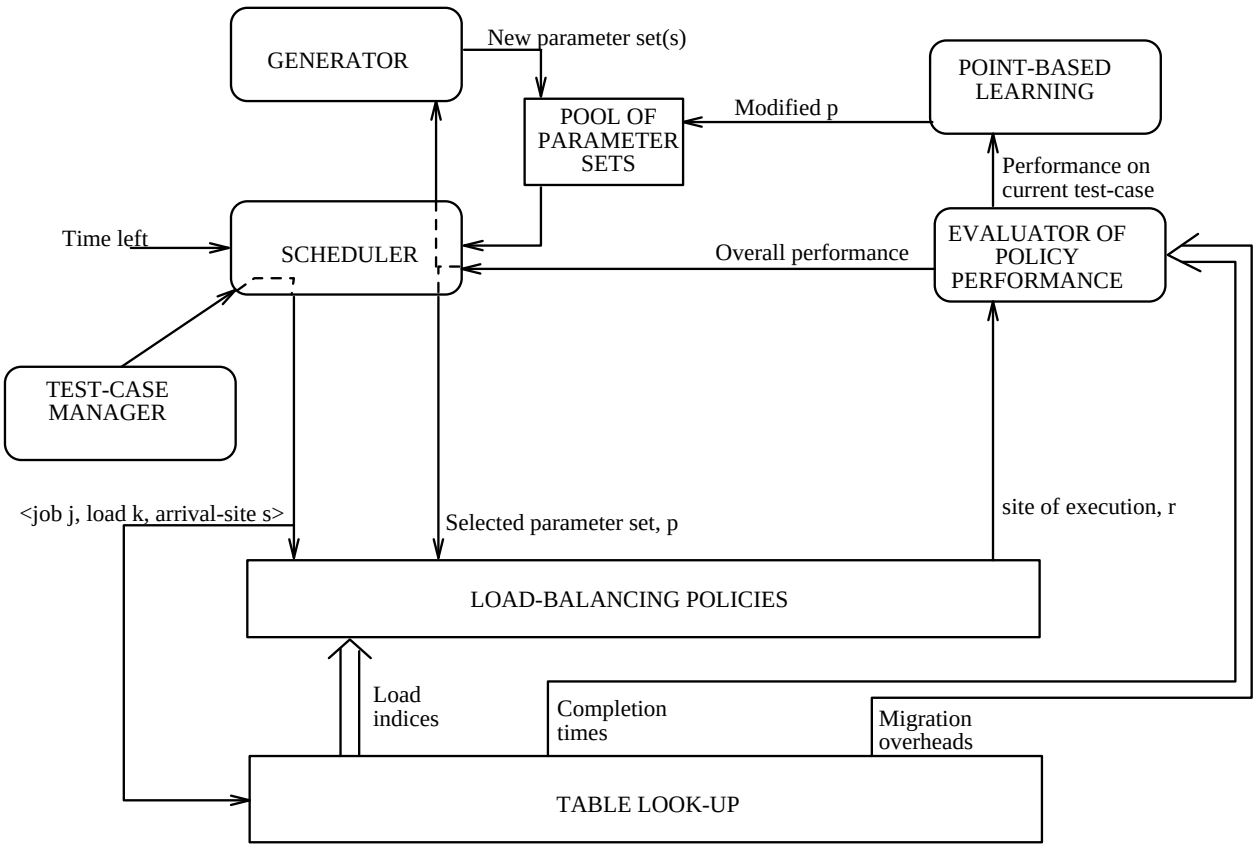


Figure 10. Operation of SMALL: Policy learning

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 parameterized by p can be computed as follows. First, the record C(j,k,r) — the completion time of job j at site r under load-pattern k — is consulted. Next, if the chosen destination r is not the same as the site of arrival s, then an overhead of migration 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 C(j,k,s), the completion time under local execution. Using these numbers, the speed-up achieved by parameter set p for the given testcase can be calculated.

The *Evaluator of Policy Performance* (Figure 10) 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 theory and implementation of the *Scheduler* were developed by Wah, *et al.* [185, 186].

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.

Details of the policy-learning phase are presented in Chapter VI. At the end of this phase, one can select a (possibly new) parameter set that, with high likelihood, will have the best performance of all parameter sets tested, on future test-cases drawn from the same population as the given pool of test-cases.

4) Application phase

With the new load-index functions and the selected policy parameters in place, the scheduling system is now ready for application. Figure 11 shows the information flows during the application phase. At the site of arrival (site j) of an incoming job, the sender-side rules of site j are instantiated. This figure shows the case when a remote destination (site i) is selected by the sender-side rules of site j. A request is made to site i, which instantiates its receiver-side rules. If the receiver-side rules at site i succeed, then the site of arrival (site j) initiates a job transfer by remotely invoking the job-execution facilities at site i.

Implementing the application phase requires that the filtering and extrapolation facilities, as well as the computation of the load-index functions, be made on line. Since our present experimental set-up does not implement these functions on line, we can only simulate the application phase by table look-up, as

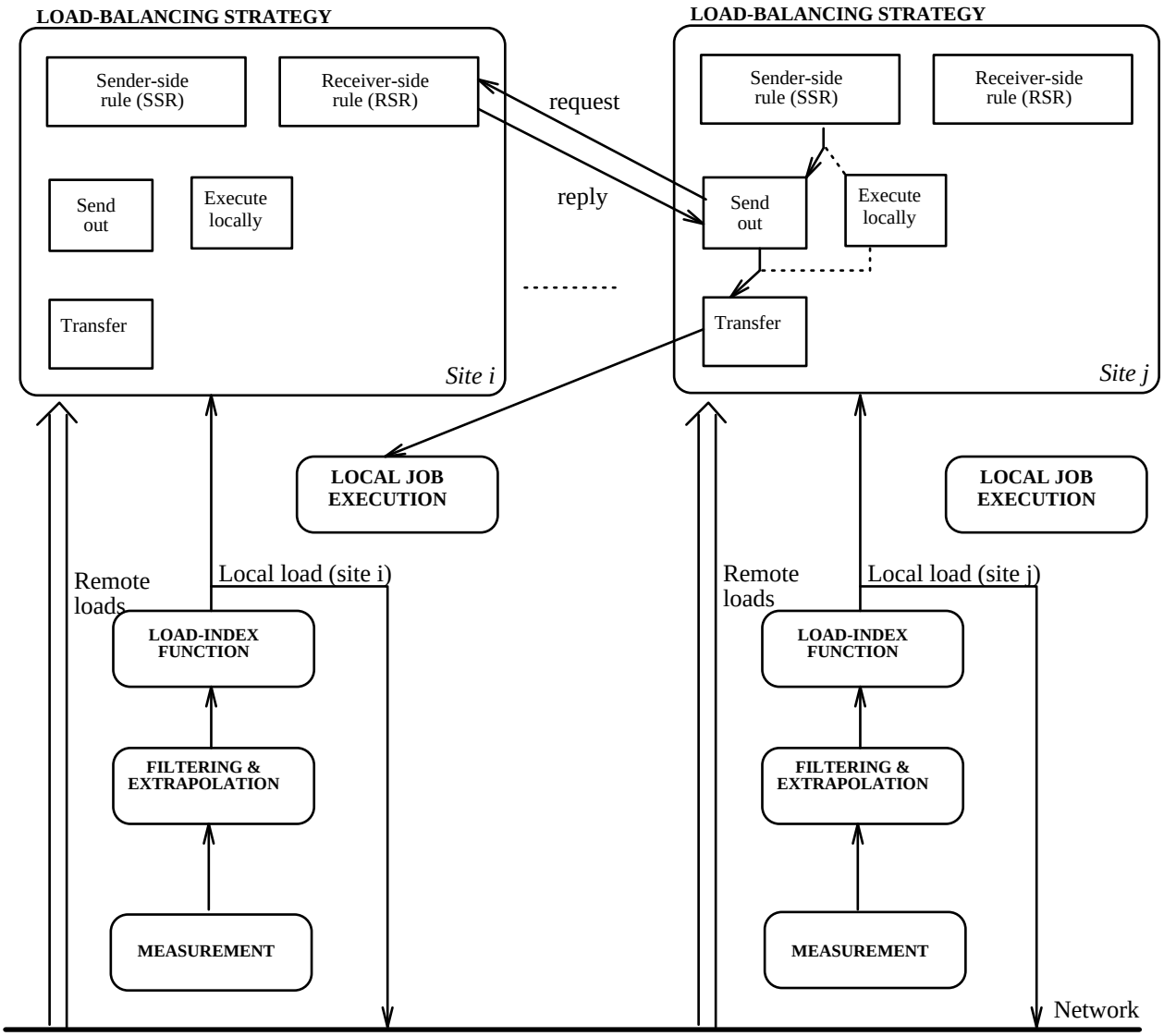


Figure 11. Operation of SMALL: Application phase

illustrated in the previous subsection.

C. Summary

Table II summarizes the times required by different phases of SMALL, our automated system for learning load-balancing strategies. The typical times reported in the rightmost column are based on our

Table II
Times required by different phases of SMALL's operation

Event	Work Required	Formula for time needed	Typical time†
Down time for data collection (phase I)	Recording p different background load patterns, each l time units long	$l(p+1)$	6¼ hours
Down time for data collection (phase II)	Measurement of completion times of n different benchmark jobs under p patterns, each l time units long, on d configurationally-distinct sites	$ndpl$	10 days
Learning times for load-index learning (phase III)	Learning to compare the completion times of N different checkpoints of benchmark programs, measured under p different load patterns, on d distinct sites; each comparison takes c time units and the iterative learning procedure requires e passes through the data.	$Ndpce(p+d-2)$	15 hours
Time for tuning the parameters of load-balancing policies (phase IV)	Learning new parameter sets for load-balancing policies under s different delay models; many different policies are tried, but at least k of them are expected to undergo full evaluation on f different test-cases. Each test-case can be evaluated within t time units. Each of the s policy-learning experiment must finish within a deadline of T time units	$s \cdot max(T, fkt)$	3 hours
New sites, new configuration, or OS upgrade	Data collection	10 days downtime	
	Retraining comparator networks off line	15 hours	
	Tuning policy parameters off line	3 hours	
Removal of a site, or new load-balancing software	Tuning policy parameters off-line	3 hours	

† Values used in estimating typical times are: $n=10$, $p=24$, $d=4$, $l=15$ minutes, $N=60$, $c=0.06$ seconds, $e=6$, $T=15$ minutes, $s=12$, $f=1,000$, $k=4$, and $t=0.2$ seconds.

experience with data collection on a network of four Sun 3 workstations, and off-line learning on a Sun Sparcstation 10, Model 20. We now explain briefly our derivation of the formulae for times required. In phase I, first an idle load pattern needs to be measured and then p others. Since each is l time units long, we get the formula shown. In phase II, n jobs need to be run on d different sites under p different load patterns. Once again, since each experiment takes l time units, we get the formula shown for phase II. In

phase III, one needs to compare, for each checkpoint of a benchmark program, both different loading situations at the same site and loading situations across different sites at the same time. The numbers of such comparisons are, respectively, $Np(p-1)d$ and $Npd(d-1)$. Since each comparison takes c time units, and since learning requires e passes through the data, we obtain the formula shown for phase III. Finally, in phase IV, an experiment can (and often does) use up all the time available (T units) for evaluating different parameter sets on different test-cases; alternatively, one can stop when k different parameter sets have been fully evaluated on f test-cases. Since population-based learning will not go through with evaluation for poor-quality parameter sets, these k sets are likely to be good. Also, the time spent on poor parameter sets is usually negligible compared to the time for full evaluation of good candidates. The entire learning experiment may need to be repeated if there is a change in the nature of either the migration overheads or the age of the load index; s different delay models are considered. Since the evaluation of each parameter set on one test-case requires t time units, we obtain the formula shown for phase IV.

In this chapter, we have outlined our approach to the automated learning of load-balancing strategies for the distributed-system model described in Chapter I. Subsequent chapters present details of our approach: Chapter III describes the implementation of DWG; Chapter IV, our algorithm for learning load indices using comparator neural networks; and Chapter VI, our approach to tuning policy parameters using integrated (population-based cum point-based) learning. Unlike physical-level workload generation and load-index learning, strategy learning is a well-researched area; therefore, in Chapter V, we identify key issues in strategy learning and classify strategy-learning tasks and techniques on the basis of these issues, and further justify our choice of an integrated learning model.

CHAPTER III.

DWG: A TOOL FOR SYNTHETIC WORKLOAD GENERATION

It is a capital mistake to theorize before one has data.

- Sir Arthur Conan Doyle

This chapter describes DWG [117], a generator of realistic and reproducible workloads. DWG employs *synthetic workload generation*, a technique that uses artificial programs to mimic the resource-utilization patterns of real workloads. DWG accurately replays system-wide load patterns recorded by it earlier, providing an ideal setting for load-balancing experiments. Since DWG is implemented inside the operating-system kernel, it has complete control over the utilization levels of local resources. It controls the utilization levels of four key resources: CPU, memory, disk, and network. In order to accurately reproduce the behavior of the process population generating the measured load, DWG gives up a fraction of its resources in response to the arrival of new jobs, and reclaims these resources when the jobs terminate. Section B describes our objectives and constraints in designing an experimentation environment for load balancing. Section C describes DWG's architecture and Section D, its operation. Section E compares different settings for DWG's parameters, and evaluates the synthetic workloads generated using DWG against true workloads observed in corresponding control experiments.

A. Design Goals

Our primary goal is to create an experimentation environment that will allow us evaluate alternative load-balancing policies. In order to compare alternative policies, we need to perform several experiments, each with the same background workload pattern but with a different site of execution for the foreground job. (Afterall, two policies differ in their performance only when they schedule an incoming job at different sites. No matter how many policies we need to compare, the number of possible destinations for

an incoming job is finite and equals the number of sites in the distributed system.) Our secondary goal is evaluate alternative policies under *realistic* loading conditions. In particular, we wish to be able to record and replay workloads that arise in *interactive* environments: i.e., workloads that are highly *dynamic* and include the resource-utilization patterns caused by a variety of *asynchronous* events, such as mouse and keyboard interrupts. The problem of recording and replaying workloads, called the workload-generation problem, is addressed in this chapter.

Test jobs and background workloads affect each other by competing for resources. For real workloads, such competition is resolved by a resource scheduler, which allocates resources among competing processes. While a background load pattern is being recorded, the process population generating that load has complete 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 amounts to a *feedback path* from the experiment to the experimentation environment; it is an important characteristic of the workload-generation problem considered in this chapter.

Since the performance of the test job under a given background workload depends solely on the resources utilized by that workload, it is natural to represent workloads by their resource-utilization patterns. This is called the *physical-level view* of a workload [59]. However, in order to model the feedback from a test job to the generated workload, one needs to adopt a *process-level view*. Representing asynchronous events such as keyboard interrupts and context switches at the process level 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 timings of interrupts and context switches, and replaying them at the same fine grain. Obtaining and regenerating such process-level workloads is prohibitively expensive because it entails costly hardware instrumentation for collecting information and high-resolution timers for driving the

generator.

It follows from the above that there is an incompatibility between the grain of the resource-scheduler and the grain at which workload needs to be recorded and replayed. The constraints of our problem dictate that we represent and regenerate workloads at the physical level, without recreating the entire process population. In this case, we need to artificially model the process-level interactions between the recorded load and test jobs, and use our model to (dynamically) adjust the amount of generated load while there are active foreground test jobs on top of the replayed workload.

B. Previous Work

1) Workload generation without feedback

The design of a workload generator depends on the purpose it is used for. There are several problems in performance evaluation of computer systems where there is no feedback from the mechanism or policy being evaluated to the experimentation environment. For example, in evaluating alternative implementations of file systems, it can be assumed that the file-access patterns of a user are implementation-independent [23, 44, 93]. Another example is in the evaluation and refinement of virtual-memory algorithms using traces of memory references. Once again, the memory-access patterns of programs can be assumed to be independent of (say) the particular cache-coherence protocol being evaluated. Figure 12 shows the schematic of a workload generator without feedback. In this case, the workload generated does not depend upon the mechanism being measured or modeled. Most existing synthetic workload generators follow this paradigm [48, 63, 74, 98, 139, 168]. While adequate for their specific applications, such generators are not appropriate for our workload-generation problem because of the feedback between test jobs and background workload.

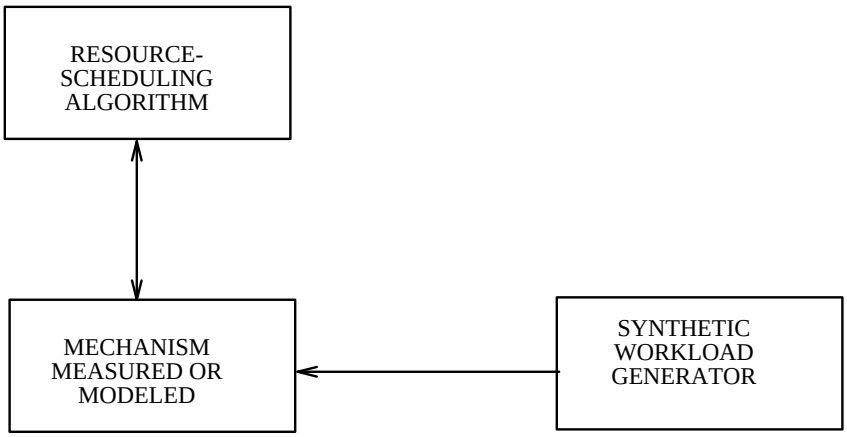


Figure 12. Workload generation without feedback

2) Workload generation with feedback

The traditional approach to developing experimentation environments for load balancing uses a stream of real user commands to create the workload: earlier jobs form the background workload for later jobs [19, 47, 204]. Since this approach generates and represents workloads at the process level, it is incapable of modelling asynchronous events, such as arrival of electronic mail and mouse interrupts, on account of its large granularity. Since foreground-background interactions are handled by the resource scheduler (Figure 13), no adjustment in generated load is required. The key problem with this approach is

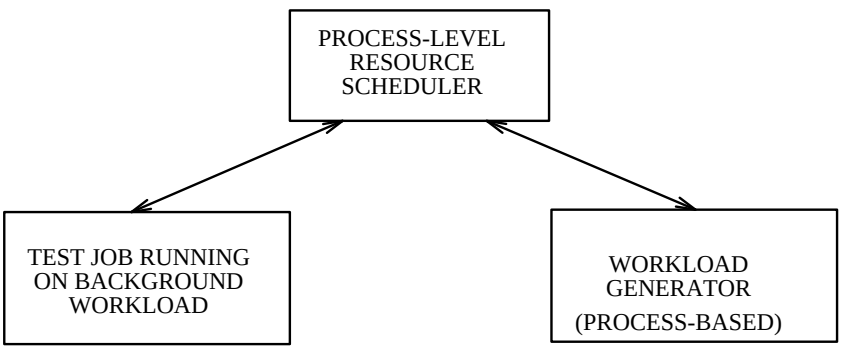


Figure 13. Workload generation with feedback using real workloads

that real workloads cannot be precisely represented using only user-level programs and commands.

One possible way to combine the expressiveness of physical-level workloads with the correct scheduling behavior of process-level workloads is to model the physical-level behavior of all the processes, including system processes. This amounts to recreating an entire process population. (See Figure 14.) As already discussed, the quantity and the rate of information necessary for reproducing such a workload precludes feasible implementation in software alone. Even if hardware support for measurement is available, it is non-trivial to model a large process population with complex interactions.

Our approach (shown in Figure 15) is to represent and regenerate workloads only at the physical level. In order to simulate foreground-background interactions, we consider a simplified model of the resource scheduler, and collect information during measurement and generation to drive this model. The simplified model requires that process counts be recorded with the recorded load, and that the generator keep track of the number of test jobs introduced on top of the workload being replayed. Using these

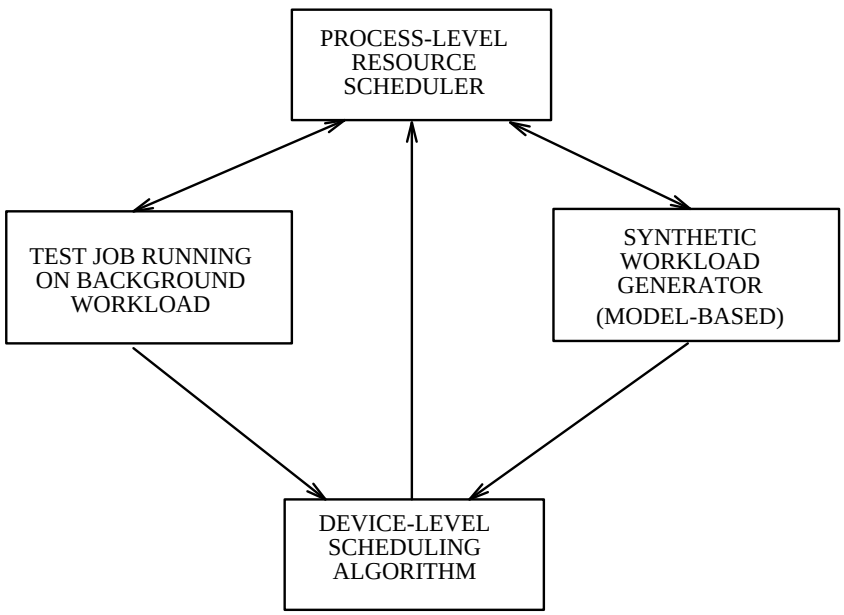


Figure 14. Workload generation with feedback using synthetic processes

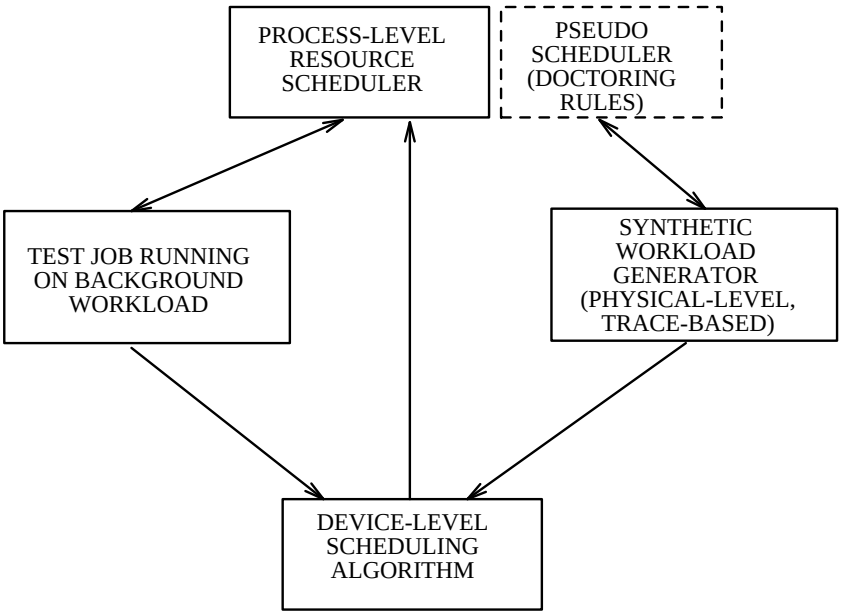


Figure 15. Workload generation with feedback using dynamic doctoring

counts, the model provides a way to dynamically alter generated loads in response to initiation or termination of test jobs; formulae for computing load reductions are encoded in *doctoring rules*, which make the workload generator behave as though it were under the control of the resource scheduler even when it is not. The architecture and operation of our generator are described next.

C. Architecture of DWG*: A Dynamic Workload Generator*

Figure 16 shows the basic architecture of DWG. The components of DWG include processes (shown as unshaded dark boxes in the figure), callable kernel routines (shown as shaded dark boxes), and certain buffers and files (shown as boxes with rounded corners). These include mechanisms for (i) measurement and generation (box labelled **K**); (ii) data transfer in and out of the kernel (boxes labelled **L** and **G**); (iii) asynchronous event management using traps (boxes labelled **H** and **J**); and (iv) dynamic determination of generated load levels (box labelled **D**). These mechanisms are organized into layers of software. The lowest of these layers (shown second from right in the figure) comprises functions implemented

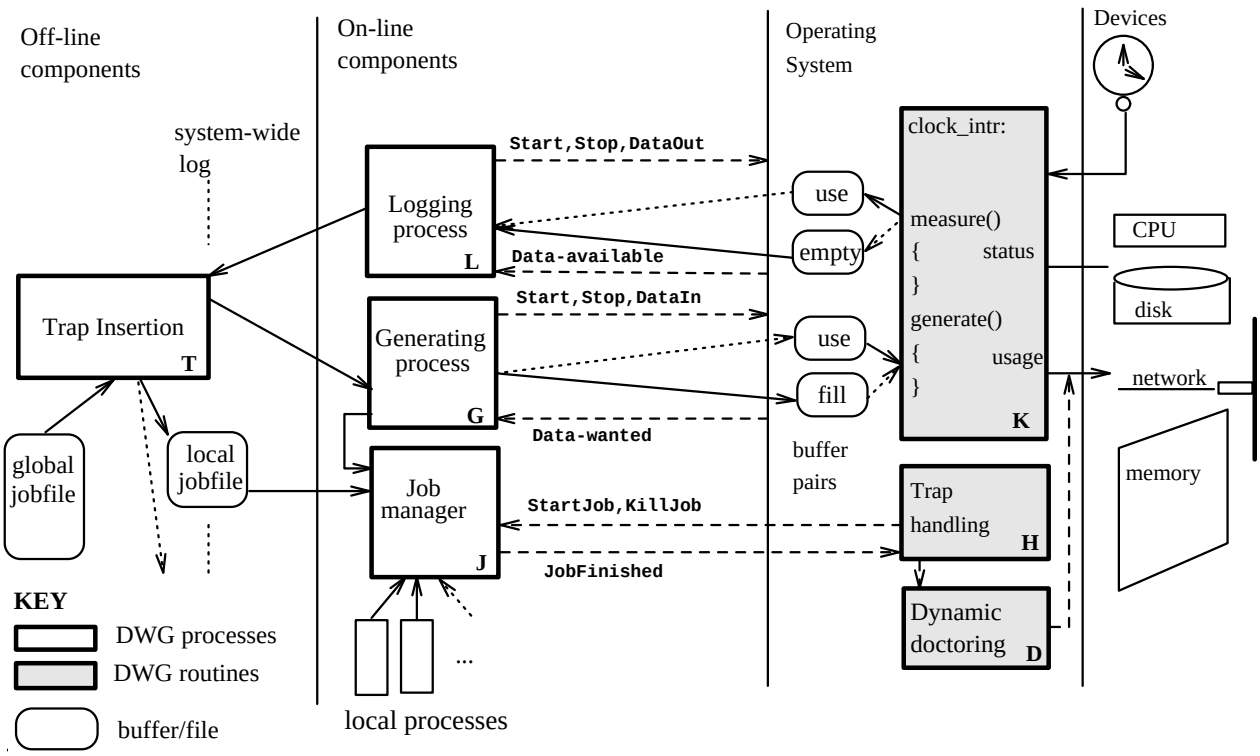


Figure 16. Architectural description of DWG

inside the operating-system kernel. The next higher layer comprises user-level processes that control the kernel-level functions. The topmost layer (shown leftmost in the figure) comprises off-line mechanisms for controlling the initiation and termination of foreground jobs.

1) Functions implemented inside the kernel

The core of DWG consists of its measurement and generation routines (boxes labelled **K** in Figure 16); these, respectively, measure and control the utilization levels of resources, such as CPU, memory, disk, and network. At every clock interrupt, the measurement routines estimate the current utilization as follows. If the CPU was busy (under the control of a user process) when the clock interrupted, then it is assumed to have been busy for *c* percent of the time since last interrupt, where *c* is a tunable parameter of DWG. For the memory, the number of free pages is recorded. For the disk, the number of blocks

transferred since the previous interrupt and, for the network, the number of packets received or transmitted since the previous interrupt, are recorded. Also recorded is the number of user processes contributing to the measured load at any given time.

At every clock interrupt, the generation routines determine how much workload needs to be generated for each resource. For the CPU, the amount of work to be done is expressed as a fraction of the interval between successive interrupts. For the memory resource, the work to be done is expressed as the number of pages to be occupied until the next interrupt. For the disk and network resources, the number of disk transfers and the number of network packets, respectively, are computed.

When there are foreground processes, the computation of generated load employs certain rules, known as *dynamic doctoring rules* (box labelled **D** in Figure 16), in order to compensate for foreground-background interactions. The amount of reduction caused by these rules depends upon the relative sizes of the foreground and background process populations. The trap-handling routines of DWG keep track of the foreground population size. (The size of the foreground process population equals the number of test jobs introduced on top of the background workload being replayed.) The size of the background process population is already recorded in the log being replayed.

While CPU and memory loads are generated inside the kernel, requests for generation of disk and network traffic are passed on to external processes. CPU load is generated by repeatedly executing a segment of pure computational code. Memory load is generated by taking pages out of the pool of free virtual-memory pages, thereby making them unavailable to user processes. The external process responsible for generating disk and network traffic does so by, respectively, performing unbuffered output to a file and broadcasting synthetic packets over the local-area network. CPU and memory loads are generated at each site; disk traffic, only at diskful sites; and network traffic, only at a selected site on the network.

The measurement and generation routines switch buffers upon reaching the end of the current buffer. (See Figure 16.) While switching buffers, they signal their respective external processes (the

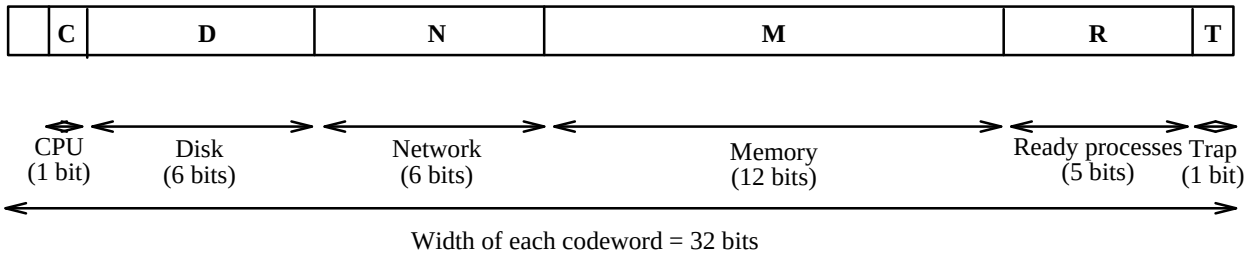
logger process, labelled **L** in the figure, and the generator process, labelled **G**) in order to initiate data transfer. While the kernel is busy with the other buffers, the external processes load/unload the idle buffers. Buffer sizes are chosen large enough that data transfers happen only a few times a minute, and yet small enough that the memory overhead of synthetic workload generation is at worst two or three pages. (This is small overhead compared to the hundreds of pages occupied by typical operating-system kernels.)

2) Functions implemented at the process level

DWG requires three processes at each site: (i) *the logging process* (labelled **L** in Figure 16), which transfers measurements out of the kernel into the log file; (ii) *the generating process* (labelled **G**), which transfers data from past logs (possibly instrumented with traps) into the kernel; and (iii) *the job manager* (labelled **J**), which initiates and terminates test jobs upon receiving signals from the trap-handling routines (labelled **H**) of the kernel, as well as measures the completion time of test jobs. The interface between these processes and the kernel-based functions is via a system call.

In addition to the functions described above, the generating process is signalled by the kernel when there is some disk or network traffic to be generated. It determines how much traffic to generate and does the necessary input/output. The logging and generating processes are also responsible for starting and stopping measurement and generation, respectively, inside the kernel.

In DWG, it is possible to synchronize measurement and generation so that measurement begins and ends exactly when generation does. This capability allows us to compare actual and generated loads, as explained in Section E, and allows us to translate the time of occurrence of an experimental event into the offsets of the corresponding codeword in a log file. Figure 17 shows the typical format of a DWG log, including the formats for encoding resource-utilization information in codewords, and for managing asynchronous events using trap commands. (*Traps* are data bytes in a special format.) Upon hitting the trap command, the trap-handling functions of DWG (labelled **H** in Figure 16) queue up the trapped requests,



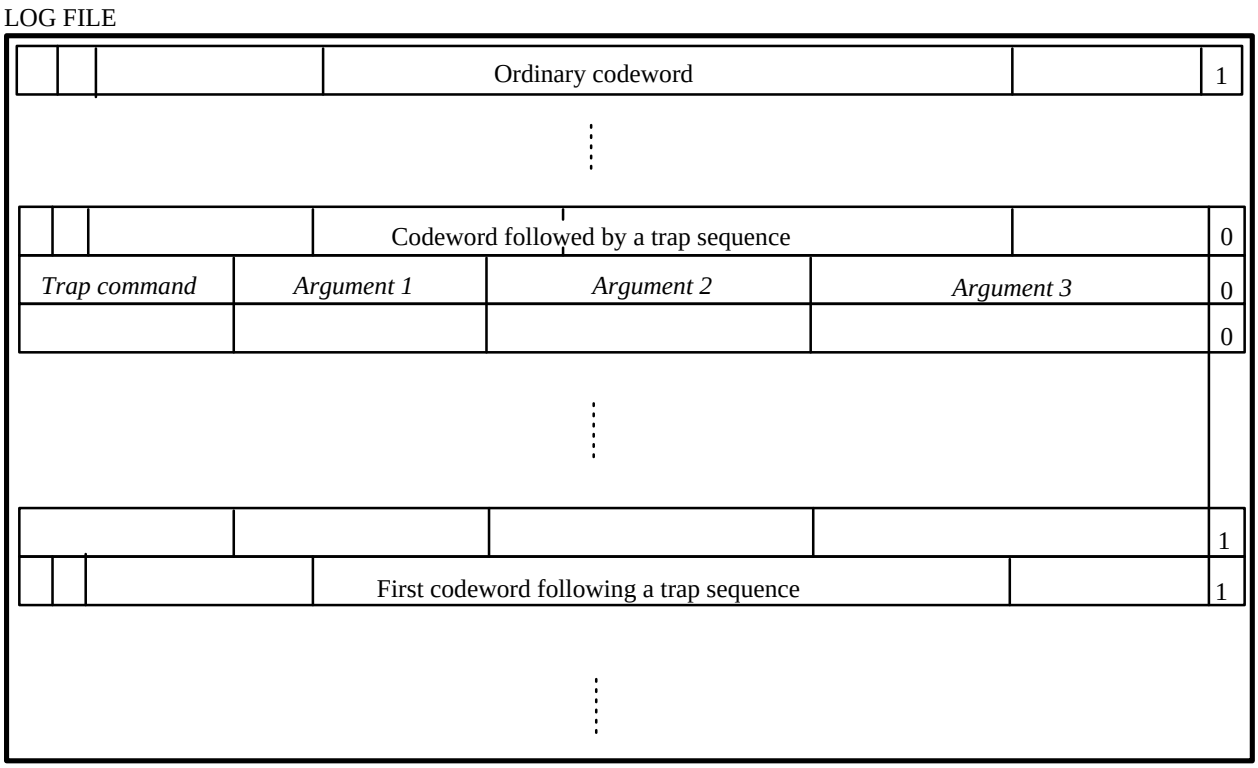


Figure 17. Format of DWG log files

---

and signal the local job-manager process (labelled **J** in Figure 16) to carry out the appropriate requests. If the trap command entails a change in the size of the foreground process population, then these routines also alter the process-population counts appropriately.

3) Off-line functions of DWG

These components come into play after a system-wide log has been measured but before it can be replayed. An initial log contains no traps. The trap-insertion routines (labelled **T** in Figure 16) insert traps at suitable offsets into each log file; each log is instrumented so that test processes can be started or stopped at precise moments relative to the start of the experiment.

Traps allow the dynamic doctoring routines (labelled **D** in Figure 16) to maintain a count of processes that exist during generation but did not exist at the time of measurement. The presence of such processes warrants a reduction in generated load; the amount of reduction depends upon the number of foreground processes. Since that number changes only when either a new process starts or an old process finishes, traps can trigger an update of process counts inside the generator precisely when the size of the competing process population changes. Thus, before measured data are given back to the kernel for replay, traps are inserted at the starting and (if known) stopping points of jobs. If the stopping point of a job is unknown, then the job manager notifies the kernel dynamically when such a job stops.

In determining where to insert traps in the pre-recorded system-wide logs, the trap-insertion routines use global *jobfiles*. These contain information about: (i) the site at which a job will be executed; (ii) starting time of the job; (iii) stopping time of the job, if known; (otherwise, upon noticing job termination, the job manager makes a system call that has exactly the same effect as a job-stop trap;) and (iv) the command and arguments needed for starting the job. The trap-insertion routines associate a unique global identifier with each job, and partition the global jobfiles into local jobfiles, which are passed on by the generating process to the individual job managers at each site.

D. Operation of DWG

The overall operation of DWG can be described in three phases: measurement, trap insertion, and generation. In the first phase, utilization levels of four key resources — CPU, memory, disk, and network

— are recorded at each clock interrupt. (In our implementation, there are 50 interrupts per second.) In the second phase, which is performed off-line, provisions are made for running test jobs on top of the recorded load. This is done by inserting *traps*, or data items in a special format, at appropriate points in the recorded data. During the generation phase, at each clock interrupt, DWG dynamically determines the amount of load to generate for each resource. It does so either by reading the instrumented log or by assessing the work left pending from previous interrupts. It then generates the requisite load by issuing synthetic resource-usage instructions. While reading the instrumented log, if DWG encounters a trap, then it decodes the trap and carries out the corresponding functions — such as updating process-population counts, and signalling the local job manager to perform appropriate job-control functions. When a test job started by the job manager finishes, the job manager records its completion time. Thus, background loads are replayed, test jobs introduced at precise instants, and their completion time measured under controlled loading conditions. The rest of this section describes the three phases of DWG's operation.

1) Workload measurement

The `measure()` routine of the kernel is periodically invoked by a clock interrupt. It samples the system's state and records (i) whether or not the CPU is busy; (ii) the number of free memory pages; (iii) the number of disk transfers since the previous interrupt; and (iv) the number of packets active on the network since the previous interrupt. Also recorded with each data item are the number of local and global processes generating the current load. Since the interrupts can occur several tens of times per second, the measured data can grow at a phenomenal rate. We keep such growth in check through efficient coding of information and periodic removal of data from the kernel by an external process. Similarly, during generation, information needs to be transferred into the kernel at the rate of a few hundred bytes per second. In order to keep the number of data transfers to a minimum, buffer pairs are allocated inside the kernel. Data transfer can proceed using the idle buffer while the kernel is busy reading/writing the other buffer.

Buffer sizes are chosen large enough that there are at most only a few transfers per minute.

2) Trap insertion

The trap-insertion routines essentially perform an event-driven simulation, using the offset into the logged data as simulated time. The events in jobfiles are sorted, first, by their starting times, and, second, by their stopping times (if known). The intuition is that stop events (if any) for the same job must necessarily follow the corresponding start events. The trap-insertion routines maintain two event-lists headed by, respectively, the next job to start, and the next job to stop (if its stopping time is non-zero). At every instant, lists of 'fired' events are computed. Every event results in at least one trap at the associated job's site of execution, and possibly others at sites generating disk and network traffic. This phase ends with the creation of *instrumented* logs, one per site; the traps inserted into these logs contain instructions for the kernel, which executes those instructions upon hitting these traps during the generation phase.

3) Generation

The processes generating the real workload are subject to scheduling. In UNIX and related operating systems [11], the scheduler maintains queues of ready-to-run processes, each queue corresponding to a priority level. It allocates resources to these processes in a round-robin fashion within each queue, and in order of priority among the queues. Processes having low priority can be preempted by the ones having higher priory. Priorties are recomputed periodically, thus causing processes to move between queues. In contrast, the generating process is not subject to scheduling; it behaves like a high-priority real process. If the generator were to always reproduce the measured load exactly, test jobs introduced into the system would encounter greater delays under a generated load than under the corresponding real workload. Therefore, synthetic workload amounts read from the log file need to be adjusted (or 'doctored') dynamically in order to have the same effect as real workloads.

Ideally, the generator would need to implement the entire queueing discipline in order to emulate the true behavior of the recorded load in the presence of an additional process. If the starting and stopping times of all test jobs were known ahead of time, such emulation could possibly be done off-line in a fashion similar to the trap-insertion process. However, stopping times of test jobs are usually unknown, and are, in fact, load-dependent. Therefore, an ideal generator would need to implement the process scheduler's queueing discipline *on-line*! That would be prohibitively expensive computationally. As a compromise, DWG makes certain simplifying assumptions about the scheduler's queueing discipline; these assumptions allow it to compute the altered behaviors dynamically without incurring too much computational overhead. This component of our generator, containing rules for altering the generated load in the presence of competing test jobs, constitutes its *doctoring rules*. These rules, and the assumptions on which they are based, are described below.

Rule 1: Reduction in generated load due to foreground processes. We assume, first, that all the processes (measured and new) have the same priority and that resources are allocated to them in a round-robin fashion; and, second, that the amounts of different resources consumed by a background process are reduced by the same proportion when it faces competition from one or more foreground processes. For example, a 15% reduction in CPU usage warrants a 15% reduction in memory occupancy, and so on. Also, we model CPU and memory as *constrained resources* whose usage levels are bounded by, respectively, 100% busy and 100% occupied. In practice, all resources have physical limits on utilization levels. The logical limits on disk and network appear to be infinite because requests for their use can be buffered in memory space; such buffers do not exist for the CPU and memory resources. Therefore, CPU and memory usage need to be explicitly spread out (see Figure 18) over time by buffering unfulfilled requests as pending work in the generator.

We assume that the load levels on private resources (CPU and memory) are affected only by the local process population, and those on shared resources (disk and network), by the system-wide process population. (For shared-memory systems, memory would also be treated like a shared resource.) The

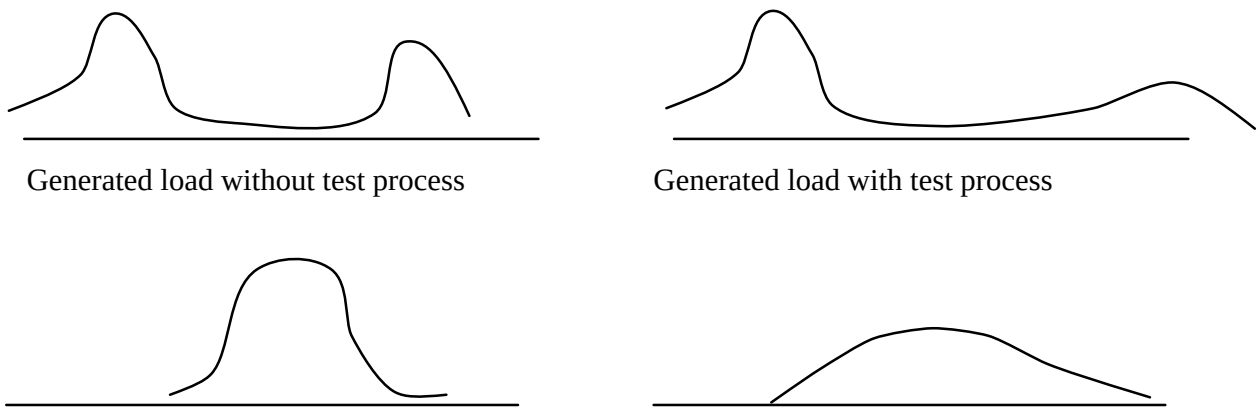


Figure 18. The need for *dynamic doctoring* of generated load

Consider a recorded load pattern shown to the top left of the figure, and a test job whose resource-utilization pattern is shown to the bottom left. The load patterns on the right illustrate the ‘‘smearing’’ effect: because of the foreground-background interaction, both the background and the foreground loads take longer to complete.

---

treatment of disk as a shared resource is specific to the client-server model of distributed file systems; in other models, disk may be treated as a private resource. Under these assumptions, the reduction in generated load can be computed as a ratio between the process-population sizes at the times of measurement and generation, respectively. Let *b* be the number of background processes, as recorded in the log being replayed; and *f*, the number of foreground processes, as maintained by the trap-handling routines of DWG. Then, the percentage of the needs can be satisfied at each clock interrupt is at most

$$p_1 = \frac{100 \cdot b}{b + f}. \tag{3a}$$

Further, the visible capacities of constrained resources (CPU and memory) are reduced to p_1 percent of their maximum values.

Rule 2: Principle of conservation of work. Plain reduction in load levels is insufficient for reproducing the true behavior of the measured process population. The processes constituting that population, when deprived of the full use of resources, would have taken longer to finish. Therefore, whenever

the generator fails to achieve the recordeded load levels, either due to competing processes or due to reduced resource capacities, it should carry over the difference between recorded and generated loads as pending work for subsequent cycles.

When a foreground process remains active for a few consecutive cycles of generation, the pressure of pending work may overwhelm the generator to such an extent that, instead of getting new work from the log, it will be forced to spend one or more cycles just getting rid of pending work. This happens when the pending loads on constrained resources (CPU and memory) exceed the corresponding resource capacities. DWG is said to be 'on hold' when in this mode. Holding allows us to slow down the replay of a recorded log; the rate of this slowdown is governed by the first rule. When on hold, the generator determines the maximally constrained resource. It then computes, with respect to that resource, the fraction of pending work that can be accommodated in the current generation interval. The same fraction of pending loads on other resources is then added to their respective load levels.

During generation, priority is given to pending work. For constrained resources, the rest of the (possibly reduced) capacity is allocated to (possibly reduced) background workload from the current interval. The combination of new and pending workloads for a resource may exceed its visible capacity; when that happens, the overflow is simply added to the pending work for future cycles.

E. Evaluation, Parameterization and Tuning of Generation Mechanisms

The generation mechanisms described in the previous section allow us to record and replay workloads generated by a population of test jobs, and replace the background process populations used in traditional experiments with synthetic workloads. However, we still need to assess how well the generated patterns approximate those caused by real workloads. In order to achieve high-quality generation, we need to first parameterize the generation mechanisms, then experiment with many different parameter sets, and finally select the best one for the data-collection phase of SMALL. The following subsections

address evaluation, parameterization, and tuning of genertion mechanisms.

1) Evaluation of generated loads

In order to evaluate DWG, we designed an experiment to compare the workloads generated by it against workloads generated by real processes. In this experiment, we use certain test jobs that have been instrumented to produce checkpoints when they reach certain preset points in their program codes. The measurement functionality and the system-call interface of DWG were expanded to include mechanisms for recording the most recent checkpoint of every active job, be it in the foreground or the background. An overview of our experiment is shown in Figure 19. Each experiment involves one pair of jobs, and proceeds as follows. First, the two jobs are executed in the foreground on top of an idle background load;

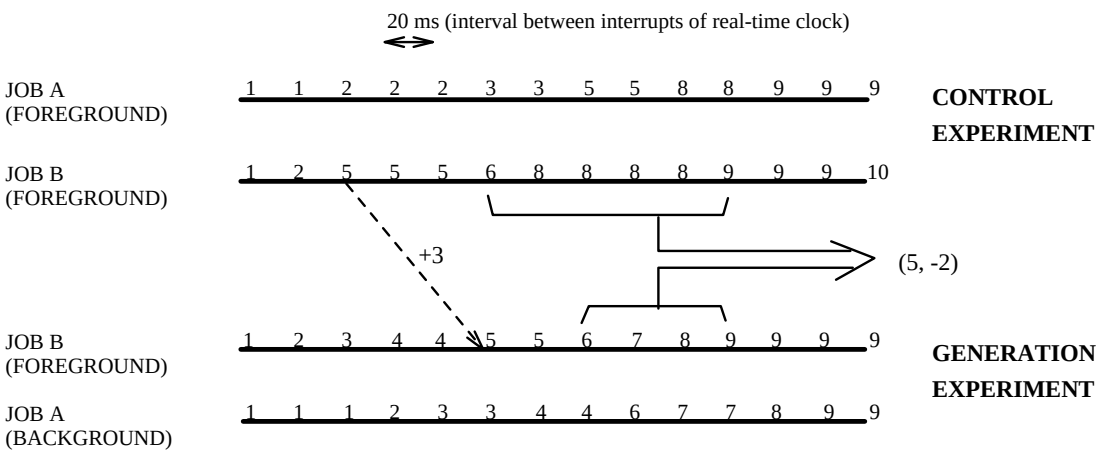


Figure 19. Measuring the accuracy of generated loads

Suppose that we perform a *control experiment* in which two jobs, A and B, are run in the foreground, and that the checkpoints of the two jobs occur as shown on the top two time lines. (Each of the uniformly spaced ticks on a time line shows the *most recent checkpoint* of the corresponding job.) Next, suppose that another experiment is performed with the load pattern for job A in the background and with job B in the foreground, and that the checkpoints of the two jobs occur as shown on the bottom two time lines. Errors can be computed for individual checkpoints: e.g., for checkpoint 5 of job B, the signed error is +3 (it occurs 3 ticks too late with respect to the control experiment). Likewise, errors can be computed for segments of each job: e.g., for job B, the segment that begins at checkpoint 6 and ends at checkpoint 9 takes 5 clock intervals to complete in the control experiment, but only 3 clock intervals in the generation experiment.

at each clock interrupt, the most recent checkpoint of each job is recorded in the resulting log. This is our *control experiment*. Next, only the first of these jobs is executed on top of an idle load and the resulting load pattern (including checkpoint timings), recorded. In the final step, the recorded log of the first job is replayed using DWG while the second job is run in the foreground. Once again, the resulting log (now including the checkpoint times of both the foreground and the background jobs) is recorded by the measurement routines. This is our *generation experiment*. As illustrated in Figure 19, the quality of generation can be assessed using the errors between offsets of corresponding checkpoints in the logs of control and generation experiments.

Suppose that checkpoints c_i and c_j of a job occur at times t_i and t_j, respectively, in the control experiment. Further suppose that the same checkpoints occur at times T_i and T_j in the generation experiment. Then, the error e_i of the ith checkpoint is given by

$$e_i = T_i - t_i; \tag{3b}$$

and the error for the job segment contained between the ith and jth checkpoints, by

$$e_{ij} = (T_j - T_i) - (t_j - t_i). \tag{3c}$$

Both signed and absolute values of errors were considered.

In our experiments, we used seven different benchmark jobs; these are described in Table III. They include three jobs of UNIX `sort` utility with different file sizes and memory requirements, two jobs of

Table III
Benchmark programs used in evaluation and tuning of DWG

Name	Description
Sort1	Sorting a small file by multiple fields with unlimited memory
Sort2	Sorting a large file by a single field with unlimited memory
Sort3	Sorting a small file by a single field with limited memory
UC1	Uncompressing a compressed file (#1)
UC2	Uncompressing a compressed file (#2)
W.TF	The Perfect Club benchmark FLO52Q — solving Euler equations
W.TI	The Perfect Club benchmark TRFD — two-electron integral transformation

the UNIX `uncompress` program, and two Perfect Club benchmarks [22]. Since each experiment involves two jobs, we made sure that the selected combinations represented a variety of foreground-background combinations. We allotted fifteen minutes for each experiment. The data for the benchmark programs were chosen such that idle completion times would be close to five minutes on the slowest site. Each benchmark was instrumented to produce checkpoints at certain fixed points in their execution. For this purpose, additional functionality was added to the kernel-based components of DWG; at each checkpoint, the benchmark programs incremented a checkpoint identifier and notified DWG of the checkpoint's occurrence by making a system call. Each benchmark program was instrumented to produce approximately 200 checkpoints at almost regular intervals during its execution. (While we inserted checkpoints manually, certain automated utilities are now available that can be suitably modified for automatic insertion of checkpoints [109].) Each benchmark program was also assigned a unique job identifier so that its checkpoints could be distinguished from those of another (concurrently active) program.

2) Parameterization of generation mechanisms

DWG's behavior is significantly affected by two components: the formula for computing p_1 in the first doctoring rule (Equation 3a); and *c*, the percentage of clock interval consumed by the CPU generator when the recorded log shows that the CPU was busy in that interval. We can parameterize the doctoring rule as:

$$p_1 = \frac{100 \cdot b}{b + f \cdot x},$$

where the parameter *x* controls the relative importance of processes generating the foreground load with respect to the processes that generated the recorded load that is now being replayed in the background. Table IV shows nine different parameter sets for DWG. In this table, both *x* and *c* are expressed as percentages. For example, $x = 100$ means that foreground and background processes are given equal weight; and $x = 110$, that foreground processes are given more weight than background processes. The latter may be appropriate when the count of background processes includes some inactive (operating system)

Table IV
Parameter sets for doctoring rules

Set No.	x	c
1	100	80
2	95	80
3	110	90
4	100	90
5	95	90
6	110	80
7	110	70
8	100	70
9	95	70

processes. Continuing the example, $c = 70$ means that CPU-load generation can use up to 70% of the clock interval.

Other parameter sets are possible. However, since we do not have automated mechanisms for parameter selection at the current time, we have chosen to limit our attention to the parameter sets described in Table IV. Our goal is to first evaluate each of these parameter sets using the experiments described in the previous section, and then select the best one among them for the data-collection phase of SMALL.

3) Results of comparing different parameter sets

Each of the parameter sets described in Section III.E.2 was evaluated using the method described in Section III.E.1. We performed seventeen experiments involving different combinations of foreground and background jobs. For each experiment, and for each job, we computed the following statistics (also see Equation 3b):

$$E_s = \sum_{i=1}^{c} \frac{(e_i - e_{i-1})}{c} \tag{3d}$$

and

$$E_u = \sum_{i=1}^{c} \frac{|e_i - e_{i-1}|}{c}, \qquad \textbf{(3e)}$$

where c is the total number of checkpoints of the foreground job. E_s is the average signed delay incurred by a checkpoint in the generation experiment, relative to the control experiment. Positive values of E_s indicate that, on the average, checkpoints in generation experiment are delayed with respect to their 'true' timings from the control experiment, and negative values, that they occur too soon in the generation experiment. Since positive and negative errors can sometimes cancel out, it is necessary to consider the absolute delays as well; E_u is the average (unsigned) error between the times at which the same checkpoint occurs in the generation and control experiments. We refer to E_s and E_u as, respectively, the signed and the unsigned errors due to generation. These statistics were computed for both jobs of each evaluation experiment. Tables V and VI show, respectively, the signed and unsigned errors for seventeen different evaluation experiments using the benchmarks described in the previous subsection. Each column presents the errors for one parameter set from Table IV. Two values are reported for each experiment: the top value is the error for the background job, and the bottom value, the corresponding error for the foreground job. Errors are in units of clock intervals, each of which spans 1/50 seconds. Parameter sets 3, 4, and 5 have unacceptable performance on experiments 9, 11, and 12, especially in terms of absolute errors (Table VI). Parameter sets 1 and 7 appear to have no unacceptably large errors, but there is no clear winner.

Because of the way benchmarks are instrumented, each pair of checkpoints describes a program segment. With preemptive load-balancing policies (Chapter I), both complete jobs and program segments can be scheduled. The size of the scheduled segments depends upon the preemption interval of the policy: the longer the preemption interval, the longer is the maximum size of a scheduled program segment. Since it is expected that the generator will incur larger errors for longer jobs, we wish to study the relationship between errors and preemption intervals. To this end, let us consider *all pairs of checkpoints* in the foreground job of each experiment, and for the program segment between the *i*th and *j*th checkpoints, let us create a data-pair $<l_{ij}, e_{ij}>$, where $l_{ij} = (t_j - t_i)$ represents the true running time of that segment, and

Table V
Signed errors due to generation for different parameter sets

Parameter Set # ->	1	2	3	4	5	6	7	8	9
Experiment #1 **BG=Sort1 FG=UC1**	-14.7 -8.9	-15.6 -6.5	-4.1 9.2	-14.7 4.3	-15.3 3.9	-7.0 -4.9	-5.5 -5.8	-14.7 -9.1	-15.7 -10.1
Experiment #2 **BG=Sort1 FG=W.TF**	-16.6 -13.8	-14.9 -11.5	2.1 7.1	-6.1 6.6	-6.8 6.6	-12.9 -12.9	-13.1 -14.1	-16.6 -13.8	-17.1 -13.9
Experiment #3 **BG=Sort1 FG=Sort3**	-11.8 -2.0	-10.1 0.5	6.2 13.4	-4.4 12.5	-5.1 12.2	-8.0 -1.4	-8.2 -2.3	-11.9 -2.1	-12.3 -0.6
Experiment #4 **BG=Sort2 FG=UC2**	-7.8 -0.2	-8.5 9.5	-0.2 27.4	-7.9 20.6	-8.4 20.4	-5.3 -1.4	-5.2 -2.7	-8.3 -2.8	-8.9 -19.7
Experiment #5 **BG=Sort2 FG=W.TI**	2.4 -3.2	3.2 5.5	11.4 15.8	3.9 16.5	3.5 16.7	5.2 -2.0	5.6 -2.0	2.1 -3.3	2.3 -2.2
Experiment #6 **BG=Sort2 FG=Sort3**	-7.1 -1.3	-5.3 2.2	3.3 14.5	-4.0 11.4	-4.6 11.1	-4.7 -1.4	-4.7 -2.0	-7.1 -1.2	-7.7 -1.5
Experiment #7 **BG=Sort3 FG=UC1**	-4.0 -1.0	-4.4 1.2	1.4 14.8	-4.0 10.0	-4.2 10.3	-0.1 2.0	0.6 1.9	-4.0 1.1	-4.5 -1.5
Experiment #8 **BG=Sort3 FG=W.TF**	1.8 -4.8	1.4 -1.4	7.1 14.8	1.8 18.5	1.5 16.1	3.5 -5.6	3.9 -5.3	1.6 -5.9	1.3 -5.8
Experiment #9 **BG=Sort3 FG=Sort1**	-3.6 -2.0	-4.0 2.4	1.7 20.7	-3.5 15.6	-3.7 18.8	0.7 6.9	1.1 6.0	-3.6 0.2	-2.2 2.1
Experiment #10 **BG=UC2 FG=W.TF**	6.9 1.3	4.6 10.9	1.8 23.0	6.9 23.2	5.2 22.3	12.5 0.6	12.6 -0.2	6.9 1.7	4.1 -0.0
Experiment #11 **BG=UC2 FG=Sort1**	-7.5 2.3	-9.8 10.3	4.0 24.3	-7.4 20.7	-9.1 19.4	1.1 11.1	2.7 8.0	-7.4 6.0	-9.8 3.9
Experiment #12 **BG=UC2 FG=Sort2**	-16.4 -0.8	-18.0 5.0	-4.4 14.8	-16.9 12.1	-18.2 11.6	-8.6 3.6	-5.8 2.8	-13.8 1.3	-19.7 2.1
Experiment #13 **BG=UC2 FG=Sort3**	-0.2 5.1	-2.4 7.8	11.1 15.1	-0.1 9.1	-1.8 11.9	8.2 5.1	9.6 4.2	-0.1 4.7	-2.9 4.0
Experiment #14 **BG=W.TI FG=UC1**	-10.8 -7.2	-10.8 -6.7	-4.4 10.1	-10.8 5.4	-10.8 5.2	-6.2 -3.6	-5.3 -3.5	-10.8 -7.3	-10.8 -6.7
Experiment #15 **BG=W.TI FG=Sort1**	-8.9 -1.0	-8.8 -1.5	-2.5 21.1	-8.9 16.2	-9.0 17.8	-4.4 4.8	-3.3 5.0	-8.8 2.0	9.0 0.8
Experiment #16 **BG=W.TI FG=Sort2**	-10.1 2.2	-10.1 3.3	-3.3 18.1	-10.1 14.3	-9.5 14.1	-3.0 5.0	-2.5 4.1	-7.2 3.7	-6.8 4.7
Experiment #17 **BG=W.TI FG=Sort3**	-6.6 -0.8	-6.6 -0.9	-0.2 13.6	-6.6 12.7	-6.6 12.3	-2.1 -0.3	-1.1 -0.6	-6.6 -0.7	-6.6 -0.6

Table VI
Unsigned errors due to generation for different parameter sets

Parameter Set # ->	1	2	3	4	5	6	7	8	9
Experiment #1 **BG=Sort1 FG=UC1**	15.9 24.0	16.6 28.0	11.9 41.8	15.9 43.9	16.5 44.1	12.0 23.6	11.8 22.2	15.9 23.8	16.8 24.3
Experiment #2 **BG=Sort1 FG=W.TF**	21.8 16.0	19.6 15.4	23.0 15.0	21.5 14.5	21.2 14.2	25.1 15.0	26.1 16.1	21.7 15.8	21.2 16.0
Experiment #3 **BG=Sort1 FG=Sort3**	21.1 12.9	23.8 14.4	27.1 17.3	28.2 20.9	28.2 20.2	16.3 13.4	15.0 13.1	21.0 13.3	21.9 13.3
Experiment #4 **BG=Sort2 FG=UC2**	17.9 31.3	18.1 40.0	16.5 51.5	18.0 56.7	18.1 57.0	13.1 27.9	12.6 27.4	17.4 28.3	18.4 29.5
Experiment #5 **BG=Sort2 FG=W.TI**	9.2 8.8	7.7 15.2	13.3 16.8	8.2 17.2	7.9 18.7	13.7 8.0	14.5 9.9	9.3 9.7	8.1 7.5
Experiment #6 **BG=Sort2 FG=Sort3**	14.7 16.0	17.3 17.3	18.0 24.0	17.6 26.4	17.8 26.9	12.7 15.9	13.4 15.3	14.7 15.7	15.4 15.8
Experiment #7 **BG=Sort3 FG=UC1**	7.4 21.5	7.5 25.5	6.8 43.6	7.4 45.6	7.5 46.5	6.7 19.0	6.6 17.9	7.4 21.2	7.5 20.8
Experiment #8 **BG=Sort3 FG=W.TF**	6.6 10.7	6.5 17.3	9.5 17.1	6.6 21.4	6.6 19.2	9.7 11.5	10.3 10.8	6.8 11.5	6.5 11.3
Experiment #9 **BG=Sort3 FG=Sort1**	9.5 26.8	9.5 34.2	10.2 62.0	9.6 57.6	9.6 70.4	10.3 32.8	10.1 30.5	9.5 30.5	10.7 32.1
Experiment #10 **BG=UC2 FG=W.TF**	10.5 17.0	9.3 21.8	19.4 26.4	10.6 28.2	9.6 29.2	18.3 16.9	20.2 16.9	10.6 17.2	9.0 16.8
Experiment #11 **BG=UC2 FG=Sort1**	16.3 40.5	16.6 57.9	18.1 71.2	16.2 80.7	16.6 79.7	17.3 47.0	17.6 40.2	16.3 48.3	16.9 45.9
Experiment #12 **BG=UC2 FG=Sort2**	26.4 34.6	28.0 43.3	28.6 54.5	26.4 61.6	27.0 62.5	26.8 36.7	27.5 32.8	28.7 36.81	27.0 39.1
Experiment #13 **BG=UC2 FG=Sort3**	9.7 18.4	9.9 24.3	14.0 29.1	9.8 22.0	9.8 31.5	12.2 16.7	13.1 15.9	9.7 17.9	10.0 18.5
Experiment #14 **BG=W.TI FG=UC1**	11.3 16.8	11.3 17.2	5.1 39.1	11.3 40.9	11.3 41.6	6.8 15.2	5.9 13.9	11.3 16.3	11.3 17.1
Experiment #15 **BG=W.TI FG=Sort1**	9.8 14.2	9.7 15.3	4.4 45.2	9.8 49.7	9.9 53.8	5.9 17.4	5.0 15.3	9.7 20.5	9.9 18.5
Experiment #16 **BG=W.TI FG=Sort2**	10.5 11.0	10.5 12.6	4.7 32.0	10.5 34.1	9.9 33.2	4.4 11.7	4.9 9.3	7.6 12.1	7.2 12.6
Experiment #17 **BG=W.TI FG=Sort3**	7.0 7.3	7.0 7.4	3.0 14.3	7.0 16.7	7.0 15.5	3.2 7.5	3.1 7.3	6.9 7.3	7.0 7.1

e_{ij} represents the net delay in the execution of that segment due to generation (Equation 3c).

Considering that each job produces close to 200 checkpoints during its execution, the information about all pairs of checkpoints is too voluminous to be meaningfully visualized; therefore, we group different data pairs into *frequency classes*. Let us consider a resolution of 12 seconds along the *l*-axis, and 4 seconds along the *e*-axis. For notational convenience, let us refer to each frequency class as <E,L>, where both E and L denote ranges: E represents a 4-second range along the *e*-axis, and L, a 12-second range along the *l*-axis. Let f_E^L denote the number of data points whose *l* and *e* values lie within these ranges; i.e.,

$$f_E^L = |\{<l,e> \text{ such that } l \in L \text{ and } e \in E\}|. \tag{3f}$$

Further, let R(E) denote a super-range formed by including all the ranges to the left of (and including) E on the *e*-axis. The, we can compute the cumulative probability of error:

$$p_E^L = \frac{f_{R(E)}^L}{\sum_E f_E^L}. \tag{3g}$$

p_E^L represents the probability that jobs whose length lies inside the range *L* will incur an error that lies either inside or to the left of the range *E*.

Figure 20 shows contour plots of p_E^L in the *e*-*l* space. Each contour connects points with equal cumulative probability of error. Since the line of zero error is a vertical line right down the middle of the contour plot, we can see why parameter sets 3, 4, and 5 are so bad: almost all the errors are large for any interval of reasonable size. This figure also shows that most of the contours for parameter set 7 lie close to the line of zero error; this parameter set was selected for use in the data-collection phase of our experiments.

A detailed view of the contour plot for parameter set 7 is shown in Figure 21. The Y-axis represents length of possible preemption interval in seconds; the X-axis represents possible delays (also in seconds). Parameter set 7 was selected from the nine sets shown in Figure 20 because its contours are clustered tightly around the line of zero error. The nineteen contours divide the space into twenty regions of equal

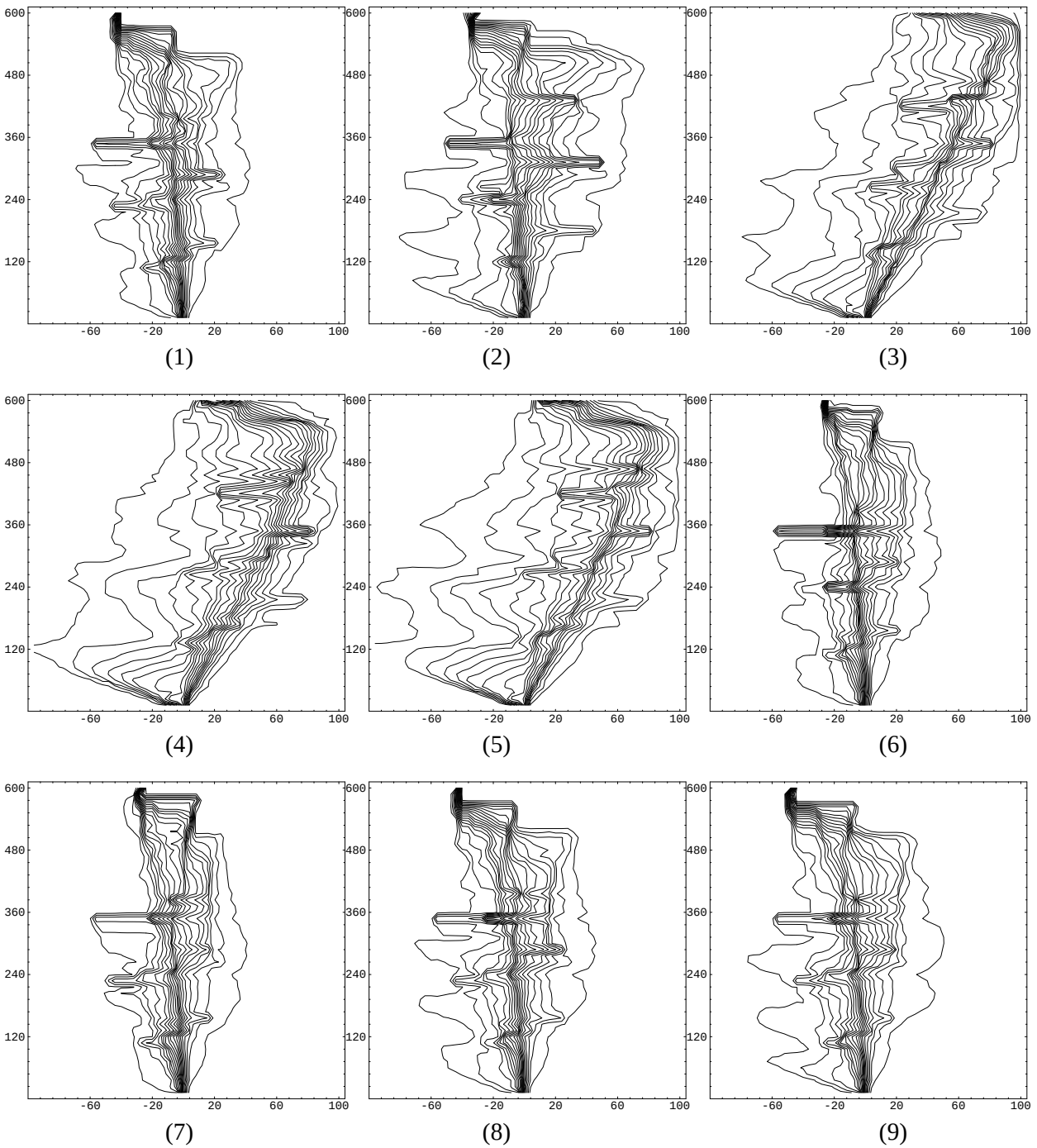


Figure 20. Contour plots of cumulative probability of error with nine different parameter sets

Each contour connects points with equal cumulative probability of error. The Y-axis represents possible lengths (in seconds) of the preemption interval; and the X-axis, errors due to generation (also in seconds). There are twenty regions of equal probability: the leftmost region corresponds to cumulative probabilities of error between 0 and 0.05; the one to its right, between 0.05 and 0.1; and the rightmost one, between 0.95 and 1.

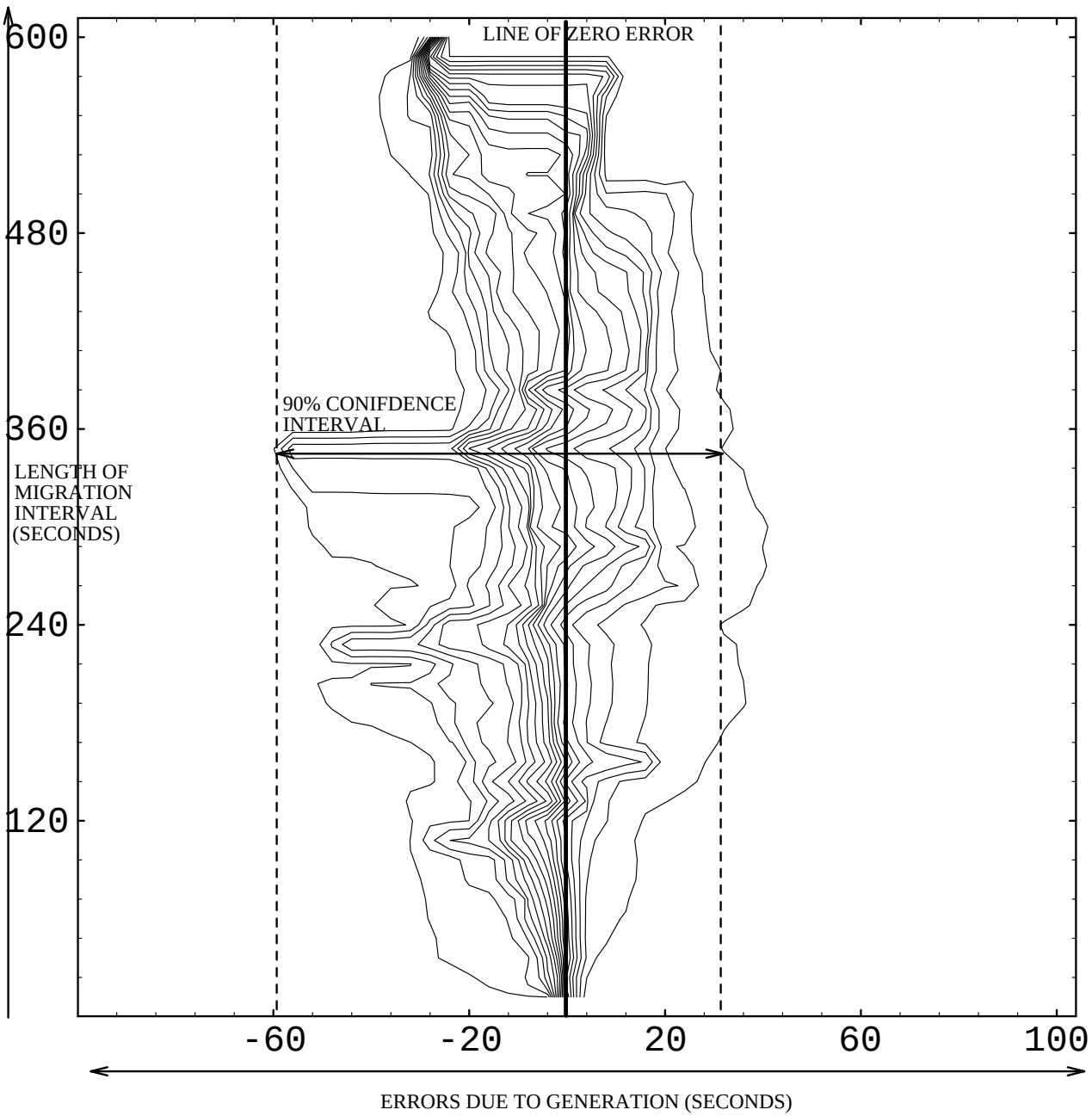


Figure 21. Contour plot of cumulative probability of error for the selected parameter set (7)

probability; thus, if we omit the leftmost and rightmost regions, then we can say (with 90% confidence) that all the errors for a preemption interval of 6 minutes (360 seconds) will lie in the range [-60,+32] seconds. Further, we believe that the somewhat larger errors along the middle of the Y-axis are due to repeated inclusion of a few small intervals having large errors. Even so, Figure 21 shows that there is

room for improvement in both the parameterization and parameter-selection of DWG. Perhaps, better modeling of the process-level scheduling algorithm and better parameterization of the generation process will bring DWG even closer to perfect generation.

We performed additional experiments in which no foreground jobs were introduced, in order to evaluate DWG's behavior in the absence of foreground jobs. In all such experiments, DWG was able to reproduce the checkpoint timings exactly, with zero error.

4) Comparison of recorded and reproduced loads

For the selected parameter set (7), we compared the load patterns of individual resources from the control experiment against the corresponding load patterns from the generation experiment. Figures 22 and 23 show, respectively, the comparison for experiments #15 and #17 (also see Tables V and VI). Visually, we can confirm that the generator reproduces the utilization patterns rather well when the doctoring rules use the selected parameter set.

F. Chapter Summary

Physical-level synthetic workload generation is vital for load-balancing experiments. We have described DWG, a workload-generation tool that accurately replays measured workloads in the presence of competing tasks. To do so, it needs to compensate for foreground-background interactions using dynamic-doctoring rules. Perfect doctoring of generated loads is precluded by the impracticality of exactly modelling process-level scheduling policies. DWG performs reasonably even though it maintains only a few scalar items of information about pending work. Completion-time measurements obtained using DWG are used for learning new load indices (Chapter IV) as well as new load-balancing policies (Chapter VI).

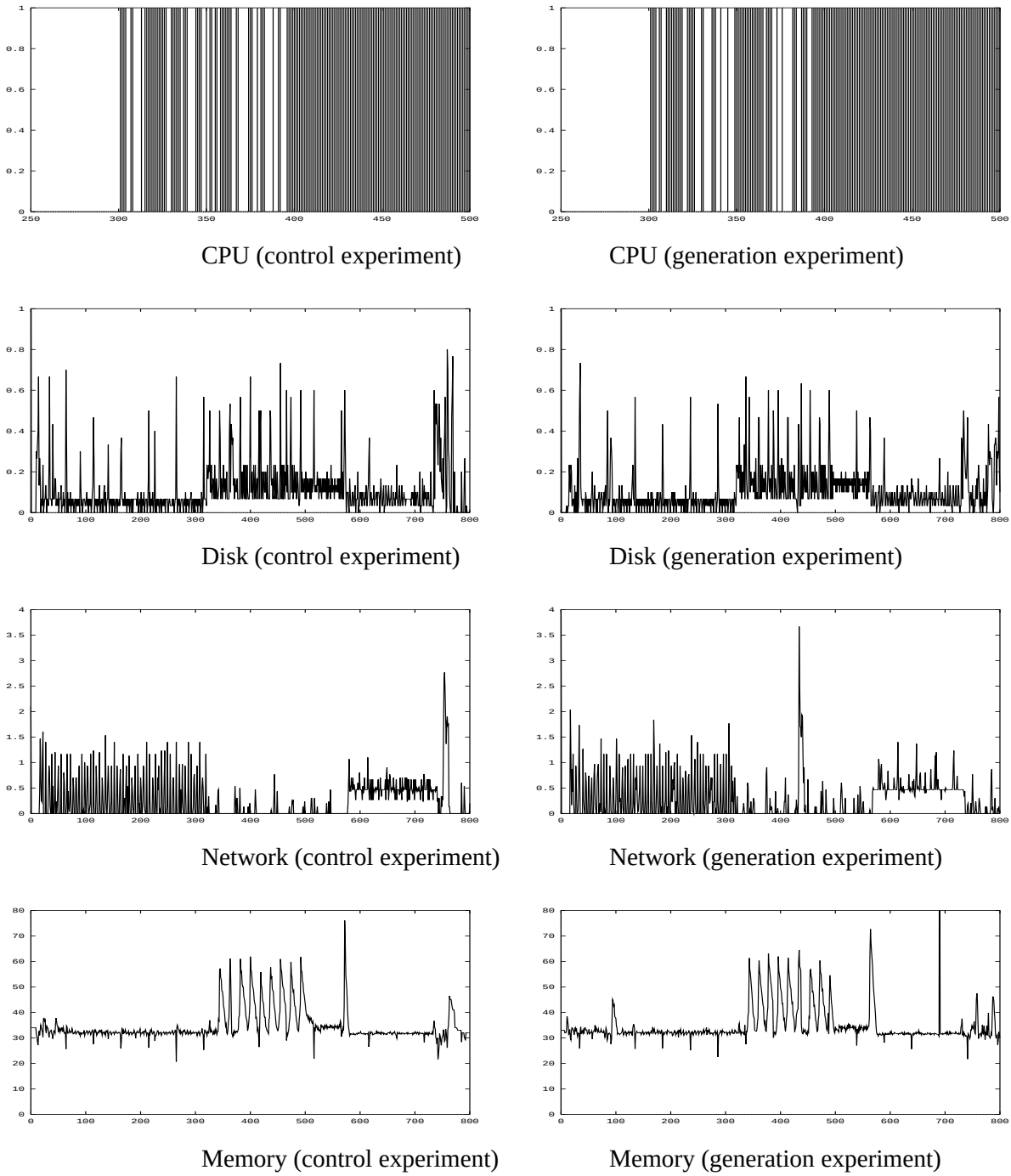


Figure 22. Comparison of true and generated resource-utilization patterns (Expt. #17)

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 plots 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.

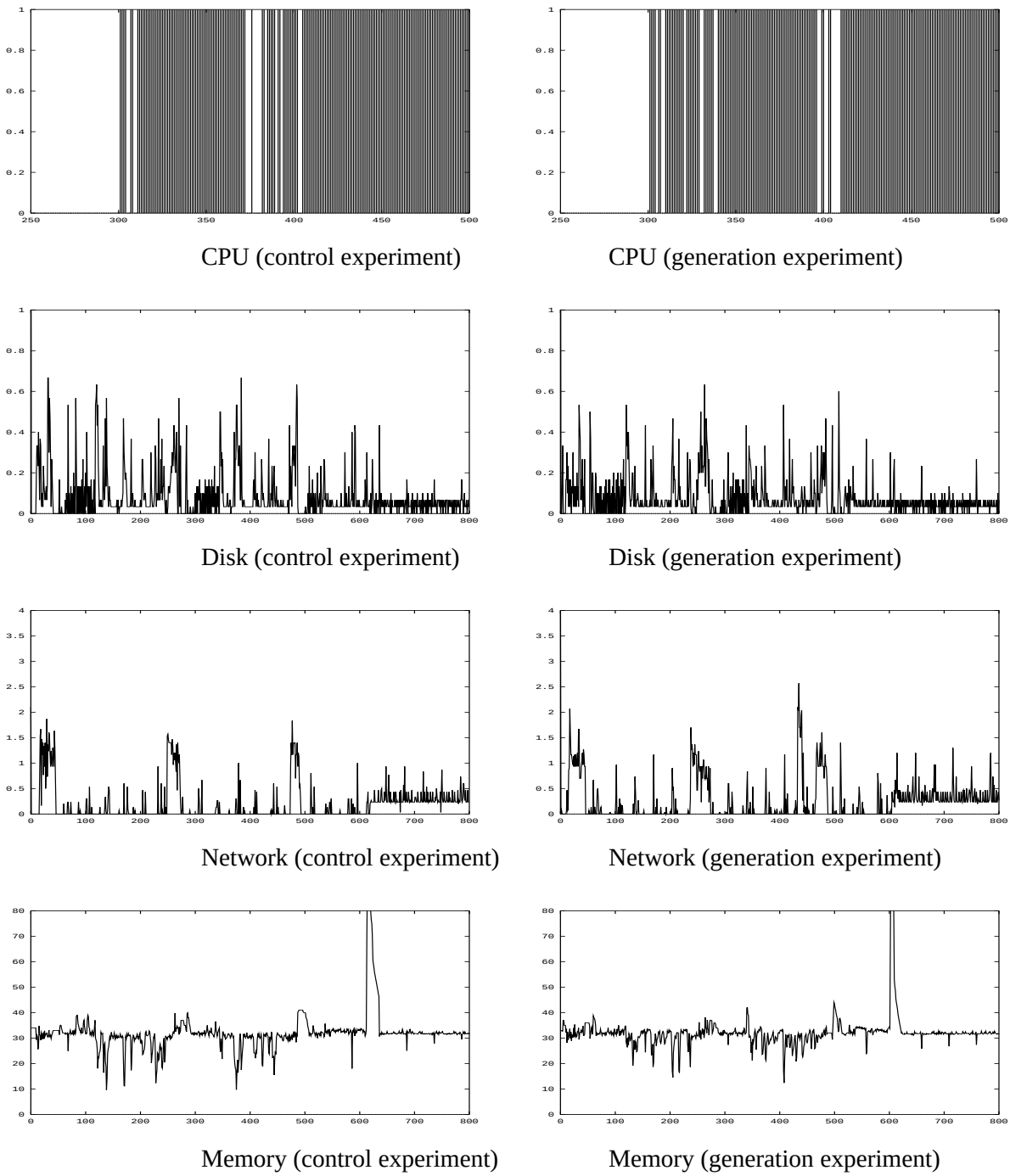


Figure 23. Comparison of true and generated resource-utilization patterns (Expt. #15)

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 plots 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.

CHAPTER IV.

COMPARATOR NEURAL NETWORKS FOR LEARNING LOAD INDICES

O! this learning, what a thing it is.

- William Shakespeare

This chapter presents the details of our approach to off-line learning of load indices. In Section A, we formally state our learning task: learning to compare functions of multivariate time series. In Section B, we present the comparator-network architecture and its associated learning algorithms. Section C describes how the raw measurements collected using DWG are smoothed using digital lowpass filters, and short-term trends in resource-utilization extracted using linear and exponential extrapolation. Section D presents the learning curves for the comparator networks; and Section E, methods for the statistical assessment of new load indices. Section F concludes this chapter, and illustrates how the index functions learnt by our method convert raw measurements collected by DWG into load indices suitable for use by load-balancing policies.

A. Design Goals

Overheads of communication preclude transmission of large quantities of status information among sites. This limitation applies to both the number of items transmitted and the rate of transmission. Raw utilization levels of resources are highly dynamic vector quantities, whose communication would incur too much overhead; they are, therefore, not suitable for communicating load information among sites. Sites need this information in order to compare alternative destinations (or loading conditions) for an incoming job. Since different loading conditions lead to different completion times, and since our objective in load balancing is to find the destination that leads to minimum completion time for an incoming job, the comparison of different loading situations should be consistent with the comparison of their com-

pletion times for the incoming job.

Completion times of jobs are unknown at their time of arrival. Given our lack of information about the resource-intensities of different jobs, and unavailability of models relating resource-utilization patterns to job-completion times, we have two options: (i) we can attempt to model various characteristics of tasks and resources, and then analytically compute completion times using these models; or (ii) we can directly estimate the completion times of tasks using functions statistically estimated from data. The first approach is complicated by the data-dependence and program-dependence of task models; the information necessary for constructing such models cannot be obtained without some prior knowledge of the relationships between the input data and program behavior.

The data for the second approach, direct estimation of completion times, can be easily obtained using DWG, our synthetic workload generator. These data contain information about (i) the utilization patterns of different resources prior to the arrival of a test job; and (ii) completion times of that job at each of the alternative destinations. As far as tasks are concerned, the second approach suffers from the same lack of information as the first approach. However, for resources, the problem with the second approach is the overabundance of information, rather than the lack of it. Ignorance of task models may be overcome by estimating load-index functions that apply reasonably well to data collected for many different jobs representing different kinds and different levels of resource intensities. Overabundant information can be handled by load-index functions that make predictions using only a small number of resource-utilization values simultaneously.

The information content of utilization patterns along the time dimension can be reduced by using smoothing techniques, such as moving averages [25] and lowpass filtering [78]. Trends can be extracted from these smoothed patterns and used for estimating the average utilization of a resource in the near future. Further reduction in information can be achieved by combining the information from multiple resources into a scalar load index.

Utilization levels are available for only the resources local to a computer — CPU, memory, and (only at diskful sites) disk — and the (shared) network resource. Therefore, the scalar load indices should not depend on the utilization of resources at remote sites (such as remote disks).

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

Let $F_A^W(k)$ denote the value site A's load-index function, where k identifies the background load pattern and W denotes the current values of the *weights* (or parameters) of F_A. k refers to a vector of values derived from the recent behavior of loads on different locally accessible resources. Whenever it is obvious, we will omit the superscript W from our equations. In this context, the objective of load-index learning (Figure 9) can be formulated as:

$$F_A(k.j_A) - F_B(k.j_B) = \frac{C(j,k,A) - C(j,k,B)}{C(j,\,0,0)}, \tag{4a}$$

for all jobs j, for all background load patterns k, and for all pairs (A,B) of sites. The right-hand side of this equation denotes the difference of relative completion times. As to why we do not, instead, predict the absolute difference in completion times, $C(j,k,A) - C(j,k,B)$, the reason is the lack of information about the resource intensities of job j; without knowing these, there is no way to estimate (in absolute terms) how long a job will run under any loading conditions.

It is enlightening to examine the conditions under which the value of the expression on the right-hand side of Equation (4a) will not depend on the job j being scheduled. Let us consider the expression $\frac{C(j,k,A)}{C(j,\,0,A)}$, which is the ratio of the completion time under load to the idle completion time for the same job and the same site. This ratio is known as *stretch factor* [58], and its denominator, as the *service time*

of job j at site A; the latter depends only upon the job and the site, but not on the load. Further, the completion time $C(j,k,A)$ can be expressed as a sum of service time and waiting time:

$$C(j,k,A) = C(j,0,A) + W(j,k,A), \tag{4b}$$

where $W(j,k,A)$ is the time spent by job j waiting for resources. When the operating system employs round-robin scheduling policies at the process level, the waiting time $W(j,k,A)$ of job j grows linearly with its service time; that is, the longer the job, the more time it spends waiting for resources. The rate of (linear) growth depends upon the load: when the load is high, even short jobs can take a long time to complete. Therefore, we can rewrite Equation (4b) as:

$$C(j,k,A) = C(j,0,A)[1 + G(k,A)], \tag{4c}$$

where $G(k,A)$ is a site-specific function that increases monotonically with load.

Under the model of completion times described above, the stretch factor $\frac{C(j,k,A)}{C(j,0,A)}$ depends only upon the load k and site A, but not on the job j. 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

$$C(j,0,A) = K(A)C(j,0,0), \tag{4d}$$

where $K(A)$ is some site-specific constant; that is, the service times of a job at different sites are constant multiples of each other. Now, we can rewrite Equation (4a) as

$$F_A(k.j_A) - F_B(k.j_B) = K(A)[1 + G(k,A)] - K(B)[1 + G(k,B)]. \tag{4e}$$

Notice that the right hand side does not depend upon the job j being scheduled. Therefore, under the assumptions described above, the objectives described by Equation (4a) are achievable without prior knowledge of the job being scheduled. This should not come as a surprise since stretch factors have been used previously as load indices in computer systems [60].

The two assumptions required for the just-proved independence between the characteristics of a job j and the right-hand side of equation (4a) are: (i) waiting time grows linearly with service time; and (ii) different sites come from the same architecture family. The first of these assumptions is a reasonable

requirement of any fair scheduling policy, and is approximately true of UNIX-related operating systems that employ variants of round-robin scheduling. The second assumption is generally true of workstation-based computing environments, because even workstations from different vendors often employ the same microprocessor architecture.

Given the objectives described in Equation (4a), we have two options. We can either solve the learning-to-predict problem whose objectives can be defined as

$$F_A(k.j_A) = K(A)[1 + G(k,A)], \tag{4f}$$

or we can solve the learning-to-compare problem whose objectives are defined by Equation (4a). As we argue in Chapter I, we prefer the latter problem because it is more naturally related to our overall objective of *comparing* alternative destinations for an incoming job.

How well the objective described by Equation (4a) can be satisfied depends upon several factors. Let us denote by **W** the space of all parameter values for functions F_s, and let **F** = $\{F^W: W \in \mathbf{W}\}$. The load-index function we shall learn must belong to the family of functions **F**. Even when the best function from this family is used, the error between the left and right sides of Equation (4a) may not be zero for all data points; such errors contribute to the *approximation error* of a learning approach. Approximation errors are defined independently of the learning algorithm used for adjusting the weights *W*.

A given training algorithm may not be capable of finding the optimal weights *W* in reasonable amount of time. The error incurred due to suboptimality of weights *W* is called *estimation error*; it depends upon the particular training algorithm used.

We need an approach that reduces both the approximation error as well as the estimation error. More importantly, we need to examine how exactly we compute errors. For example, we could minimize the sum of errors (or the sum of squared errors) between the left and right sides of Equation (4e). Notice, however, that the problem we are solving is one of obtaining *accurate comparison in most situations*; we would like both the left and right sides of Equation (4a) to have the same sign, without worrying about

the magnitude of the difference. Our goal, therefore, is to learn load-index functions that, when compared across sites, correctly predict (for most test-cases) the site with the lower relative completion time. Since our point of reference (completion time on idle fileserver) is fixed for any given job, the site that has lower relative completion time will also have the lower absolute completion time.

We have a database of test-cases, each of which contains loading information and completion-time measurements *for the same job* on a pair of sites. We used a total of ten test jobs: the seven jobs shown in Table III, and three additional benchmarks from the Perfect Club suite [22]: (i) QCD (quantum chromo-dynamics); (ii) TRACK (target tracking); and (iii) DYFESM (structure analysis). Each benchmark 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. The period from the beginning of an instrumented benchmark to one of its checkpoints was treated like an independent job. The basic idea here is to simulate short jobs that might result if the test job were preempted and rescheduled at some point during its execution. Even though this technique creates a larger set of test-cases, it also creates an approximation effect since two jobs corresponding to different checkpoints of the same benchmark program may have similar resource requirements. In order to avoid choosing checkpoints too close to each other, we created our final database by selecting the first checkpoint of each benchmark and thereafter, the next checkpoint that took at least 5% longer than the one selected before it. In addition, the final checkpoint of each job was always included. We ended up with a total of only 58 jobs, with about five checkpoints per program being selected; each of these jobs was run at each of the four sites and under about 24 different load patterns, for a total of 5,324 test-cases.

Of the 24 background load patterns used in our experiments, 20 were created using the procedure described in Section II.B.1, by running jobfiles created randomly from the pool of 10 test-jobs described above. The remaining 4 load patterns were designed to create suprises for the load-index function. We started with an actual, heavy, system-wide workload and patched together pieces of it and the idle workload pattern P(0), such that the resulting load patterns would frequently contain loading conditions just

the opposite of those prior to the job's arrival. Since the load index has access only to the loading condition before a job's arrival, its predictions would falter under these 'surprising' loading conditions. Since real workloads often have unpredictable changes in workload, whose likelihood increases with the length of the load pattern, we packed more surprises near the end of the load pattern. We expect that, because of these load patterns, the performance (speed-up over local execution) attained by a task at the site with the least load-index function will tend to drop with the length of the task, approaching the performance of random choice for very long tasks.

We partitioned the final database into a *training set* and a *test set*. We used the training set to tune the weights of the load-index functions, and the test set to evaluate the trained functions. Suppose that the test set $\mathbf{T} = \{t(j,k,A,B) \mid j \in \mathbf{J};\ k \in \mathbf{K};\ A, B \in \mathbf{S};\ A \neq B\}$, where **J** is the set of test jobs, **K** is the set of load patterns, and **S** is the set of sites. Each test-case $t(j,k,A,B)$ is a 5-tuple <k.j_A, C(j,k,A), k.j_B, C(j,k,B), C(j,0,0)>. The *objective function for load-index learning* can be formally defined as shown in Figure 24. Our goal is, therefore, to learn those load-index functions from data that maximize the number (and, thereby, the percentage) of correct comparisons.

Minimize $\sum_{t \in \mathbf{T}} E(t)$,

where

$$E(t(j,k,A,B)) = \begin{cases} 0 & \text{if } (F_A(k.j_A) - F_B(k.j_B)) \cdot (C(j,k,A) - C(j,k,B)) > 0 \\ 1 & \text{otherwise} \end{cases}$$

Figure 24. The objective function for load-index learning

The objective function shown above counts the number of test-cases for which the comparison of load-index functions does not give the same answer as the comparison of actual completion times. For any given test set, minimizing the above function is tantamount to maximizing the percentage of correct comparisons.

B. The Comparator Network: Architecture and Learning Algorithms

A variety of methods are available for learning functions from data, such as regression [50], inductive learning [148], and feedforward neural networks [56, 154]. In comparison, little is known about the problem of learning to compare functions. One exception is the work of Tesauro [179], who invented the comparator-neural-network architecture for learning to compare alternative moves for the game of backgammon. His approach does not directly carry over to the problem of comparing functions of time series. Our approach, described in this section, was motivated by Tesauro's work; however, in adopting his work to the index-learning problem, we have made significant departures from both his network configurations and training algorithms.

Figure 25 shows a schematic of our comparator neural network. Compare this figure with Figure 9. The new figure shows the details of the training algorithm, and the flow of information during a typical

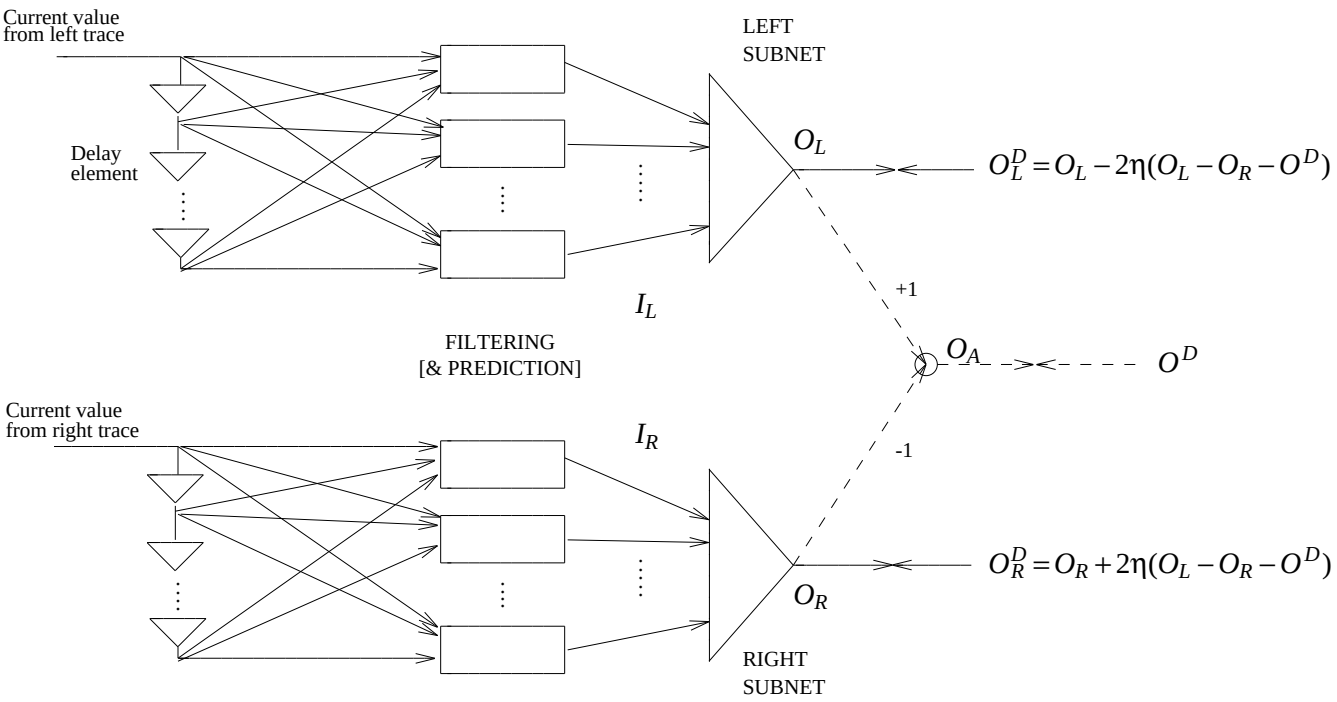


Figure 25. A typical episode in training a comparator neural network

learning trial. Each learning trial involves one test-case from the training set: first, the resource-utilization information from a randomly selected test-case is presented at the inputs (to the left); then the *actual outputs* of the two index functions are computed; based on these outputs, the training algorithm computes the *desired outputs* for each of the index-functions; finally, the two index functions are modified so that future presentations of similar inputs will generate outputs closer to their respective desired outputs. The rest of this section is devoted to the derivation of the training algorithm.

Raw utilization patterns enter from the left of Figure 25; the delay elements create a window of recent values; the traces from each window are smoothed using lowpass filtering and a measure of future resource utilization predicted by extrapolating the smoothed trace (see next section for details). Vectors of these predictions constitute the inputs I_L and I_R of the load-index functions (shown as triangles in Figure 25).

We implemented load-index functions using feed-forward neural networks [118]. Each network comprises three layers of units: an input layer, a hidden layer, and an output layer. The output layer has only one unit called the output unit. Units are interconnected by links. Links 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 parameter called the *weight* of that link. The weight on a link going from the *i*th unit to the *j*th unit of the network is denoted w_{ji}. Our networks are fully connected; that is, 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. Figure 26 shows the schematic of such a feed-forward neural network. The set of parameters *W* for the load-index function at a site consists of all the w_{ji} values, where *i* and *j* are units in the feed-forward network for that site.

There are a number of reasons for using feed-forward networks to implement empirical learning of functions from data. Of these, the most compelling is a theorem of Cybenko [39]: When the output of each unit of a feed-forward network is given by the *sigmoidal function* of its net input, then a network having the structure shown in Figure 26 can approximate any [continuous] function [defined on a unit

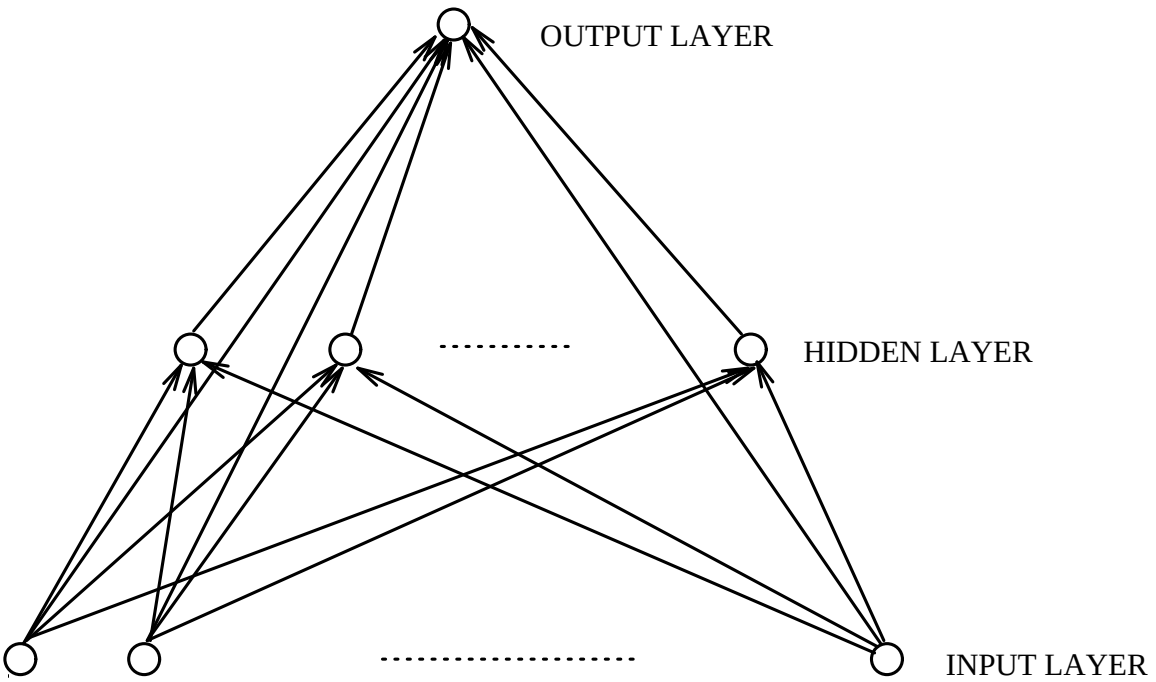


Figure 26. Schematic of a feed-forward neural network

---

cube in the space of inputs] with arbitrary accuracy, provided that the network has sufficiently many units in the hidden layer. The sigmoidal function is defined as follows:

$$f(x) = \frac{1}{1+e^{-x}}. \qquad \text{(sigmoidal function)}$$

A second important reason for using feed-forward neural network is the common availabilty of several *supervised learning methods* [80]; given the actual and desired outputs for a feed-forward neural network, these methods can determine the appropriate modifications to the weights of that network. Of these, the most popular is the generalized delta rule (also known as the ‘‘back-propagation algorithm’’) of Rumelhart, *et al.* [154]. While other better (more accurate and more efficient) learning procedures are known [33, 55, 56], we chose to use the ‘vanilla’ back-propagation algorithm because of its availability in a public-domain simulation package [69].

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 test-case 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 $\sum_j w_{ji} o_j$, where o_j denotes the output of unit j.

Let us denote by O_L the output of the only unit in the output layer of the left subnet; and, correspondingly, O_R, of the right subnet. In order to use the back-propagation learning procedure for training the subnets, we need to determine their desired outputs for every input. Let us denote by O_L^D 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 Equation (4a), 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 Equation (4a). That is,

$$O^D = \frac{C(j,k,A) - C(j,k,B)}{C(j,0,0)}.$$

The value of the objective function shown in Figure 24 will be reduced if both O_A and O^D have the same sign. That can be achived by driving their values closer together. Let us denote by E_{LMS} the sum (over all test-cases in the training set) of squared errors between the actual and the desired outputs of the comparator. That is,

$$E_{LMS} = \sum_t (O - O^D)^2.$$

We can minimize E_{LMS} by performing gradient descent [188]; that is, by adjusting the outputs of the left and right subnets along their respective partial derivatives of error:

$$\Delta O_L \propto -\frac{\partial E_{LMS}}{\partial O_L},$$

$$\Delta O_R \propto -\frac{\partial E_{LMS}}{\partial O_R}.$$

Introducing a constant of proportionality, η, also known as the *learning rate*, we obtain:

$$\Delta O_L = -\eta \frac{\partial E_{LMS}}{\partial O_L},$$

$$\Delta O_R = -\eta \frac{\partial E_{LMS}}{\partial O_R}.$$

Hence, we obtain the desired outputs for the left and right subnets:

$$O_L^D = O_L - 2\eta(O_L - O_R - O^D), \qquad \text{(Training algorithm)}$$

$$O_R^D = O_R + 2\eta(O_L - O_R - O^D).$$

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 test-cases to the comparator. Test-cases are presented in pairs, one after another. If the first test-case in the pair is t_1(j,k,A,B), then the second test-case must be t_2(j,k,B,A). 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. Such an order of presentation should improve the consistency of comparison, while obviating certain architectural constraints imposed by Tesauro [179] in his original design for solving the same (consistency) problem. Tesauro's design achieves consistency between left and right subnets by sharing (or linking [113]) the corresponding weights of the left and right subnets.

C. Data Preprocessing

During actual operation, the preprocessing stage precedes the training stage. In this stage, information about short-term trends of resource-utilization patterns is extracted from the raw measurements made by DWG just before a test job was introduced. This is the best possible scenario as far as communication among sites is concerned; i.e., all sites have the most up-to-date information. Effects due to overhead of communicating this information between sites, as well as those due to the interval of computation (age)

of the load index, will be accounted for in the load-balancing policies.

Smoothing removes high-frequency variations from signals and exposes (short-term as well as long-term) trends in data. Trends exist in resource-usage patterns because of locality in the behavior of programs that generate resource requests. The nature of the trend exposed depends upon the degree of smoothing. Lowpass filtering and moving averages are equivalent ways of smoothing [25, 78]. We prefer the filtering formulation because it allows precise specification of the degree of smoothing. In a lowpass filter, the degree of smoothing is controlled by the *cutoff frequency* of the filter, which (for our purposes) is that fraction of the maximum frequency in the frequency-domain representation of the signal beyond which attenuation is almost total. (We use the symbol F_{max} to denote the maximum frequency.) The lower the cutoff frequency, the greater is the degree of smoothing, and the longer is the interval over which the trend of the smoothed signal can be extrapolated. Figure 27 illustrates the effects of five different cutoff frequencies on the disk-utilization pattern shown in Figure 22, Chapter III. Notice how the pattern at the bottom right of the figure (generated from the raw signal by filtering out 99% of its frequency content) contains just the very long-term trend of disk usage. In comparison, the one above it contains a little more short-term trend because it attenuates only 95% of the frequency content. We implemented our smoothing algorithm using Butterworth filters [78], which are easy to design and implement, and have good BIBO (bounded input/bounded output) stability. The accuracy with which a Butterworth filter approaches complete attenuation beyond the cutoff frequency depends upon its *order*. Larger orders give sharper cutoff but introduce greater delays at the output.

A priori, it is not clear how much smoothing will be adequate. One solution to this problem is the use of adaptive filters [198], which adapt their parameters in response to errors at their output. However, unlike Butterworth filters, adaptive filters cannot be reliably implemented using off-the-shelf techniques. Therefore, we chose to use a bank of Butterworth filters, covering a reasonable range of cutoff frequencies. Specifically, we chose the five cutoff frequencies illustrated in Figure 27: 1%, 5%, 10%, 25%, and 50%. All our filters were of order 7, which is large enough to get a sharp cutoff and yet small enough that

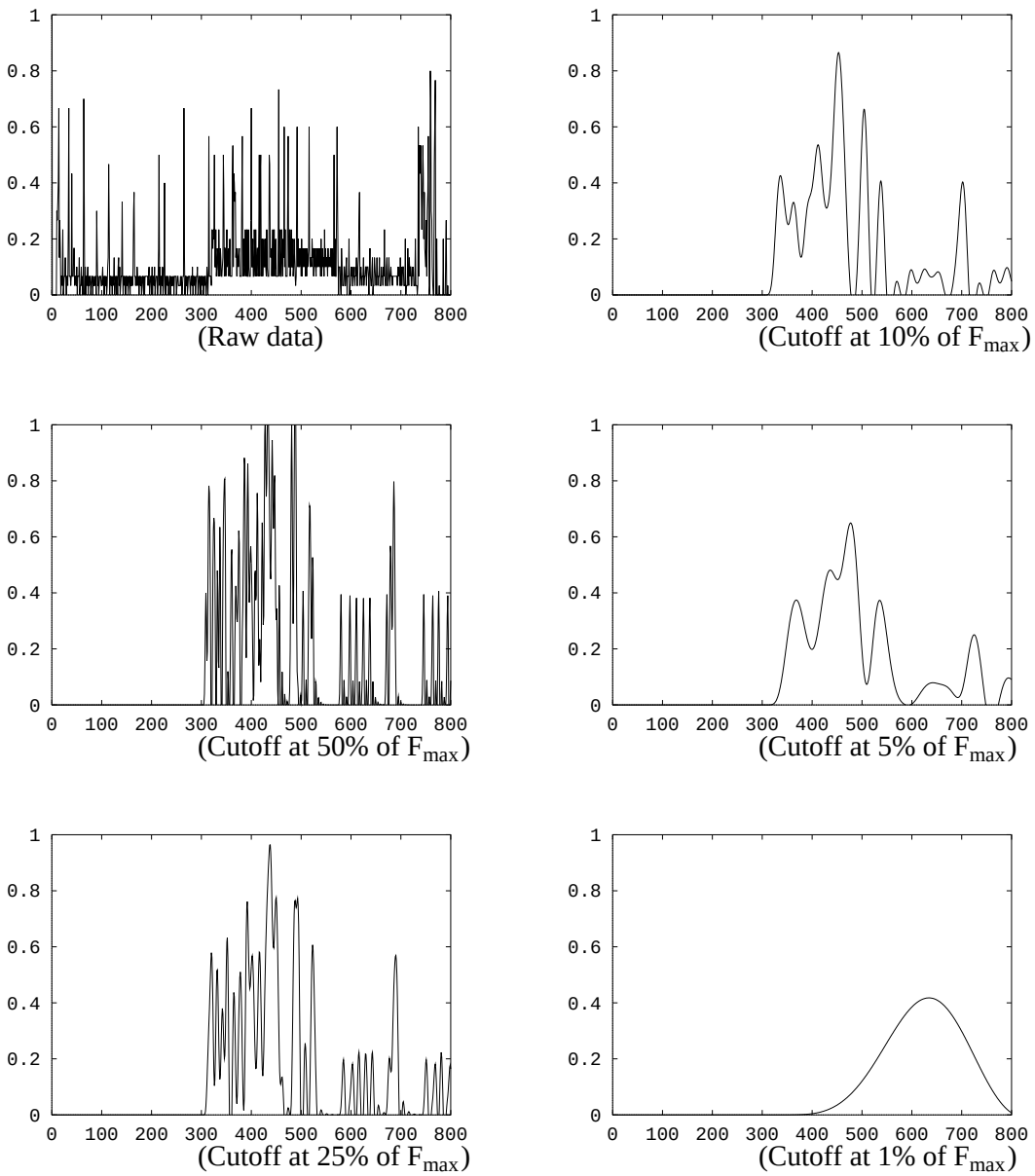


Figure 27. Effects of lowpass filters of order 7 with different cutoff frequencies

---

it does not incur too much delay. Any other reasonable choice of cutoff frequencies and filter order should work equally well.

Trends in resource usage are useful because they are indicative of the competition an incoming job could face from the (background) processes generating the pattern observed just before the incoming job's arrival. The amount of work that the background processes will create for the incoming job can be represented by the area under the true resource-utilization curve. However, the true curve is not known at the time of computing the load index; instead, we can *estimate* the amount of future background workload by extrapolating the current trends and computing the areas under the extrapolated curves.

In our implementation, trends are detected by fitting a parametric curve to a window of recent (filtered) resource-utilization values. Two types of trends are considered: linear and exponential. Other parameteric forms are possible; however, our experience suggests that resource-utilization patterns of typical real workloads are adequately represented by a combination of linear and exponential forms. Figure 28 illustrates our approach with a schematic diagram of the fitting process. First, we use (dynamic) regression [123] to fit the parameteric form to a moving window of recent filtered values. (In our implementation, we used windows containing 250 uniformly-spaced samples 20 milliseconds apart.) Next, we compute the areas under each fitted curve. Since there are 5 different filters and 2 different extrapolation

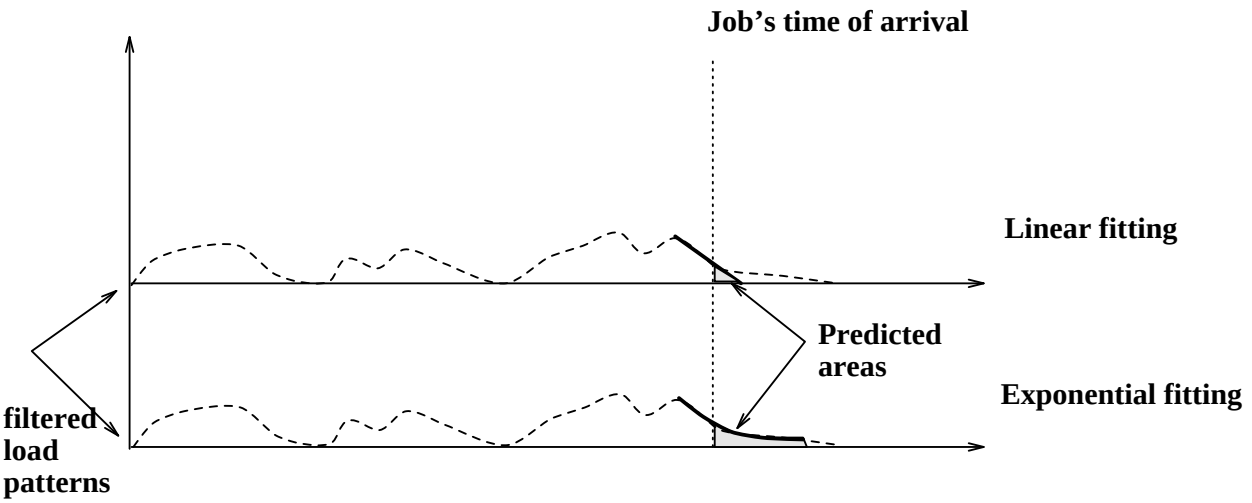


Figure 28. Trend extraction via linear and exponential fitting

techniques, we obtain 10 values of predicted areas per resource. Since each site has access to four resources — CPU, memory, network, and disk — each load pattern on a site yields 40 areas just before the job's arrival. (Disk utilization is set to zero for diskless sites.) A vector containing the areas of a site's local and shared resources is applied at the input layer of that site's feed-forward network. The function computed by the network becomes the predicted value of that site's load index.

Numerous alternatives are available for smoothing and extrapolation. For example, one can use (i) adaptive filters [198] instead of a bank of pre-designed filters; (ii) singular-value decomposition [70] to identify the recent utilization values useful for prediction of relative completion time; or, (iii) recurrent neural networks [192], which consider only the current input at any given time. The approach we have chosen is pragmatic and off-the-shelf tools and techniques can be used to implement it. Most importantly, our preprocessing methods achieve their purpose with low overhead and, in that respect, are at least as good as any other alternative we could have implemented.

D. Empirical Results

This section presents our results on learning load indices for a system containing four sites. Each test-case contains 40 extrapolated values of areas under extrapolated resource-utilization patterns and two additional fields, $C(j,k,A)$, and $C(j,0,0)$, where j identifies the job, k identifies the background load pattern, and A the site at which j was executed.

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

We begin by randomly assigning 10 percent of the jobs to the test set and, the remaining 90 percent, to the training set. The networks are 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 alternate; during each *epoch* of training, the networks are trained on 1,000 randomly chosen comparisons from the training set, and tested on 100 randomly chosen comparisons from the test set. Each comparison involves two loading situations for the same job. Training is done in two stages. Stage 1 starts with two identical networks (with random initial weights) for each site; these are trained to compare different loading conditions for the same job at the same site. In Stage 2, just the left networks from each site are further trained to compare different sites for the same job under the same system-wide load pattern. Unlike in Stage 1, the roles of the left and the right load-index functions (Figure 25) cannot be reversed during Stage 2.

If we were solely interested in the four-processor system that we measured, Stage 1 would be unnecessary. However, as explained in Chapter VI, we can simulate larger systems by 'cloning' sites. In cloning, the loading conditions for a clone at any given time are selected randomly from the loading conditions of its parent at all times. Without getting into the details of cloning, we would like to note that comparisons between clones (or between clones and their parents) are tantamount to comparisons between differrent loading situations for the same job at the same site.

Figures 29 and 30 show the learning curves for Stages 1 and 2, respectively. These figures show that (i) accuracy improves with training; and (ii) almost 90% accuracy is attained within only a few epochs. Stage 1 starts with near-50% accuracy because initial weights are set are randomly. (Comparisons using random weights should be about as accurate as those using the toss of a fair coin.) Stage 2 starts with networks already trained to compare different loading conditions for the same site; therefore, its learning curve exhibits (i) initial accuracy higher than 50%; and (ii) steeper growth than Stage 1's curve.

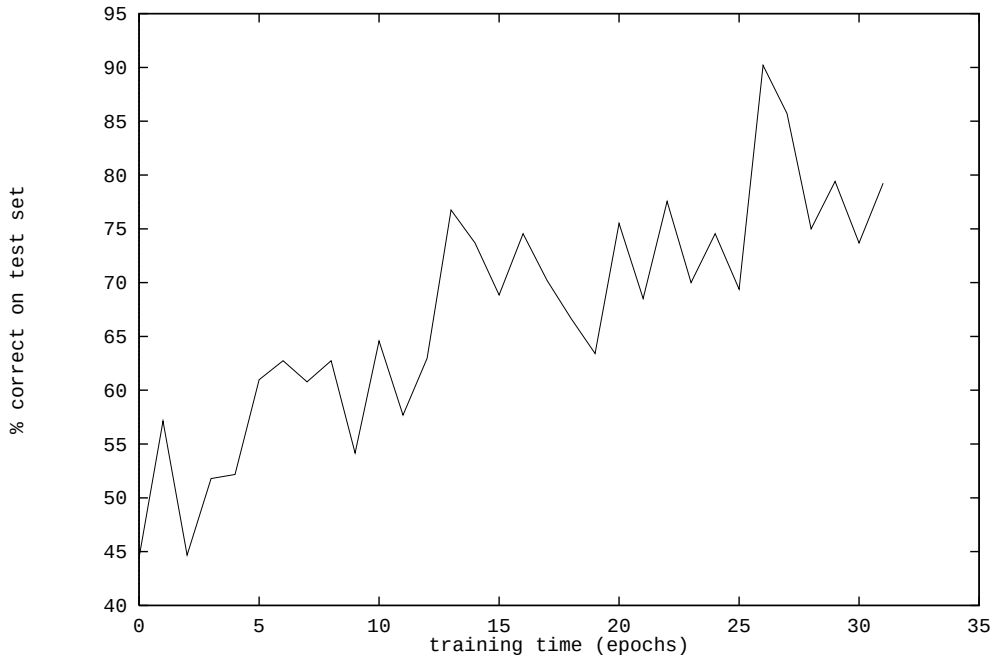


Figure 29. Learning curves for within-site comparisons

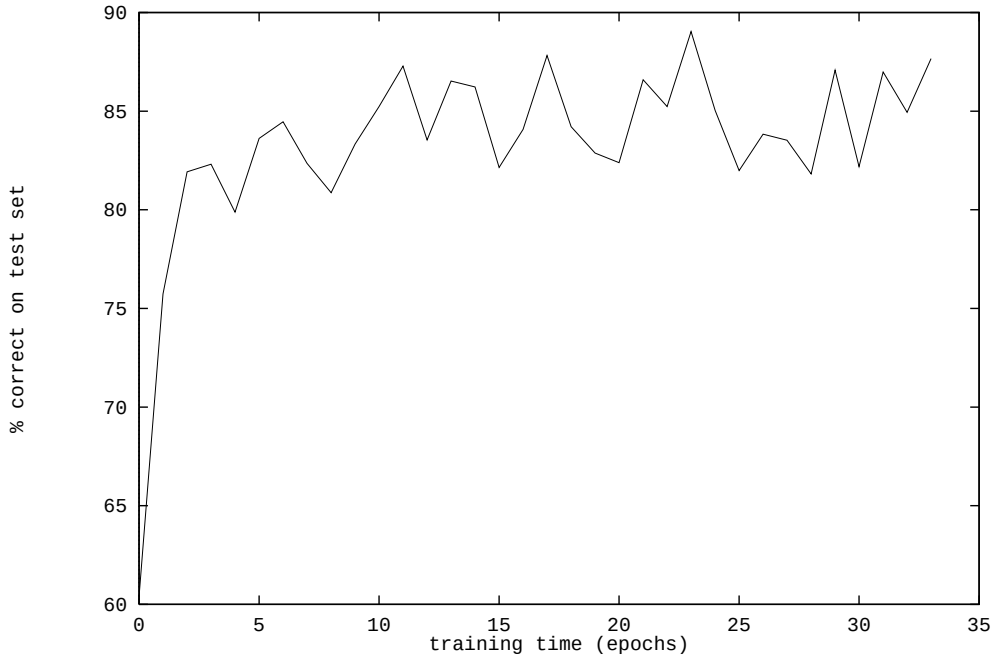


Figure 30. Learning curves for across-site comparisons

E. Evaluation of Load Indices Learned using Comparator Networks

Our overall goal in training comparator networks is to learn load-index functions using which alternative sites for an incoming job can be *ranked* as though by their (as yet unknown) relative completion times. If we rank all available test-cases (all the different sites and loading patterns) of a job by their load indices, then we would like this ranking to have high correlation with the *true ranking* of these test-cases. The true ranking is, of course, the one induced by the measured completion times, $C(j,k,A)$.

One way to measure the correlation between two different rankings of the same data is to compute the *rank-correlation coefficient* [122], defined as:

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

where d_i is the difference between the two different ranks of the ith test-case, and n is the number of test-cases ranked. The value of r ranges between -1 and +1, where -1 indicates strong disagreement between rankings, 0 indicates no correlation, and +1 indicates strong agreement. This coefficient can be computed for each of the 56 jobs in our test-case database.

If there is *no relationship* between the two rankings, then the sampling distribution of r can be approximated with a normal distribution N(0, σ), where σ, the standard deviation, equals $(n-1)^{-0.5}$ [122]. If we let $z = r/\sigma$, then z will be distributed according to the two-sided standard normal distribution N(0,1). Using this fact, we can set up the following *null hypothesis*:

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

We would like to *reject* H0 with high confidence. Given a confidence level α, we can look up the table of standard normal distribution to find an A such that

$$\Pr(|z| \le A) \ge \frac{1-\alpha}{2}.$$

In particular, *A* takes on the values 2.58, 1.96, 1.65, and 1.15, respectively, for corresponding α values of 0.01 (99% confidence), 0.05 (95%), 0.1 (90%), and 0.25 (75%). Knowing *A* and *n*, we can determine *significance levels* for *r*. If, for instance, the absolute value of *r* exceeds its 99% significance level, then H0 can be rejected with 99% confidence. Tables VII and VIII show, for each test-job, the number of test-cases *n* of that job, the associated σ value, and the measured rank-correlation coefficient *r*. The four

Table VII
Significance test for coefficient of rank-correlation (I)

Job		n	σ	r	Significance levels			
j	c				99%	95%	90%	75%
0	1	96	0.102598	0.7113	0.264702*	0.201092	0.169286	0.117988
0	2	96	0.102598	0.754341	0.264702*	0.201092	0.169286	0.117988
0	4	96	0.102598	0.786096	0.264702*	0.201092	0.169286	0.117988
0	8	96	0.102598	0.809441	0.264702*	0.201092	0.169286	0.117988
0	17	96	0.102598	0.818218	0.264702*	0.201092	0.169286	0.117988
0	35	96	0.102598	0.781009	0.264702*	0.201092	0.169286	0.117988
0	77	96	0.102598	0.782976	0.264702*	0.201092	0.169286	0.117988
0	168	92	0.104828	0.818612	0.270457*	0.205464	0.172967	0.120553
0	999	96	0.102598	0.818638	0.264702*	0.201092	0.169286	0.117988
1	1	96	0.102598	0.667797	0.264702*	0.201092	0.169286	0.117988
1	2	96	0.102598	0.735879	0.264702*	0.201092	0.169286	0.117988
1	4	96	0.102598	0.759251	0.264702*	0.201092	0.169286	0.117988
1	9	96	0.102598	0.790505	0.264702*	0.201092	0.169286	0.117988
1	18	96	0.102598	0.805195	0.264702*	0.201092	0.169286	0.117988
1	39	96	0.102598	0.767512	0.264702*	0.201092	0.169286	0.117988
1	81	96	0.102598	0.778608	0.264702*	0.201092	0.169286	0.117988
1	999	96	0.102598	0.7906	0.264702*	0.201092	0.169286	0.117988
2	1	84	0.109764	0.780622	0.283192*	0.215138	0.181111	0.126229
2	46	84	0.109764	0.744882	0.283192*	0.215138	0.181111	0.126229
2	115	84	0.109764	0.803402	0.283192*	0.215138	0.181111	0.126229
2	999	96	0.102598	0.786367	0.264702*	0.201092	0.169286	0.117988
3	1	96	0.102598	0.502957	0.264702*	0.201092	0.169286	0.117988
3	13	96	0.102598	0.663714	0.264702*	0.201092	0.169286	0.117988
3	999	96	0.102598	0.785106	0.264702*	0.201092	0.169286	0.117988
4	1	96	0.102598	0.831823	0.264702*	0.201092	0.169286	0.117988
4	37	96	0.102598	0.786069	0.264702*	0.201092	0.169286	0.117988
4	139	92	0.104828	0.7905	0.270457*	0.205464	0.172967	0.120553
4	999	96	0.102598	0.779748	0.264702*	0.201092	0.169286	0.117988

Table VIII
Significance test for coefficient of rank-correlation (II)

Job		n	σ	r	Significance levels			
j	c				99%	95%	90%	75%
5	1	96	0.102598	0.84589	0.264702*	0.201092	0.169286	0.117988
5	37	92	0.104828	0.789066	0.270457*	0.205464	0.172967	0.120553
5	116	92	0.104828	0.736634	0.270457*	0.205464	0.172967	0.120553
5	999	96	0.102598	0.732311	0.264702*	0.201092	0.169286	0.117988
6	1	96	0.102598	0.617906	0.264702*	0.201092	0.169286	0.117988
6	4	96	0.102598	0.588009	0.264702*	0.201092	0.169286	0.117988
6	11	96	0.102598	0.566793	0.264702*	0.201092	0.169286	0.117988
6	25	96	0.102598	0.574986	0.264702*	0.201092	0.169286	0.117988
6	55	96	0.102598	0.543652	0.264702*	0.201092	0.169286	0.117988
6	135	96	0.102598	0.45898	0.264702*	0.201092	0.169286	0.117988
6	999	96	0.102598	0.526614	0.264702*	0.201092	0.169286	0.117988
7	1	96	0.102598	0.595496	0.264702*	0.201092	0.169286	0.117988
7	3	96	0.102598	0.688565	0.264702*	0.201092	0.169286	0.117988
7	7	96	0.102598	0.73923	0.264702*	0.201092	0.169286	0.117988
7	16	96	0.102598	0.768557	0.264702*	0.201092	0.169286	0.117988
7	31	96	0.102598	0.799267	0.264702*	0.201092	0.169286	0.117988
7	60	96	0.102598	0.777767	0.264702*	0.201092	0.169286	0.117988
7	122	96	0.102598	0.768421	0.264702*	0.201092	0.169286	0.117988
7	999	96	0.102598	0.770062	0.264702*	0.201092	0.169286	0.117988
8	1	96	0.102598	0.78615	0.264702*	0.201092	0.169286	0.117988
8	41	96	0.102598	0.776492	0.264702*	0.201092	0.169286	0.117988
8	195	96	0.102598	0.776207	0.264702*	0.201092	0.169286	0.117988
8	999	96	0.102598	0.779965	0.264702*	0.201092	0.169286	0.117988
9	1	96	0.102598	0.801845	0.264702*	0.201092	0.169286	0.117988
9	29	96	0.102598	0.804164	0.264702*	0.201092	0.169286	0.117988
9	67	96	0.102598	0.793977	0.264702*	0.201092	0.169286	0.117988
9	148	96	0.102598	0.802686	0.264702*	0.201092	0.169286	0.117988
9	999	96	0.102598	0.804164	0.264702*	0.201092	0.169286	0.117988

rightmost columns of each table show the four significance levels. Each job is identified by a tuple <j,c>, where j identifies the benchmark and c identifies a selected checkpoint within that benchmark. Complete benchmarks are denoted by a c value of 999. The significance level achieved in rejecting H0 is identified by an asterisk in the corresponding column for each test-job. We conclude from these tables that the null hypothesis can be rejected with 99% confidence.

One final detail that needs to be worked out involves the sensitivity of our load index to job size. While pairwise comparisons of loading conditions may form the basis of 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. Keeping in mind the objectives of load balancing (maximum 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. It does not matter too much whether the site with the least load index is also the site with the least completion time, so long as the speed-up achieved by scheduling at the least-loaded site is close to the optimal speed-up. In Figure 31, 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 system. Test jobs are 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 checkpoints are recorded. Two policies are compared: (i) opt, which always places the incoming job at the site with the optimum (least) completion time; (ii) min, which always schedules an incoming task at the site with the smallest load index. Assuming no overhead of remote execution, we calculate, for each test-case, the speed-up over local execution achieved using these policies. In the bottom part of Figure 31, 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, preemptive 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.

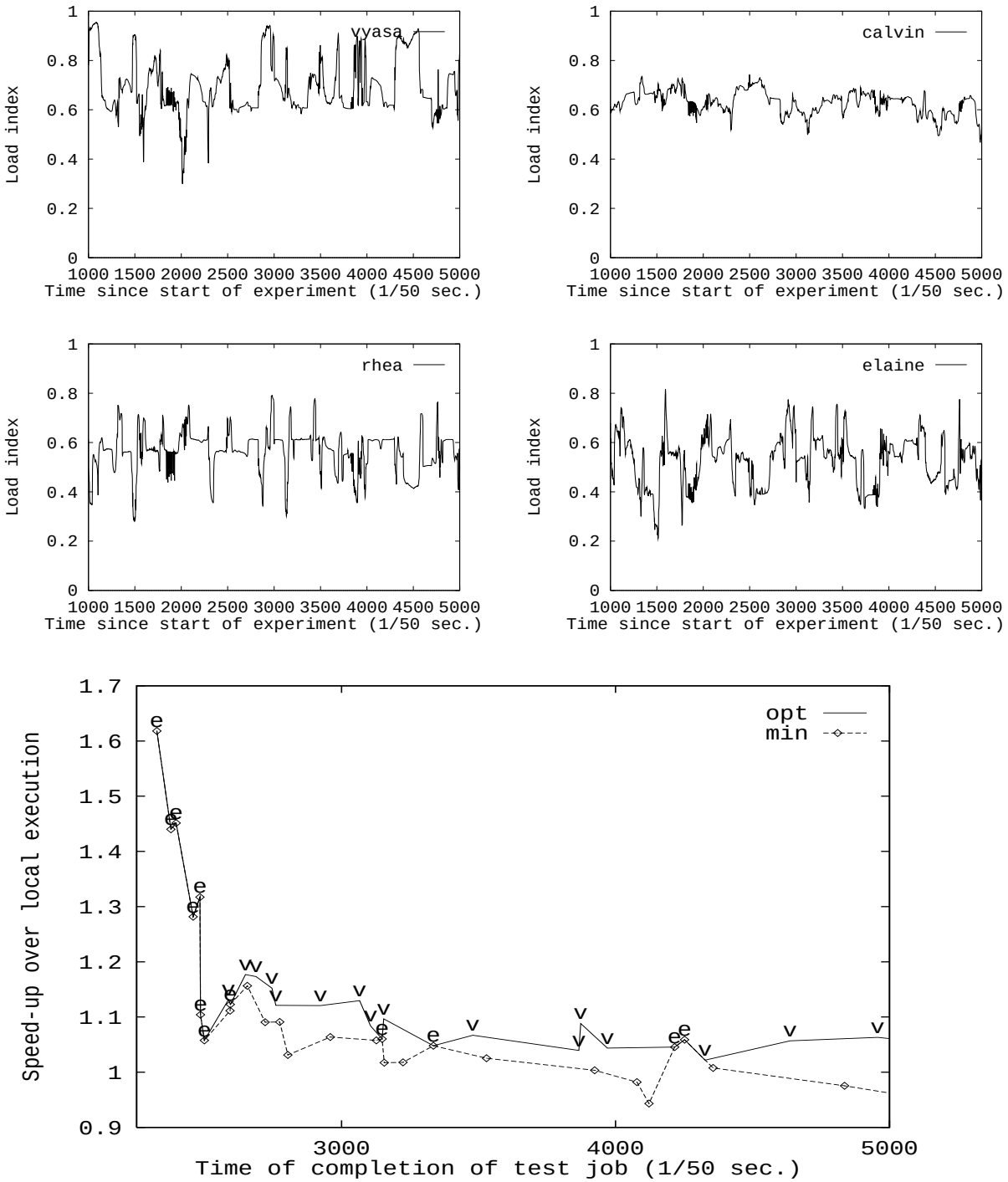


Figure 31. Performance of all test-jobs under load #23; policies: opt, min

Figure 32 compares (once again using speed-up over local execution) the performance of four different strategies for the load pattern and arrival time shown in Figure 31. This time, the speed-ups are plotted against job lengths. Besides min and opt, two additional policies are considered: (i) rnd, which randomly chooses one of the four destinations with equal probability; and (ii) max, which places each incoming job at the site with the worst (maximum) completion time. The figure shows that opt and max determine the bounds on speed-up, and that min consistently outperforms rnd for short jobs. As the length of the job increases, the performance of min drops to the same level as rnd.

Figure 33 shows the performance of the same four policies under one of the load patterns created using test jobs during the first phase of data collection. Since this load pattern has relatively fewer surprises, the speed-up of min is the same as that of opt, except for one or two long jobs. This load pattern is also more typical of the load patterns used in our experiments. We find that when loads exhibit locality

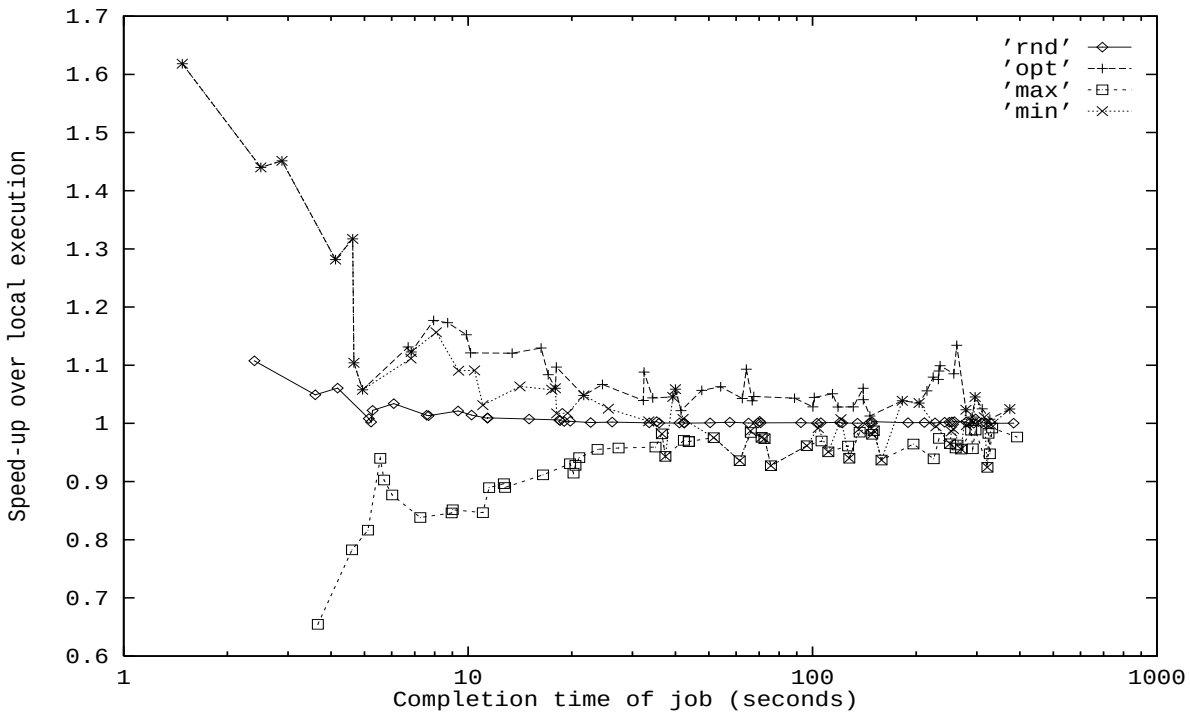


Figure 32. Performance of all test-jobs under load #23; policies: opt, min, rnd, and max

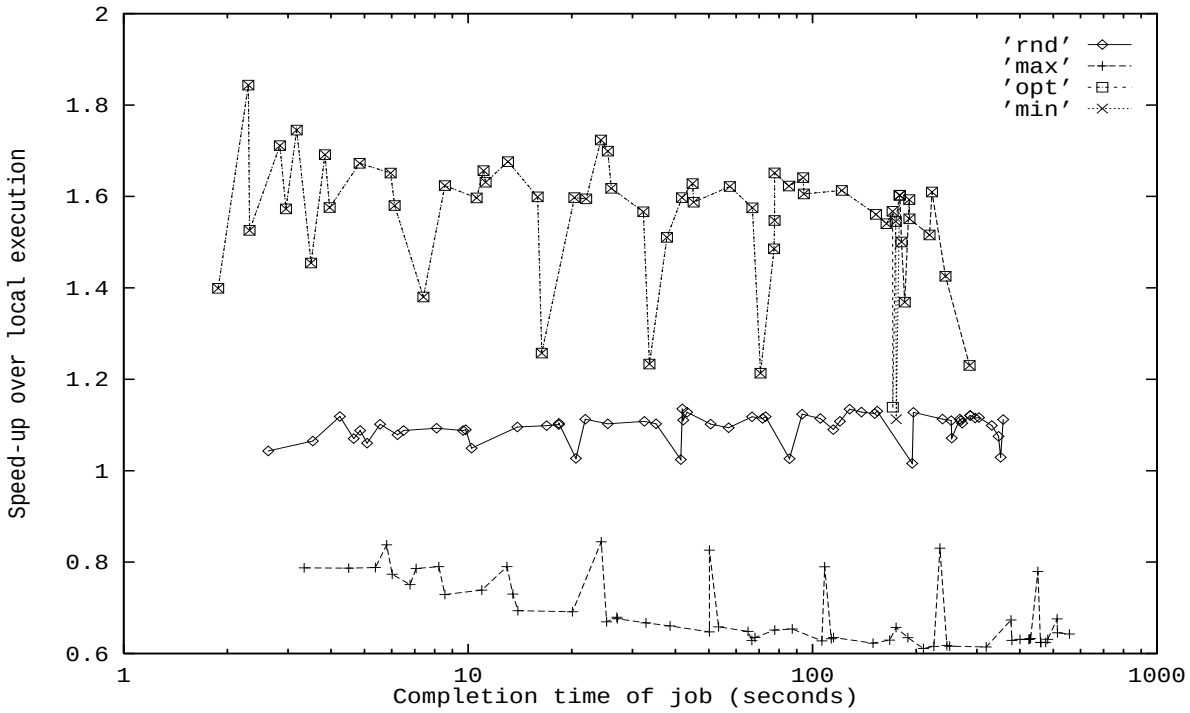


Figure 33. Performance of all test-jobs under load #11; policies: opt, min, rnd, and max

---

then the indices learnt using comparator networks work well even for long jobs.

We now describe our experiments for comprehensive evaluation of min, the policy that always picks the site having the minimum of load indices predicted by comparator networks at different sites. We evaluated this policy on all test-jobs under all 24 load patterns collected in phase I of data collection. In order to study the relationship between the length of a job and the speed-up achieved for that job, we created data-pairs $<l_i,e_i>$, where the first item of each pair is the length of the job and the second, the corresponding speed-up over local execution. (We continue to ignore the age of load indices due to overhead of communication and due to the periodic nature of load-index computation.)

Figure 34 shows a contour plot of the probability of achieving certain speed-up for jobs of certain length. The values along the X-axis represent speed-up over local execution, and those along the Y-axis,

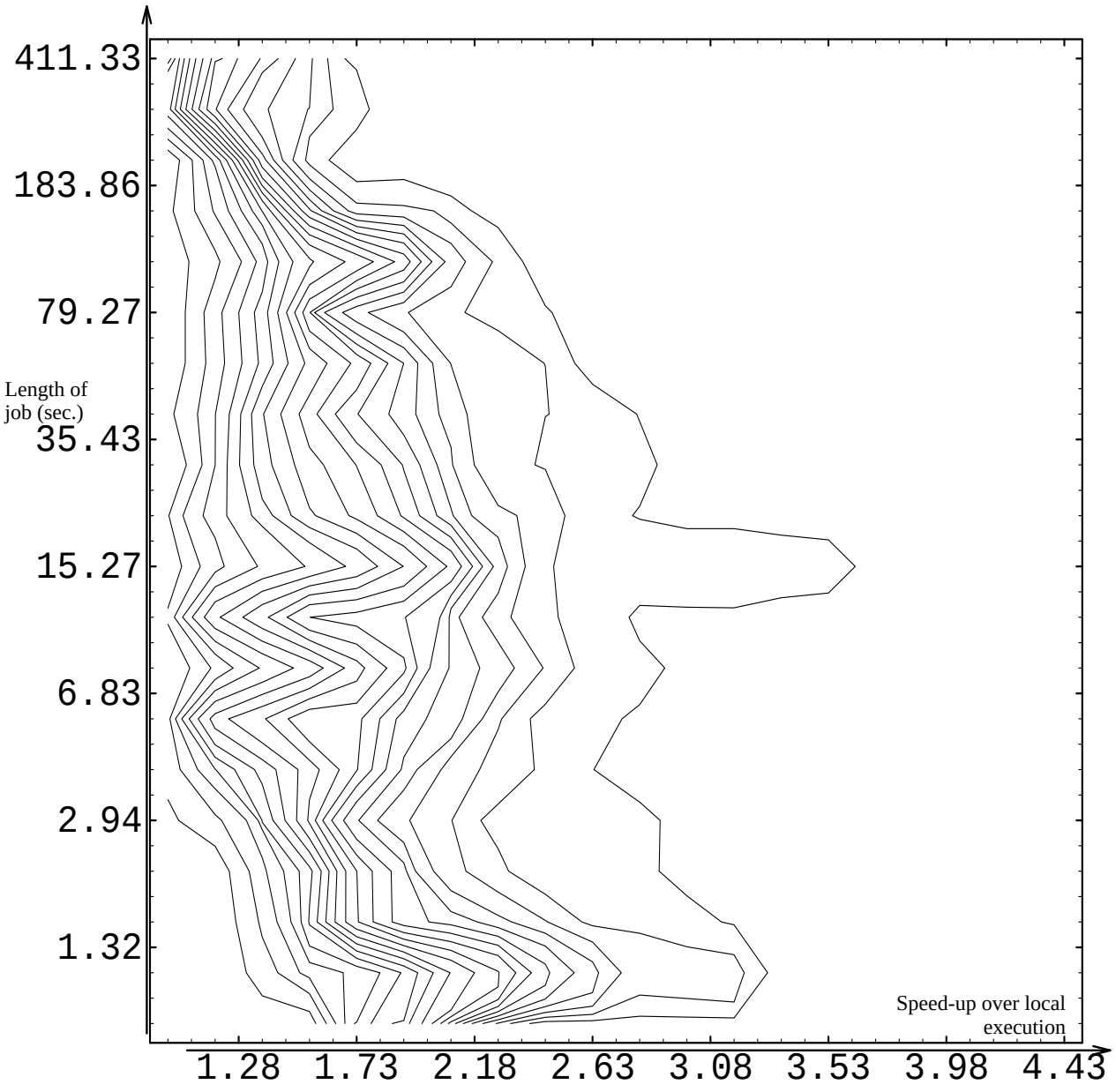


Figure 34. Contour plot of cumulative probability of speed-up; policy: min; no overheads or delays

---

the length of the job in seconds. Y-values (job lengths) range from 0.66 seconds to 439.94 seconds. X-values (observed speed-ups) range between 0.924524 and 4.92187. Since there are many more short jobs than there are long ones, we used logarithmic scaling for the Y-axis. Nineteen contours, each connecting

X-Y points having equal cumulative probability of speed-up, are shown in the figure; they divide up the space into twenty regions of equal (5%) probability. While speed-ups higher than 1.5 (more than 50% improvement over local execution) occur quite frequently for short jobs, they almost never occur for jobs that take more than 3 minutes to complete.

Before concluding this section, we would like to emphasize the importance of DWG in effective evaluation of load indices. Suppose we did not have a synthetic workload generator. We could still measure (real) load patterns, introduce test jobs, and measure their completion times. However, since real patterns do not repeat in a predictable fashion, we could only measure the completion time of a test-job under a particular loading pattern for only one site. Then, there would be no way to compare the load indices at different sites with respect to the same job and *the same system-wide loading pattern*. Thus, synthetic workload generation is necessary for effective evaluation of load-indices meant for use in load-balancing experiments.

The weights of the trained networks are dumped into a file. We have developed software that automatically generates a C function from these weight dumps. Given a vector of 40 inputs, the new function produces exactly the same output as the dumped network. This load-index function can be linked with any program, including the operating-system kernel. Given the projected resource-utilization values produced by the filtering and extrapolation utilities, this function computes a load index. Thus, we have fully automated the process of creating meaningful load indices for our distributed computer system.

Our results, both from this section and the preceding one, demonstrate that the comparator network learns quickly and effectively. The indices learnt by it induce meaningful rankings on loading conditions across the system, as well as on loading conditions for the same system at different times.

F. Chapter Summary

We have achieved our goal of learning meaningful load-index functions from data. Using the data collected by DWG, we are able to train comparator networks to effectively compare alternative loading conditions for an incoming job. The training process is fully automated and requires no down time because it is performed off line. Further work is needed to fully integrate the learnt load-index function into the operating-system kernel; this will involve the development of on-line utilities for filtering and extrapolation.

As a final illustration of load-index learning, we show in Figure 35 the raw utilization patterns and the associated load patterns for rhea, a diskless Sun 3/60 client with 24 megabytes of main memory, which is part of our four-processor system. The load index is sensitive not only to the CPU but also to

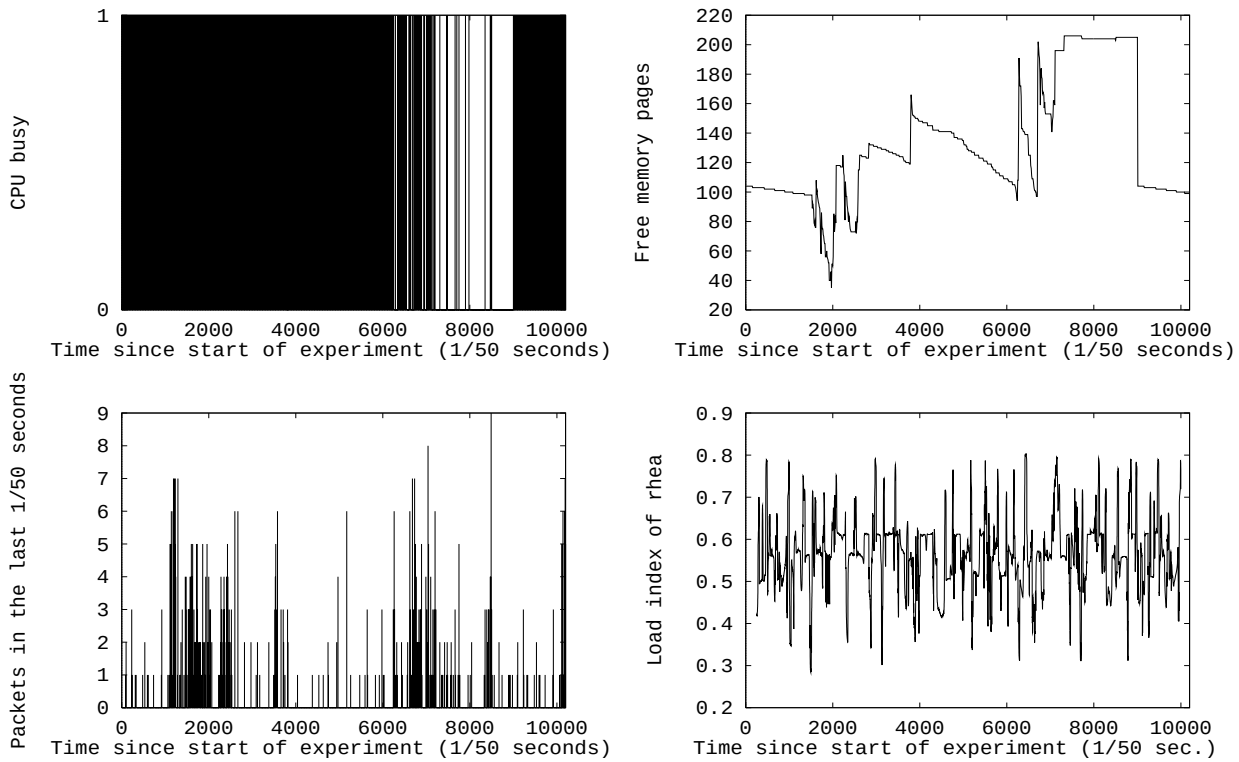


Figure 35. Raw resource-utilization patterns and their load index

network traffic and memory. Just after 4,000 time units into the experiment, there is a period of low network traffic and high memory availability. Despite the fact that CPU stays busy through this period, the load index registers a noticeable drop. Around 8,000 time units into the experiment, there is a lull in CPU activity and plenty of memory is available; however, the load index registers a drop only after network traffic quiets down as well. Furthermore, the load index shows its highest stable value at about 2,000 time units into the experiments, when CPU is busy, memory is unavailable, and network trafic is high.

CHAPTER V.

STRATEGY LEARNING: A SURVEY OF ISSUES AND APPROACHES

Knowledge is the small part of intelligence that we arrange and classify.

- Ambrose Bierce

In this chapter, we identify the key issues in strategy learning and characterize their complexity in terms of various attributes of learning tasks. We survey existing approaches to these issues, and group strategy-learning systems by the commonality of their approaches into four general architectures. We also address practical issues in implementation and comparative evaluation of strategy-learning techniques.

A. Introduction

Much recent work in machine learning has targeted *sequential problems* in decision, control, and optimization. Sequential problems abound in robotics, navigation, tracking, and dynamic scheduling. They require the problem solver to make multiple choices or initiate multiple actions, one after another, in order to reach and/or maintain a ‘‘desired state.’’ Such a sequence of choices or actions constitutes a solution; the knowledge used for generating it is a *strategy*. Problem solvers employ strategies to systematically generate solutions to given problem instances. This survey addresses automated learning of such strategies.

1) State of the art

Traditionally, strategies have been designed by experts using prior knowledge, and refined manually using trial and error. Recent attempts to automate these processes have produced *strategy-learning systems* that run the gamut of applications from dynamic load balancing to symbolic reasoning and

combinatorial optimization. Despite its volume and diversity, the literature on strategy-learning systems lacks a systematic characterization of the relationship between applications, algorithms, and architectures. Prior surveys [96, 104, 131, 169] have focused on well-defined symbolic learning tasks characterized by simple feedback schemes and knowledge-rich learning environments.

Time is an important parameter of sequential behavior. When strategy-learning systems are applied to sequential problems, several temporal problems arise. These generally involve *projection* for characterizing the state of a system either into the future or in the past; the former are called prediction (or forward projection) problems, and the latter, backward projection. The state of a system may evolve under the influence of either the problem solver's decisions (*causal dynamics*), or the passage of time (*natural dynamics*), or both. Difficult projection problems result in the last case. Learning systems developed by artificial intelligence (AI) researchers have limited success in coping with natural dynamics [6, 7, 28, 31, 37, 43, 53, 99, 130, 136, 142, 147, 150, 152].

When learning strategies by trial and error, a learning system may experiment with multiple strategies (candidates). The measured performance of tested candidates provides *feedback* for guiding both the selection of candidates for future tests and the modification of incumbents toward improved performance. The translation of feedback into strategy modifications is called *credit assignment* [125]. In sequential decision making, several decisions may be evaluated at once; credit/blame for observed performance must be distributed first among decisions and then among the (heuristic) rules leading to each decision. These two stages give rise to, respectively, the temporal and structural credit-assignment problems [173]. Credit assignment requires extensive knowledge of the problem domain. The temporal credit-assignment problem (like the projection problems described above) is complicated by time-varying states. Existing systems solve this problem using either ad hoc schemes [132, 199], prior knowledge [43], or *Markovian representations* (future states may depend on only the current state but not on any past state) [15]. There is no general yet rational solution to this problem.

In the absence of closed-form objective functions, strategy-learning systems are forced to optimize *measured* performance. They need to transform such (ill-posed) problems into well-defined ones, which the problem solver can solve. This is usually achieved by inducing a model of the true objective function from data. Several existing systems can learn *evaluation functions* (mappings from the state space to scalar performance metrics) [37, 106, 161, 178]; however, the learning of mappings from state sequences to scalar metrics has not been addressed in the literature, especially for non-Markovian representations.

The challenging problems in strategy learning are, therefore, the development of techniques to cope with time-varying parameters (natural dynamics), non-Markovian representations, and ill-defined objective functions.

2) Overview of the chapter

We first introduce attributes for classifying the applications, methods, and architectures of strategy-learning systems. We use the term *strategy-learning task* to denote a triple (P, PS, E), comprising a performance task (P), a problem solver (PS), and a learning environment (E) (Table IX). Various aspects of P, PS, and E dictate the form and complexity of learning algorithms and system architectures. Studying these aspects is important for both characterizing the limitations of existing systems and finding an appropriate learning system for a given application.

Figure 36 shows a coarse classification of strategy-learning tasks. Broadly, learning tasks can be classified as either well-posed or ill-posed on the basis of the objective functions of their performance tasks. Ill-posed learning tasks have performance tasks with *ill-defined objective functions*; that is, the objective functions are not specified as closed-form functions of the problem solver's inputs. Ill-defined objective functions can be further classified as either measurable or unknown. Orthogonally, one may classify learning tasks on the basis of the feedback structure of their learning environments. Here, learning environments can be classified as either reactive (those that produce feedback) or non-reactive (those that don't). Further, feedback may be immediate (which occurs regularly after each decision) or delayed

Table IX
Components of strategy-learning tasks

Component	Description
Performance Task	Defines a class of problems whose instances are to be solved.
Problem Solver	Solves instances of performance task using strategies to decide what operation(s) to perform at each decision point.
Learning Environment	Reacts to problem solver's actions by providing feedback; may also provide prior knowledge of problem domain.

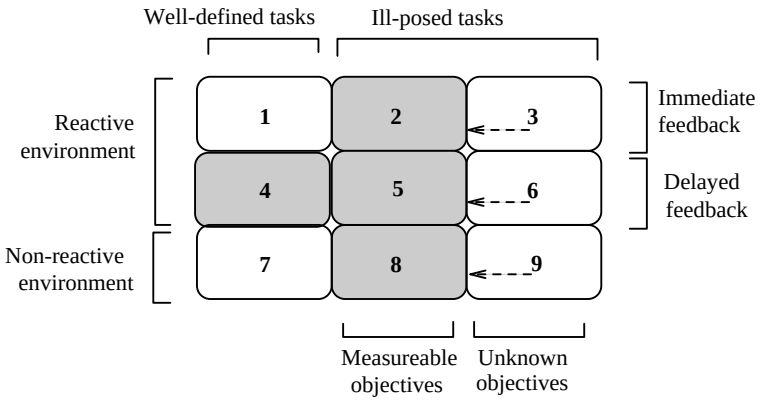


Figure 36. Classification of strategy-learning tasks

Shaded areas in this figure indicate classes of strategy-learning tasks addressed in this thesis. A problem with unknown objective may be solved either by finding an optimal solution to the same problem with a user-specified objective function (as shown by the dashed arrows), or by finding a feasible solution to the same problem with additional constraints.

Table X
Examples of strategy-learning tasks

Class	Target Problem	Class	Target Problem
1	The monkey-and-banana problem [57]	6	Dynamic load balancing to minimize completion time [116]
	Shooting distant targets		Foraging for food and mates [189]
2	Controlling a robotic arm to reach around obstacles using visual feedback [120]	7	Numeric optimization [110]
	Tracking/chasing an unpredictable moving target		Solving the 15-puzzle problem in the fewest moves
3	Determining action/inaction probabilities from rewards [140]	8	''Blackbox'' optimization [1]
	Planning patrol routes for policing a territory		Maximizing fault coverage in digital VLSI circuits by adaptive selection of test patterns [182]
4	Symbolic integration [131]	9	Iterative stereo vision [164]
	Adaptive routing and congestion-control in packet-switched networks [170]		Designing neural networks to optimize complexity as well as accuracy [184]
5	Balancing a pole by moving it left or right [14]		
	Backing a trailer-truck to a loading dock [143]		

(which occurs intermittently). Table X illustrates the resulting classification with examples of strategy-learning tasks.

Our foucs in this thesis is on the classes of strategy-learning tasks shown shaded in Figure 36. These problems are characterized by slow reactive learning environments and performance tasks having ill-defined but measurable objective functions. The unshaded boxes in the third column of Figure 36 correspond to problems with one or more unknown objective functions in their performance tasks. For these problems, it is not clear at the outset which objective to optimize. A common technique to solve a problem in this class is to transform it into another with either a well-defined or an ill-defined but measurable objective function This transformation is indicated by the dashed arrows in the figure. Another common technique is to add constraints to the problem and to solve for a feasible solution in the constrained problem. Finally, the unshaded boxes in column 1 represent well-posed problems with immediate feedback as well as those without feedback. Such problems have been studied extensively and can be learned by existing learning techniques in AI [121].

In Section B, we analyze the structure of strategy-learning tasks. In Section C, we identify the key issues in strategy learning and characterize their complexity in terms of the various attributes of learning tasks. Section D describes techniques for addressing these issues, and Section E compares four different architectures of strategy-learning systems. Problems of implementation and evaluation are addressed in Section F.

B. Structure of strategy-learning tasks

In this section, we elaborate on our view of strategy-learning tasks as triples (P, PS, E), comprising a performance task (P), a problem solver (PS), and a learning environment (E). P defines a *class* of problems whose instances are presented to PS. An *instance* is defined by an initial assignment of values to the input variables of P. When presented an instance of P, PS responds by instantiating the decision vari-

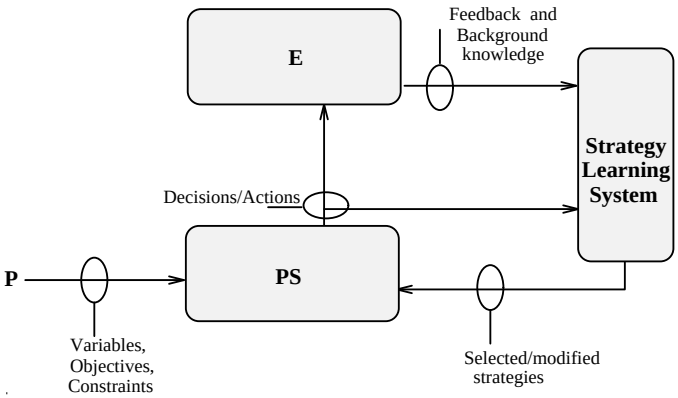


Figure 37. Typical information flow in a strategy-learning system

---

ables of P. Such instantiation results in action via operator application, transforming both the external state of E and the internal state of PS. E responds to operator application by modifying the values of some input variables, and sometimes with *feedback* that indicates the quality of the current and the recent external states. PS reacts to the new external state by either applying another operator or stopping; it uses strategies to decide what operator to apply next. The role of a strategy-learning system (Figure 37) is to use the feedback received from E in order to improve the strategies used by PS in such a fashion that i) future actions will produce more favorable feedback, and ii) optimal or near-optimal stopping states (if any) will be reached quickly.

Besides selecting and modifying existing strategies, a learning system can improve the problem solver in several other ways. For example, it can learn a model of the objective function, or learn to predict future states, or learn to predict the performance resulting from a proposed action. The first of these is critical for ill-posed tasks; the second, for coping with natural dynamics; and the third, for coping with causal dynamics. Whether or not a strategy-learning task will be ill-posed depends on the objective function of P; similarly, whether or not it will have natural dynamics depends on the input variables of P. A strategy-learning system will need learning to predict future states only if a causal model of PS is not provided as background knowledge by E. Thus, the specific issues that a learning system must address

depend on the type of components a learning task has. With this in mind, we delve further into the anatomy of learning tasks in order to identify the characteristics of P, PS, and E that determine the pertinent issues of a given learning task.

We now introduce several strategy-learning tasks drawn from diverse domains. We use these as running examples throughout this chapter to characterize strategy-learning tasks, techniques, and architectures.

Example 1. *Learning Strategies for Load Balancing. The performance task* P *in load balancing is to schedule jobs arriving at various sites of a distributed computer system. Each job can be run either locally at its site of origin, or remotely at another site with lighter load.* PS*, the problem solver, determines where to execute each incoming job, based on the job's characteristics as well as information about the past, current, and projected loads at various sites.* E*, the learning environment, provides feedback in the form of measured improvements in completion time due to load balancing relative to the case of local execution.*

A variety of problem formulations are possible, with varying degrees of realism. Abstract formulations assume that i) load at sites does not change outside the control of PS *(no natural dynamics), and ii)* E *provides background knowledge relating scheduling decisions with measured performance. More realistic formulations drop both of these assumptions.* □

Example 2. *Learning to Steer a Ship. The performance task* P *is to steer a ship in a variety of (simulated) training scenarios, eventually getting as close as possible to a specified target position, which may be different for different scenarios. A scenario is described by the current coordinates of the ship relative to the target as well as current velocity vectors. Using these values, the problem solver* PS *must determine the direction and amount of turn at each decision point. In contrast to load balancing, the learning environment* E *provides* PS *with rudimentary operator-application rules and some general domain knowledge.* □

Example 3. *Learning to Balance a Pole. The controller* PS *aims to balance a pole fixed at one end to a mobile cart with one degree of freedom. At any decision point, it can move the cart left or right by applying a fixed amount of force. The cart is mounted on a rail of fixed length and is constrained to not go off the ends of the rail.*

This performance task P *exemplifies the class of* control problems *[16, 26]. Other examples in this class*

include: i) regulation, *in which the objective is to keep the external environment close to a ‘‘desired state,’’ ii)* tracking, *in which the objective is to make certain decision variables follow the same sequence as their corresponding problem variables, iii)* optimal-path problems, *in which the objective is to get the external environment in a desired state at a desired time, and iv)* minimum-time optimal control, *in which the objective is to get the external environment in a desired state in minimum time.*

Control problems can be knowledge-intensive or knowledge-lean; the latter are classified under adaptive control. Knowledge-intensive versions assume well-defined objective functions and complete knowledge of environmental changes (modulo noise) resulting from operator application; these tend to use static strategies called open-loop control. Knowledge-lean versions assume measurability over instants (in agreement with the assumption of the Markovian property, which is considered a defining trait of control problems); these tend to use dynamic strategies also known as closed-loop control. □

Example 4. *Learning Strategies for the Towers of Hanoi Problem. The scenario consists of three towers, of which the first is surrounded by rings whose diameter decreases from bottom to top. The objective is to get all the rings on the third tower in the same order. The only operator available is one of lifting a ring from the top of one tower and placing it on top of another. The constraints are that a ring can never be placed over a smaller ring.*

This performance task exemplifies the class of symbolic problems *that have been studied in cognitive science and AI [32, 131]. These problems feature input variables that do not vary outside the control of the PS.* □

1) Performance Task

A performance task P can be specified in terms of its objectives, constraints, and variables (Table XI). Objectives may be specified either explicitly as functions to be optimized, or implicitly using scalar metrics of solution quality; the former are called well-defined objectives and the latter, ill-defined. Similarly, constraints may be defined either explicitly as truth-valued expressions of variables, or implicitly using scalar metrics of solution feasibility; the former are called well-defined constraints and the latter, ill-defined.

Table XI
Components of a performance task

Component	Description
Variables	Represent observed state (*primitive problem variables*), abstract state descriptions (*abstract problem variables*), and the controls activated, or operators instantiated, by the problem solver (*decision variables*).
Objectives	Represent goals of problem solving and, therefore, strategy learning. They are defined as either functions to be optimized or conditions to be achieved by solutions (assignments of values to decision variables).
Constraints	Define the space of feasible solutions over which the application problem can be solved. Together with objective functions, these constitute a performance standard against which strategies may be evaluated.

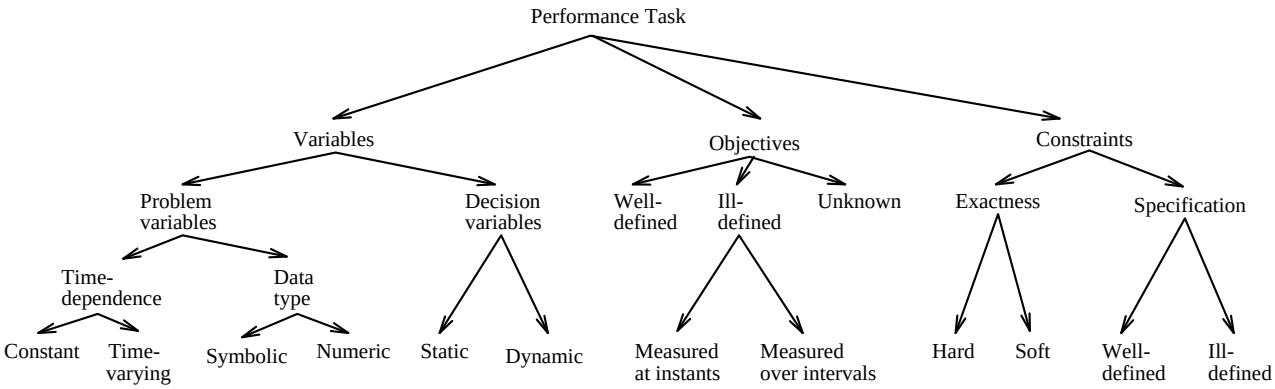


Figure 38. Attributes for classifying performance tasks

Variables are used to either represent or influence the external environment's state. They and their respective values constitute the (internal) state of the problem solver. The space of all possible assignments of values to variables is called the *state space*. An assignment of values to the set of decision variables is called a *solution*. States satisfying all the constraints of a performance task define the space of *feasible solutions*. For optimization objectives, the feasible states that satisfy the optimization criterion define the set of *optimal solutions*. In general, states satisfying both the objectives and constraints are called *goal states*. Sometimes, the term solution also denotes a state-space path from the initial state for a given instance to some goal state of that instance.

The objectives and constraints of a problem define measures of quality and feasibility, respectively, on its state space. Together, they constitute a *performance standard* [45] against which strategies for a given performance task may be judged. Problem solvers may add new constraints (such as deadlines or limits on memory usage), or may refine the objective function by trading between quality and complexity of solutions. Even so, the objectives of performance tasks are the prime drivers of problem solving and strategy learning.

Various properties of variables, objectives, and constraints determine the complexity of strategy learning; these are useful attributes for classifying strategy-learning tasks (Figure 38).

Example 1 (contd.). *Learning Strategies for Load Balancing. Variables of the load-balancing problem include* status variables *(utilization of site-specific resources, such as CPU and memory, and shared resources, such as disk devices and interconnection network),* job descriptors *(absolute and/or relative amounts of service required by the job from various resources), and* placement variables *(where the current job will be sent for execution).*

A variety of objective functions are possible: maximizing the number of jobs completed per unit time (throughput), minimizing the average completion time of all jobs in the system, minimizing the average completion time of jobs at each site, and minimizing the completion time for each job independent of other jobs. All these objectives are functions of completion time. In the absence of background knowledge about the relationship between variables and completion time, these objective functions are ill-defined. Certain other objectives, such as equalizing the load across all sites, are well-defined when ''load'' is a measured quantity. However, when status variables are the only problem variables, even the equalization of load is an ill-defined objective.

Constraints may include precedence constraints *(a job cannot start unless certain others complete execution),* size constraints *(a job may require a site with a certain minimum amount of physical memory), and* placement constraints *(for instance, a site may not allow more than one foreign job to execute at any instant). These are examples of well-defined constraints. Others, such as a requirement that communicating jobs should preferably be located at the same site, are ill-defined if the communication behavior of jobs is not known beforehand.* □

Variables. The variables of a performance task represent either quantities used by the problem solver to represent states of the external environment, or quantities set by it to control the environment. The

former are called *problem variables*, and the latter, *decision variables*; they serve, respectively, as the inputs and outputs of the problem solver.

Example 1 (contd.). *Learning Strategies for Load Balancing. Status variables and job descriptors serve as problem variables of load balancing; placements, as decision variables.* □

Problem Variables. These are used for specifying problem instances as well as for representing states of the external environment. While some of these represent primitive natural aspects of the external environment, others are abstract quantities derived from the primitive ones by operations such as aggregation and smoothing. While primitive variables are easily measured, and therefore considered given, derived quantities are sometimes costly to estimate. For example, status variables and job descriptors of load balancing are primitive variables, whereas ‘‘load’’ is an abstract quantity derived from these.

A strategy-learning system, having seen only training instances, must generalize from experience in order to solve new unseen instances. It can do so only if it can detect and use some similarity between old and new instances. Since instances are defined using problem variables, similarity between different instances is defined by the similarity of their assignments of values to problem variables. Properties of problem variables that affect strategy learning are as follows.

a) *Temporal dependence.* When problem variables vary with time, outside a problem solver’s control, strategies need to track their values in order to produce actions that will lead to optimal or near-optimal external states. Without such temporal dependence, the external state varies only under the problem solver’s control; future states are relatively easy to predict, enumerate, and evaluate. With temporal dependence, concerns such as the rate of sampling (for primitive problem variables) and the interval of aggregation (for aggregated problem variables) become significant issues.

b) *Type of Values.* Problem variables can be either numeric or symbolic. Symbolic variables can have internal structure and assume discrete values. Unlike numerical values, symbolic values are difficult to generalize; hence, learning is generally more complex. On the other hand, symbolic values are

amenable to enumerative search techniques and associative learning algorithms, which use efficient tabular representation of strategies.

c) *Measurability.* While primitive variables are easily measured, abstract and derived ones are often estimated, especially when they involve time-varying quantities. Derived quantities such as proportions require large amounts of data for accurate measurement; others, such as moving averages, delay the input to the problem solver. On the other hand, derived quantities simplify strategies by reducing the number of potential variables.

Example 1 (contd.). *Learning Strategies for Load balancing. Status variables and job descriptors are the problem variables. While status variables are numeric time-varying, job descriptors are usually numeric constants.* □

Decision Variables. The goals of problem solving are specified as functions and formulae of decision variables. A problem solver *instantiates* (assigns values to) decision variables during decision making. The final assignment of values constitutes a solution state. Frequently, decision variables represent actions and their arguments. Certain actions affect future inputs (the values of problem variables) to a problem solver while others do not; the former are called dynamic decision variables, and the latter, static.

a) *Static Decision Variables.* When decision making does not affect the state of the external environment, future values of problem variables are not affected by the current decisions. Temporal dependence is the only source of variation for such problems (called *static decision problems*). With static decision variables, the context of strategy learning can be limited to one decision point at a time.

b) *Dynamic Decision Variables.* When affected by decision making, the external state evolves under the combined influence of both its own natural dynamics and the problem solver's actions. Problems with dynamic decision variables are called *dynamic decision problems*. With dynamic variables, complex interactions among decisions need to be considered, resulting in complex strategy-learning tasks.

Example 1 (contd.). *Learning Strategies for Load Balancing. Usually, the* PS *employs a* dynamic load-balancing *strategy that decides placements based on values of status variables. When a job is migrated from one site to another, its net effects are reduced load on resources local to the originating site and increased load on resources local to the remote site. Therefore, status variables evolve under the effect of placements, making this a dynamic decision problem.*

However, there exist formulations of this problem in which all the jobs to be placed are available at the outset, and placements do not depend on status variables. Such strategies are appropriately called static load-balancing strategies. *In these, job descriptors are the only problem variables and do not evolve under the effect of placements, making this a static decision problem. The context of strategy learning includes only one decision point: the one at the initial state, although the decision of simultaneously placing all the jobs tends to be complex.* □

Objectives. Figure 38 shows that varying degrees of precision are possible in specifying the objectives of a performance task. Traditionally, problem solvers have attempted only performance tasks for which a closed-form objective function is specified. Realistic applications frequently involve ill-posed tasks whose objective functions are either unknown or empirically determined. Strategy-learning tasks can be classified according to the degree of well-posedness of their objective functions as follows.

Well-defined Objectives. Several performance tasks have implicit or explicit well-defined objectives. Examples include special-purpose problem solvers, such as classifiers, and general-purpose problem solvers, such as planners. Sometimes a well-defined objective function is built into the problem-solving strategy, and some parameters of such a strategy may be made available for tuning via strategy learning.

Example 1 (contd.). *Learning Strategies for Load Balancing. Objectives such as equalizing the load across all sites are well-defined provided 'load' itself is a well-defined entity. In abstract formulations of the problem, popular with queueing theorists, CPU is the only resource of contention, and the level of CPU utilization is considered the sole determinant of the overall load. The objective is well defined because load can be defined easily in terms of the available inputs (CPU utilization).* □

Ill-defined Objectives. In this case, a closed-form functional specification of the objective is unavailable, but states can be evaluated either individually or collectively. The major difference with the well-defined case is that in order to get evaluated, a state must occur during a problem-solving episode. Therefore, the quality of solution can be measured *a posteriori*. For these problems, it is essential to develop some models or learning methods so that alternative strategies can be compared.

An objective function may be ill-defined if it is an unknown but measurable function of either known or unknown variables. Ill-defined objective functions may evaluate states either individually or collectively. Two cases can be distinguished.

a) *Instantaneous Measurability.* This implies that states are evaluated individually, and the objective-function value of a solution can be decomposed with respect to any state. That is, the path from the initial state to a chosen state can be optimized independent of the path from the chosen state to a goal state. One can induce an evaluation function using statistical regression techniques; such a function can then be used in place of a well-defined objective function. The case involving unknown problem variables is more complex, requiring testing of possible variables for inclusion and estimation of the objective function.

b) *Interval Measurability.* This implies that evaluations are not always available for every state and, when available, represent the collective evaluation of a sequence of successive states. The evaluation of a state depends upon the path to it from the initial state. As a result, evaluations must be sensitive not only to the current state but also to some past states. Two types of evaluation-function models are possible: i) those using a finite *window* of past states; and ii) those modeling evaluations as recurrences on state space.

Example 1 (contd.). *Learning Strategies for Load Balancing. Usually, the objectives of load balancing are defined in terms of job completion times. Without extensive knowledge of hardware architecture and the operating system's scheduling policy, completion-time-based objective functions are ill-defined. Automata theorists [128] choose to*

formulate completion time as a function of only the current values of status variables; this results in an ill-defined objective function measurable at instants. In this case, a deterministic evaluation function can be learned by regressing job completion time onto the values of status variables at the time of starting the job.

In reality, the completion time of a job depends on past, current, and future values of status variables [115], which makes it an ill-defined function measurable over intervals. If window-based methods are used, then the number of past status variables to consider needs to be determined so that regression is kept simple. On the other hand, if recurrent models are assumed, then the coefficients of the model must be determined using the relatively complex time-series regression techniques. □

Unknown Objectives. When no objective function is given and no measurements are available, the problem solver may use some prior knowledge to induce an objective function either analytically or empirically, thus reducing this case to one of the previous cases. This problem boils down to one of proposing a set of objectives and asking the user to select one subjectively. Alternatively, additional constraints may be imposed on the original problem; the problem becomes one of finding any feasible solution in the constrained problem space.

Example 1 (contd.). *Learning Strategies for Load Balancing. A number of alternative objectives are possible for the load-balancing problem. For instance, if throughput (number of jobs processed per unit time) is to be maximized, then it does not matter whether or not specific jobs complete sooner with load balancing than without it. This objective is valid for users who submit jobs in batches (groups), and only the performance of the batch is relevant. On the other hand, when scheduling independent jobs, it may be important to have a high probability of speeding up each individual job. In this case, good performance on large jobs cannot compensate for poor performance on small jobs. A learning system starting without knowing a specific objective must compare alternative proposals and choose the one whose predicted value matches the actual feedback from the user.* □

Constraints. Constraints are used for specifying a space of feasible solutions over which optimization is performed. They may either be explicitly specified, or be built into the problem solver's strategy, or be part of the objective function. Constraints can be classified on the basis of their exactness (as hard or soft), as well as their specification (as well-defined or ill-defined) (Figure 38).

Hard versus Soft Constraints. Hard constraints impose sharp boundaries on the state space, demarcating feasible solutions from infeasible ones. They curtail syntactic generalization (syntactically similar problems having similar solutions) and are usually enforced by a *move generator* in the problem solver. Problems with hard constraints are sometimes solved by first solving relaxed versions of the original problems in order to obtain an approximate solution, which is then used as an initial state for an exact solution. On the other hand, problems with soft constraints are associated with a large space of feasible solutions. These constraints are usually transformed into penalty terms that are added to the objective function.

Well-defined versus Ill-defined Constraints. Well-defined constraints are defined as truth-valued functions of a performance task's variables; hence, one can determine whether a solution is feasible or not by testing whether it satisfies the constraints. In contrast, ill-defined constraints are either unknown or too complex to be modeled as functions of problem and decision variables. As a result, the constraints cannot be formulated as truth-valued functions. A common solution is to induce a model of the feasible region from empirical observation of feasibility and infeasibility of various states. The model (like soft constraints) may then be incorporated as penalty terms in the objective function. In some cases, ill-defined constraints become well-defined during the course of problem solving and are incorporated in the problem solver. For instance, if a robot is attempting to fit a projection on one part into a slot in another, and it does not have geometric models of the two parts, then the only way it can discover infeasible moves is by testing whether the move is feasible and by hypothesizing a model of infeasible regions. If the robot knew that all slots are square in shape and aligned parallel to some known coordinate system, then it may find out the exact coordinates of the slot after a few trials.

Example 1 (contd.). *Learning Strategies for Load Balancing. Graph theoretic formulations of load balancing take into account explicit precedence and placement constraints, which are hard well-defined constraints. System engineers, on the other hand, either ignore these dependences or enforce them procedurally in the problem solver. Size constraints in load balancing are examples of soft ill-defined constraints because the memory requirements of a*

Table XII
Examples of performance tasks in strategy learning

Example	Performance Task P	Comments
Load Balancing	To schedule incoming jobs	Problem variables (indicators of background workload, job characteristics) are numeric and time-varying; decision variables (placements), dynamic.
Ship Steering	To decide when and how much to turn	Problem variables (position and velocity) are time-varying; and the decision variables (amount of turn), dynamic. The objective function (navigating the ship along a trajectory) is ill-defined, measurable over intervals.
Pole Balancing	To balance an inverted pendulum	The problem variables (the pole's angle of inclination and angular velocity, and the cart's position and linear velocity) are numeric time-varying; and the decision variables (direction of applied force), dynamic. The objective is ill-defined, measurable over instants.
Towers of Hanoi	To achieve a desired configuration of disks on the towers	The problem variables (specifying which of the three towers each disk is on) are symbolic and constant; and the decision variables (which disk to move where), dynamic. The objective (to achieve a desired configuration) is well-defined; and the constraints (no disk can be placed over another of smaller diameter), exact.

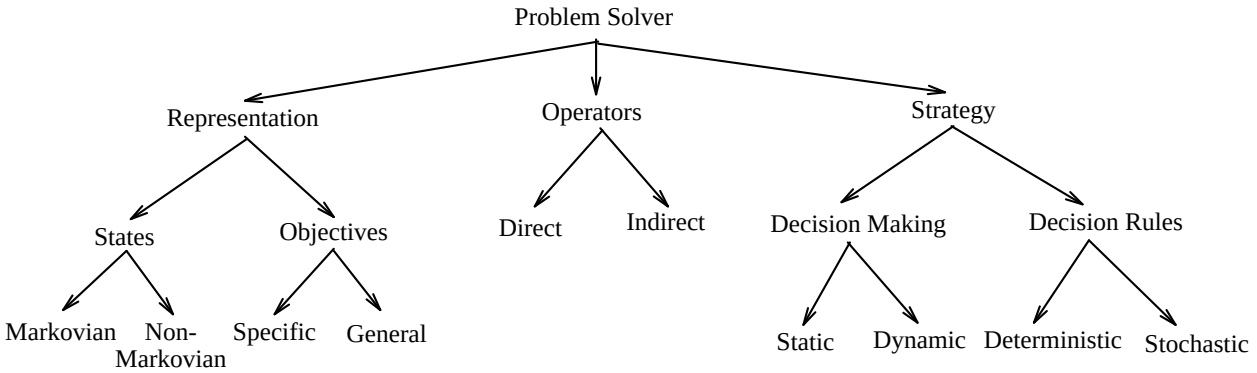


Figure 39. Components of problem solvers and their properties

---

job are usually data-dependent and cannot be modeled by closed-form formulae. Too large a job for a small physical memory causes thrashing, which causes only a gradual degradation in performance. Thus, there is no sharp divide between feasible and infeasible states. Because of this 'softness,' such constraints are often expressed as preferences, thus becoming part of the objective function. □

Table XII illustrates the structure of P for the four strategy-learning tasks described earlier.

2) Problem Solver

A problem solver employs a *representation scheme* to internally represent problems and solutions; it also has a repertoire of *operators* which it uses to transform initial states into goal states. (A sequence or a partial order on the set of operators constitutes a solution.) The problem solver uses parameters that may be tuned via learning as well as others that are constrained in its design. In this section, we identify the aspects of problem solvers' representations, operators, and strategies that influence the design of strategy-learning systems. The components of problem solvers and their properties are shown in Figure 39.

Representation. States and objective functions are significant components of strategy-learning tasks; their representation determines the type of strategy-learning technique used. Following are the key properties of states and objective functions that affect strategy learning.

Markovian versus Non-Markovian State Space. If each state carries enough information to permit optimal decision making without considering past states or past decisions, then the state space is called Markovian. For such state spaces, past states and past decisions do not influence current decisions. Otherwise, a history of past states and/or past decisions needs to be retained for use in making future decisions. Often in the former case, large state vectors may be needed to capture all the useful information in one state. When the number of problem variables is large, one may choose to track only a few significant variables rather than retain the entire state vector. Such representations are inherently non-Markovian.

General versus Specific Objectives. A problem-solver may be either *general-purpose* or *specialized*, depending on whether it can represent and solve a class of problems rather than a single problem. General problem solvers need to index their strategies by the structure and the content of their objective functions. Unlike specialized problem solvers, their ability to represent and manipulate explicitly represented objectives is paramount in this case. Specialized problem solvers are common in the literature on control [26], whereas generalized problem solvers are common in the literature on planning and problem solving

[151]. Strategy-learning systems for general-purpose problem solving face the problem of *generalizing across tasks*; that is, the experience from one problem may need to be generalized to a different problem. For specialized problem solvers, strategy learning merely warrants *generalizing across instances* of the same problem.

Direct versus indirect operators. Problem solvers transform their internal and external states by the application of operators. Operators can be either direct or indirect. *Direct operators* act independently on the objective function without interfering with other instances of operator application. The effects of direct operators combine in simple ways, satisfying criteria such as additivity and superposition [34]. On the other hand, the effects of *indirect operators* combine in complex ways, such as through causal chains or AND-OR graphs [144]. In this sense, strategies involving indirect operators have more 'structure.' Numerous examples exist, especially in planning [156], of problems that require such indirect problem-solving capability. Strategies using indirect operators require complex methods of generation.

Problem-solving strategies. A problem solver should have an internal bias towards more preferable problem-solving strategies. A problem solver without any preference would be horribly inefficient for most practical applications, despite its tremendous generality. Another reason for bias is that expert knowledge may be available only in the form of time-tested procedures. For efficiency and practicality, such procedures should be built into problem solvers as skeletal strategies, which are further refined and extended by strategy learning. In addition to the bias, problem-solving strategies can be classified on the basis of their stochastic and dynamic behavior.

Deterministic versus Stochastic Strategies. A deterministic strategy always recommends the same operator for an external state in a given context, no matter how many times this state recurs. Stochastic strategies, on the other hand, explore multiple alternative decisions at those decision points where a choice cannot be made *a priori*. Extra knowledge for guiding such a search may take the form of rules for assessing either the probability that a particular alternative will be explored, or the order in which alterna-

tives will be explored. Stochastic strategies have the potential for producing better solutions at the cost of extra computations.

Static versus Dynamic Strategies. While certain strategies make decisions based on the specific state associated with each decision point, others recommend a series of decisions based on just the initial state. The former are called dynamic strategies; and the latter, static strategies. Static strategies are faster but inappropriate for problems whose variables cannot be predicted accurately or efficiently a priori. Designing static strategies requires complete and accurate model of the external environment; dynamic strategies, because they make decisions about one or a few actions, require less prior knowledge and are simpler to design. However, dynamic strategies are computationally more expensive to apply than static ones because they warrant run-time information gathering and inference.

Strategies generated by conventional planning methods of AI [65] are static, whereas those generated by conventional dynamic programming methods of control and optimization [21] are dynamic. There exist numerous variants of conventional planning that use dynamic strategies; these are called reactive planning methods [2, 66, 91, 111, 145, 162] in AI.

Example 1 (contd.). *Learning Strategies for Load Balancing. Two types of problem solvers represent, respectively, the analytical and software-based approaches to the load-balancing problem. Both are designed for specific objectives and employ Markovian representations, using only the current state in making load-balancing decisions. Because the status variables do not directly permit such representation, these problem solvers use 'load indices,' which are abstract variables computed as moving averages of computation load. Often, the coefficients of moving averages are chosen ad hoc. An alternative (considered in Chapter IV) is to employ non-Markovian representations and* learn *the relative importance of past and current values of status variables for predicting speed-ups of tasks.*

The problem solvers designed for handling precedence constraints often employ indirect operators *in which a number of different operators contribute to the reward state(s) of the external environment. Solutions in this case are said to be* structured. *Others generally use the simpler direct operators such as sending a job to a remote site and accepting a job sent by a remote site. Analytical solutions often employ stochastic strategies [27, 62, 128],*

Table XIII
Examples of problem solvers in strategy-learning tasks

Example	Problem Solver PS	Comments
Load Balancing	Process migration software	Representation of states is non-Markovian; of objectives, specific. The operators (process migration) are direct. Practical strategies employ dynamic decision making and either deterministic or stochastic decision rules.
Ship Steering	A navigation program	Representation of states is non-Markovian; of objectives, general. The operators (turning actions) are direct. The strategy is static, and the decision rules, deterministic.
Pole Balancing	A controller to apply a fixed force left or right	The controller employs Markovian representations for states, and works for a specific objective. The operators (applying the force) are direct. The controller uses stochastic strategies.
Towers of Hanoi	A program for partitioning objectives and for searching among moves	The problem solver employs a Markovian representation of states and is able to handle general-purpose goals because multiple goals are generated for each problem instance. Its operators (moving disks) are indirect, and its strategies, deterministic and static.

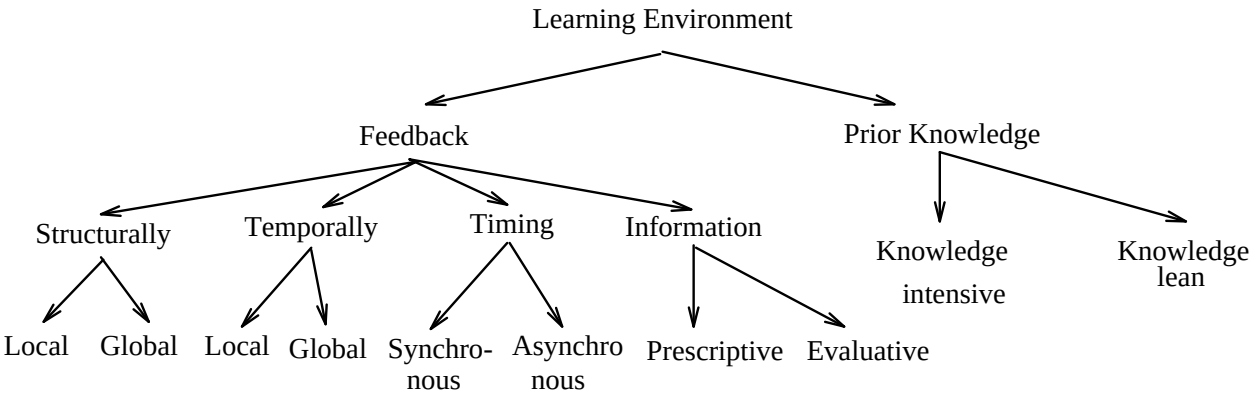


Figure 40. Components of a learning environment

although deterministic strategies are more common [17, 108].

Dynamic load balancing *strategies take into account the* load average *at each site in deciding where to send the next job, whereas static strategies schedule jobs according to a predetermined criterion unaffected by the dynamic variations of workloads. While static strategies avoid frequent communication of status information among sites, they fail to exploit dynamic imbalances in load whose occurrence cannot be predicted ahead of time.* □

Table XIII illustrates the structure of PS for the four strategy-learning tasks described earlier.

3) Learning Environment

The interface between a strategy-learning system and the external world is called the learning environment. Environments that generate feedback are called *reactive environments*. These include human trainers and/or programs that generate feedback for the learning system, as well as sources external to the learning system that provide prior knowledge relating problem variables and decision variables to feedback. If the environment is not reactive but the objective function is measurable (Figure 36), then the measurements can be used in feedback generation. In this case, certain additional issues such as the standard-of-comparison problem (Section C) need to be addressed. The components of a learning environment are shown in Figure 40.

Feedback. The nature of feedback is the single most important determinant of the form and complexity of strategy-learning algorithms. Feedback may take many forms: either a corrective error signal or a scalar evaluation signal, covering either one or more decision points, and generated either periodically or intermittently. For well-defined and measurable objective functions, feedback can be generated internally in the learning system even when little or no external feedback is available. We now consider the impact of the type of feedback on strategy learning.

Prescriptive versus Evaluative Feedback. Feedback signals carry certain explicit and implicit information useful for altering the behavior of a problem solver. Signals that carry more explicit information require simpler learning rules but a more informed source than those that carry more implicit information. Based on the amount of explicit information, one can distinguish between prescriptive and evaluative feedback.

Prescriptive feedback carries explicit information about the desired operators and/or states; the learning system can use it for computing an error signal to be minimized via strategy modification. However, generating such feedback requires a *teacher* who knows what the correct outcome should be. Learning from a teacher is called *supervised learning* [80].

Evaluative feedback carries only implicit information about the desired behavior but explicit evaluation of the observed behavior. Such feedback is intrinsically *a posteriori*, being measured or generated after the behavior has occurred. It requires only a *critic* [197] who has some prior knowledge of the objective function and can assess the goodness of external states or sequences thereof. Scalar evaluative feedback signals are called *reinforcements* [125] and learning from such signals, *reinforcement learning*.

Structural Locality of Feedback. The goal of learning is to modify either the decision rules or certain parameters thereof. Many rules or parameters may be involved at each decision point; these may be evaluated either individually or collectively: if individually, feedback is termed structurally local; otherwise, global. Usually, feedback is structurally global (the external environment evaluates decisions but not individual rules), and the evaluation of individual rules is left to the learning system. Translation of structurally global feedback into a structurally local one is called *structural credit assignment* [173]. In one case, some component (a human or a program) isolates the individual effects of various rules and feeds the information to the problem solver; in another, the rules operate on independent aspects of a problem, each associated with its own feedback signal.

Temporal Locality of Feedback. The environment produces feedback in response to the problem solver's decisions. Feedback may evaluate decisions either individually or collectively: if collectively, several decision points may elapse before feedback becomes available. Such feedback is called *delayed feedback*; it contains the combined evaluation of several decisions. Delayed feedback explicitly evaluates the current state and implicitly evaluates past states and decisions, especially for dynamic decision problems. Based on the temporal properties of feedback, one can distinguish between temporally local and temporally global feedback.

Temporally local feedback applies to decisions individually. In solving a large and complex problem, a problem solver may make many decisions; temporal locality requires that the environment should produce an explicit reaction to every decision. The burden of disentangling the interdependences among decisions is on the environment rather than on the learning system. Therefore, systems that learn from

temporally local feedback are easier to design than the ones that learn from temporally global feedback.

Temporally global feedback applies to decisions collectively. Resolution of interdependence between decisions shifts from the environment to the learning system, which must distribute the feedback between decisions using knowledge of cause-effect relationships between decisions and feedback. Translation of temporally global feedback into temporally local feedback is called *temporal credit assignment* [173].

Synchronous versus Asynchronous Feedback. The timing of feedback dictates the ease with which a learning system can partition its data into *episodes* or *trials*. When feedback occurs at predictable stages of problem solving, learning is episodic. Both the structural and temporal credit-assignment problems discussed above are simplified when feedback distribution is confined within episode boundaries. Such feedback is called synchronous. When the time of occurrence of feedback is not easily predictable, the learning system is responsible for bounding the scope of feedback distribution. Only a finite number of past decisions and/or states can be retained between successive occurrences of feedback, and the learning system is responsible for their storage and temporal credit assignment.

Prior knowledge. A strategy-learning system performs credit assignment using a *world model* to capture the relationships among states, decisions, and feedback signals. When such knowledge is not given, it must somehow be inferred before performing strategy modification via credit assignment. Environments that provide an explicit world model to the learning system are called knowledge-rich; those that do not, knowledge-lean.

In knowledge-rich environments, credit assignment is a deductive process. The learning system can explicitly construct proof trees or other computational structures relating decisions and feedback. In knowledge-lean environments, the learner is forced to induce a world model from the states observed between making a decision and receiving feedback. Interleaving such induction with problem solving and strategy modification places extra burden on the learning system.

Table XIV
Examples of environments for strategy learning

Example	Learning Environment E	Comments
Load Balancing	Measurements and models of completion time	Feedback is a function of measured completion time. It is delayed, evaluative, asynchronous, and structurally global. Effect of placement decisions on the external state is unknown, as well as models of tasks and their inter-arrival times.
Ship Steering	Measurement of target displacement and models of steering actions	Feedback (error between actual and desired trajectories) is structurally global, delayed, evaluative, and synchronous. Prior knowledge of the effects of steering actions, as well as the natural dynamics (based on momentum and acceleration), are available as closed-form rules.
Pole Balancing	Sensors to detect a fallen pole	Feedback (signal indicating a fallen pole) is delayed, evaluative, structurally global, and asynchronous. Dynamics of the system can be made available as a reference in training.
Towers of Hanoi	Knowledge of a ''desired state'' and effects of various moves	Precise knowledge of the effect of each move on the external state is available a priori. Prior knowledge for partitioning the objective function is also available. Feedback is structurally global, delayed, evaluative, and synchronous.

In knowledge-rich environments, the uncertainty about the future states, given the current state and action, is much less than that in knowledge-lean environments. Strategy-learning tasks in such environments, therefore, prefer learning the faster static strategies rather than the slower but more robust dynamic strategies.

Example 1 (contd.). *Learning Strategies for Load Balancing. The feedback to the learning system depends on the formulation adopted. Analytical formulations based on combinatorial search are common for scheduling task graphs under zero natural dynamics for load; here, multiple alternatives can be evaluated and prescriptive feedback generated. In the case of empirical formulations, feedback is evaluative. Depending on the objective function, feedback may be available either after the completion of each job or after the completion of a batch of jobs.*

The dependence of feedback on completion time introduces a delay between the occurrence of a decision and the arrival of feedback signal(s) evaluating it. In the interim, several other jobs may have arrived and been scheduled. This makes the feedback signal structurally and temporally global. Feedback is asynchronous because the time to complete a job (and generate the feedback) cannot be predicted.

Apart from systems that use the scheduling formulation, load-balancing software is usually knowledge-lean. The effects of migrating jobs to remote sites cannot be predicted precisely. □

Table XIV illustrates the concepts of this section on the strategy-learning tasks described in Section B.

C. Issues in strategy learning

Certain key issues are common to a large number of strategy-learning tasks, and their relevance depends upon the type of task given. For instance, prediction and temporal credit assignment are relevant to tasks having time-varying problem variables. Identification of such issues will allow us to abstract task-specific details. Further, an issue-based treatment of strategy learning will allow us to consider those techniques that are conventionally not used in strategy learning but are nevertheless appropriate for addressing these common issues. Each of the following subsections discusses a key issue in strategy learning as well as the relevant properties of the strategy-learning tasks. Section D surveys various approaches to some of the issues discussed in this section. Table XV lists these issues, characterizes the factors governing their relevance, and provides pointers to the pertinent approach(es).

1) Ill-posedness of objectives

For learning tasks with ill-defined objective functions, the goals of learning and problem solving are not clear at the outset; they must be inferred using either prior knowledge or goal-related information implicit in the feedback. For tasks with non-reactive learning environments but measurable and ill-defined objectives (class 8 of Figure 36), feedback must be generated internally by the learning system. The following issues arise in learning strategies for performance tasks with ill-defined objective functions.

a) Standard-of-Comparison Problem. This problem, first recognized by Ackley [1], concerns the method for assessing feedback. When the objective function is ill-defined, it is difficult to assess the solution quality, and alternative operators can only be evaluated relative to each other. In environments hav-

Table XV
Issues, tasks, and approaches of strategy learning

Issue		Characteristics of learning task	Approach
Ill–posedness of objectives	Standard-of -comparison problem	Ill-defined objective functions and evaluative feedback	Learning from relative evaluation
	Objective-function learning	ill-defined but measurable objective functions	Statistical regression when objective function is ill-defined, measured over instants; time-series regression, when measured over intervals
	Learning while searching	Stochastic strategies, knowledge-lean learning environment, and asynchronous feedback	Learning models of reward-generation mechanisms and learning to predict improvements; must address trade-off between exploration and convergence
Credit assignment	Structural	Structurally global feedback	Error-reducing approaches; problem-solver-specific implementations
	Temporal	Temporally global (delayed) feedback	Complex general solutions, requiring reasoning with causal and temporal models; reduced complexity due to the Markovian property
Prediction	Predicting future states	Time-varying problem variables and/or dynamic decision variables, especially difficult in knowledge-lean environments	Approaches based on projection and time-series analysis
	Predicting future feedback	Temporally global (delayed) evaluative feedback and/or ill-defined objective functions measurable over intervals	Modeling of the feedback-generation mechanism and approaches based on projection and time-series analysis
Violation of the Markovian Property	Using past states and decisions in current decision	Non-Markovian representation of states in the problem solver, or lagged correlations in time-varying problem variables	Explicit storage of past decisions and explicit modeling of inter-decision relationships for temporal credit assignment
Constraint handling		Exact well-defined constraints on solutions produced by strategies learned	Constraint satisfaction and constraint incorporation
Storing past decisions		Non-Markovian representation of states in the problem solver, especially important with delayed feedback	Either on-line incorporation of feedback, or temporary storage of episodes; methods for limiting the size of temporal scopes
Managing multiple objectives		Learning strategies for general problem solving; complexity of learning depends on the size of targeted problem class	Symbolic representation of problem spaces and general-purpose learning techniques
Dynamic decision making		Dynamic strategies and dynamic decision variables; complexity increased by time-varying problem variables, violation of Markov property, and lack of prior knowledge of causal models	Dynamic programming and its variants used for Markovian representations; no general solutions known for the non-Markovian case
Handling structured solutions		Indirect operators in the problem solver (usually with static strategies); especially complex in knowledge-lean environments	Preference for static strategies that consider multiple decisions at the same time; explicit representation of structured solutions during credit assignment
Controlling nondeterminism	Large search spaces for each instance prohibit search of multiple alternatives	The small number of operators in knowledge-lean environments and the generality of operator preconditions in knowledge-intensive environments; indirect nature of operators (thereby, the depth of solutions)	Preference for stochastic strategies that elegantly represent varying amounts of nondeterminism as randomness in search; learning while searching to limit episode size

ing prescriptive feedback, the standard-of-comparison problem is relegated outside the learning system to the teacher. However, for evaluative feedback, it must be addressed by the learning system.

Example 1 (contd.). *Learning Strategies for Load Balancing. To solve the problem of choosing a standard of comparison in the evaluation of load-balancing strategies, one approach is to use the case with no load balancing as a point of reference. However, this requires that two sets of experiments should be performed using exactly the same set of jobs and loading conditions: once with, and once without, load balancing.* □

b) Learning an Objective Function. When an objective function is ill-posed, information about the goals of problem solving is implicit in the feedback associated with each state. Note that the feedback is generated only *after* the state has been traversed. Two cases need to be considered. First, when states can be evaluated independently and the Markovian property holds, one can tabulate evaluations as <state,feedback> pairs. The problem of making the learning task well-posed then reduces to one of fitting a function to the tabulated data. Second, when the objective-function value depends on several states or when it is measurable only over intervals, states cannot be evaluated independently. In order to learn objective functions from such data, one can regress either a simple function upon the current and past values of problem variables, or an autoregressive (recursively defined) function on just the current values.

Example 1 (contd.). *Learning Strategies for Load Balancing. Completion time is not a well-defined function of status variables alone, and job descriptors are rarely available at the time of decision making. Not only are the completion-time based objective functions ill-defined, but also they can be evaluated only over intervals (corresponding to the duration of job execution). When queueing models of computers and jobs are available and applicable, objective functions are well-defined, and the queue sizes on various resources of contention [60, 204] can be shown to be useful for predicting completion time. In practice, the applicability of these models is questionable, and objective functions should be learned from data relating the completion times of jobs to the load conditions at the time of decision making.* □

c) Learning while Searching. This is also known as within-trial learning: the nondeterminism in later stages of search is reduced using information about states visited in earlier stages. It is especially

useful for learning stochastic strategies which, depending on the amount of prior knowledge available, may start with a random search and eventually converge to an almost deterministic search [14]. Learning while searching is also useful for problem solvers employing deterministic strategies to expand large search spaces [157, 158]. For learning with asynchronous feedback, which precludes identification of clean learning episodes, as well as for learning in changing environments, learning while searching is the only tractable way to learn. Learning while searching requires the learning algorithm to have low overhead because learning occurs at each decision point.

Example 3 (contd.). *Learning to Balance a Pole. As the learning system gets better at balancing, the episodes become longer and no external feedback is generated. In order to limit memory usage, the learning system must learn within trials. One way to do this is for the learning system to predict future feedback, and use its own predictions in place of the real feedback. Whenever external feedback does become available, it can be used to train the internal feedback-generation mechanism.* □

2) Credit Assignment

Translation of environmental feedback (F) into strategy modifications is called credit assignment. It is a problem in inverse modeling of both the environment and the problem solver: the learning system attempts to determine what changes in the decision process will bring about desirable feedback on similar instances in the future. The division of the credit-assignment process into the temporal and structural subproblems is due to Sutton [173]; these two subproblems are described next.

a) Structural Credit Assignment (SCA). When the rules encoding a decision maker's strategy combine in predictable and well-defined ways at every decision point, one can associate a 'structure' with the decision process. For instance, a proof tree might be used for representing a symbolic decision process, as in explanation-based learning [43]. Structures such as proof trees are generated dynamically. Examples of static decision structures include feedforward neural networks, as in Anderson's pole-balancing system [5]. In either case, the goal of SCA is to translate the (temporally local but structurally global) feedback

associated with a decision point into modifications associated with various parameters of the decision process.

When a problem solver encodes its strategy using algorithms rather than rules, the 'structure' of the decision process is not obvious. Here, SCA must either be avoided entirely or be used from decision-process parameters to average feedback. For instance, certain problem solvers for load balancing [87] use parameterized decision procedures rather than a rule base [108] to represent their strategies. In these, SCA is used merely to associate the average completion time over a set of test jobs with every tested set of parameter values; such SCA procedures perform *selection* among alternative strategies rather than *modification* of an incumbent strategy.

When F is prescriptive and the strategy S used by PS deterministic, SCA is relatively easy: it involves reduction of the difference between the observed and the desired values of the decision variables. For instance, in problems of tracking and trajectory planning [143], as well as in the ship-steering task discussed earlier (Example 2), the desired final state is explicitly known. On the other hand, when S used by PS is stochastic, SCA involves estimation and optimization of the *probability* of producing the desired outcome. For instance, in load balancing (Example 1), if the learning system forms internal models of jobs using Markov chains [74], then it must adjust the transition probabilities of its model to match those of the job being executed.

When F is evaluative, it is not clear what the desired values of the decision variables should be, nor are the direction and magnitude of parameter modifications obvious. In this case, SCA requires assumptions about how the feedback signals evaluate the current strategy. SCA can be simplified when the decision process involves only a small number (say two or three) of discrete outcomes [140]. For instance, one might compare the current evaluation against a moving average of past evaluations in order to determine whether the probability of the observed outcome should be increased or decreased [1]: if the difference is favorable (positive for maximization of evaluation), the probability of producing the observed outcome is made closer to 1, and that of the remaining outcomes reduced accordingly; and vice versa.

SCA with evaluative feedback always involves search among alternatives. Stochastic strategies [14, 140] perform such search implicitly, and deterministic strategies [147], explicitly. At each decision point, the decision process chooses an operator to apply. When the decision process selects the operator producing the best evaluation, the values of decision parameters are treated as positive examples; otherwise, negative examples. SCA attempts to create decision regions in the space of problem variables so that each region has associated with it the best expected outcome for instances falling in that region. In general, there is a tradeoff between exploration (via search) and testing (repeatedly applying the decision process to new instances in order to gain confidence in the quality of the moves selected by the decision process).

b) Temporal Credit Assignment (TCA). TCA is the first stage in the assimilation of feedback and precedes SCA during learning. It divides up the feedback F between current and past decisions. If the state space is Markovian, then F can be modeled as a function of only the current state; as a result, recent decisions (which directly contribute to the current state) are more eligible for feedback than past decisions (which contribute only indirectly through intervening decisions and states). In particular, with direct operators (which independently cause the evaluation), it suffices to have a *discount factor* (less than 1) [175] for determining the relative importance of each decision with respect to its immediate predecessors.

Example 3 (contd.). *Learning to Balance a Pole. Since the state of the system following a left/right move is completely determined by the knowledge of its current state and the proposed move, the state space is Markovian, and the interaction between states decays exponentially with respect to time [25]. Such exponential decays can be computed dynamically as discount factors [84]. Efficient procedures [176] proven to work for Markovian representations are known for this and related problems.* □

Non-Markovian representations and indirect operators require more complex TCA procedures [181]. Past decisions and/or states may need to be retained because they may influence feedback F independent of the current state. Ill-defined objective functions that can only be measured over intervals also require the retention of past states and/or decisions. Determination of the relative importance of

successive decisions may involve more than just a simple discount factor. Instead, the interdependence between different decisions may need to be captured explicitly using dependence graphs.

Example 2 (contd.). *Learning Strategies for the Towers of Hanoi Problem. The operators are indirect but the state space is Markovian. Feedback is delayed with respect to decisions; hence, TCA must be addressed. Because of the causal connections between decisions and states, solutions are structured. In this case, TCA separates essential states (those on the path from the initial to the desired state) from non-essential ones. SCA is then applied to generalize the reduced solution structure so it can be used for solving similar problem instances in the future [43].* □

In general, TCA can be posed as the computation of eligibility of a stored decision to receive a portion of the feedback signal [14]. Two factors contribute to such eligibility: the causal as well as the temporal relationships between it and F. Thus, TCA amounts to resolving i) whether a decision could have caused the feedback, and ii) whether its effects were still persistent at the time of feedback generation [115].

3) Prediction

Prediction can be classified with respect to future states and future feedback. Predicting future states is important in knowledge-lean learning environments because the problem solver needs to compare the eventual relative merits of different states resulting from the application of different operators in the current state. On the other hand, predicting future feedback is important when feedback is delayed; here, the learning system needs to estimate future feedback in order to determine the magnitude and direction of parameter modification.

a) Predicting Future States. The external environment changes under the influence of a problem solver's decisions as well as under the natural dynamics of problem variables. Whereas changes of the first sort can be determined by reasoning with causal models of the problem solver's operators, determining the latter requires a model of the natural dynamics of the external environment. Such may be the case,

for instance, in tracking the position and velocity variables of a projectile. A problem solver's strategy must include — explicitly or otherwise — some way of estimating the future states of the external environment.

Example 2 (contd.). *Learning to Steer a Ship. Given the knowledge of the effects of steering, the problem solver can predict where the ship will be after a few time steps if it maintains its current speed and direction. Preconditions and postconditions on steering operations are adequate for qualitative prediction of immediate changes to the ship's trajectory under the influence of the problem solver's actions. Here, the theory of knowledge-based prediction is available [40] as well as applicable.* □

b) Predicting Future Feedback. When the environment produces immediate prescriptive feedback, the learning system simply attempts to reduce the error between the observed and the desired values of decision variables. However, when the feedback is evaluative, the learning system must learn to predict the externally generated feedback signals using its own internal state. Especially when feedback is temporally global, or when the objective function is ill-defined and measured over intervals, it is not obvious which states will lead to better feedback. While prior prediction of future feedback is useful for decision making, a posteriori association of feedback and states (as in the TCA problem) is useful for learning.

Example 3 (contd.). *Learning to Balance a Pole. This task features delayed and evaluative feedback. The length of episodes grows with experience because the problem solver can keep the pole balanced longer. To continue to learn within episodes, the learning system needs to predict future feedback so that it can substitute its prediction in place of the (missing) immediate feedback for each decision. The key (for Markovian representations only) is to express the total error of prediction as a sum of differences between successive predictions [175].* □

Example 1 (contd.). *Learning Strategies for Load Balancing. Status variables in load balancing vary with time; predicting their variation as well as their effects on future values of feedback are the central issues in learning load-balancing strategies. Prediction of feedback is complicated by the violation of the Markovian property. As the states evolve with time under their natural dynamics as well as under the influence of load-balancing decisions, more effort is spent on the tractable backward projection of credit assignment, and less on the forward projection of*

predicting future states or future feedback signals. □

4) Violation of the Markovian property

The Markovian property asserts that the future behavior of a system is not affected by past states, given the current state [81]. Its manifestation in decision making is the *path-independence* axiom: optimal decision in a state does not depend on the state-space path leading to that state [21].

Violation of the Markovian property is especially easy to verify for systems having time-varying problem variables. One can study the partial autocorrelations [25] of the time series generated by problem variables: nonzero correlations at lags greater than one are the simplest evidence for the violation of the Markovian property. This entails the incorporation of past states and decisions into the current decision point: a history of past decisions needs to be maintained for both decision making and TCA. Corollary issues, such as how to 'forget' old or unimportant decisions, must also be addressed [181]. The problem is particularly acute for tasks with time-varying problem variables because past states and decisions may carry information useful at future decision points.

Example 1 (contd.). *Learning Strategies for Load Balancing The violation of the Markovian property is shown by significant partial autocorrelations in temporal variation of status variables at lags of two or more [115].* □

5) Constraint Handling

Constraints limit the solution space for decision variables on the one hand, and impose restrictions on generalization by the learning system on the other. Soft constraints can often be incorporated as (positive) penalty terms added to the objective function of a (minimization) problem [188]. Unknown or ill-defined soft constraints are handled just as unknown or ill-defined objectives. In contrast, hard constraints must be modeled explicitly [8] so that the learning system does not over-generalize from limited experience. Techniques such as Lagrange multipliers [188] are useful for incorporating hard constraints

into objective functions during problem solving. However, they increase the effective problem size and, therefore, the complexity of the problem.

Example 4 (contd.). *Learning Strategies for the Towers of Hanoi Problem. This task has one significant hard constraint, namely, that no disk can ever be placed on another of smaller diameter. When generalizing from solutions to specific subgoals to strategies for achieving more general goals, the learning system must ensure that the strategies it learns do not violate this constraint.* □

6) Storing Past Decisions

The violation of the Markovian property and the consequent importance of past decisions during TCA entail retaining past decisions and possibly even the states in which these decisions were taken. This raises a twofold problem: managing the storage of these state vectors, and optimizing the time for distributing credit among them. Not all the information in a state vector is relevant, nor is it feasible to go through the entire history each time a feedback signal is received. Determination of the past information to be retained then becomes important.

The first issue in maintaining such a history is that of size. Even with non-Markovian state spaces, and especially with time-varying problem variables, the effects of decisions become insignificantly small after a certain time interval; such decisions should be 'forgotten' or deleted from the history.

The second issue concerns the extraction of relevant information from past history so that when feedback becomes available, it can be distributed among decisions in proportion to their contribution to the state(s) being evaluated by the current feedback signal. Sutton [175] proposes the use of a scalar 'eligibility' value for each modifiable parameter of the decision maker. Holland [83] maintains a 'strength' parameter with each rule, which represents the average credit received by that rule over several problem instances. Thus, scalar indicators of eligibility can be associated with either individual parameters or rules. Yet another option is to associate such indicators with each decision stored in the history [114].

When past state information is the trace of a time-varying problem parameter, time-series methods similar to those used for learning objective functions may be employed for automatically constructing abstract problem variables, which can then be used in decision making [116]. The naive alternative — to maintain the full state vector and associated decision variables and to process feedback signals using prior knowledge — is both expensive and unsuitable for knowledge-lean environments.

Example 1 (contd.). *Learning Strategies for Load Balancing. The violation of the Markovian property entails the storage of past states. The effect of past states on future decisions is exercised through abstract decision metrics (called* load averages*) that are used in decision rules instead of status variables. The formulae for computing load averages can either be user supplied [61] or be induced from completion-time measurements (Chapter IV).* □

7) Managing Multiple and General Objectives

This requires representation schemes that bring out the internal structure of objectives as well as the techniques for achieving general goals. When multiple objectives are present, the learning system may need to use general-purpose reasoning techniques [101, 102]. In this case, the procedural component of the knowledge stored should be reduced to a minimum; such reduction in procedural knowledge may slow down learning as well as problem solving. A more important consequence is that the learning system may need to acquire multiple strategies for meeting the multiple objectives. This raises the need for efficient indexing of strategies by goals [12].

8) Dynamic Decision Making

Most problems in planning and control involve dynamic decision variables because the environment changes state in response to the problem solver's actions. Dynamic decision variables require causal models for representing the interdependence between decisions and states. Briefly, a *causal model* is defined as a set of rules for determining the new state given an old state and a decision. It must be stressed that for tasks with time-varying problem variables, the new state has a causal component (due to

decision-making) as well as a temporal component (due to the natural dynamics of the external state).

If the Markovian property is satisfied, then dynamic decision making can be addressed using dynamic programming [21] (in knowledge-rich environments) or heuristic dynamic programming [193] (in knowledge-lean environments); otherwise, there are two alternatives: (i) to develop new techniques that are suitable for non-Markovian representations [116]; and (ii) to somehow transform the given dynamic decision problem into a static decision problem, and thereby avoid TCA entirely (Figure 4).

9) Handling Structured Solutions

A solution is structured when a number of indirect operators contribute to the reward state(s) of the external environment. Indirect operators, in conjunction with hard constraints and symbolic problem variables, characterizes some of the most complex learning tasks. For generating structured solutions, static strategies are preferred over dynamic ones because static strategies can simultaneously consider multiple (interdependent) decision points.

Static strategies are possible in knowledge-rich applications in which the strategy-learning system exploits the structure of the solutions already tested in order to find solutions for problems not seen before. Such learning requires substantial deductive reasoning as well as inductive generalization of solutions. Static strategies are not possible in knowledge-lean applications because there is not enough prior knowledge to guide the design of such strategies. In this case, new knowledge acquired during learning must be incorporated into new strategies in the system, causing considerable overhead during learning.

Example 1 (contd.). *Learning Strategies for Load Balancing. Since placement is a direct operator, balancing tasks without precedence constraints result in unstructured solutions. On the other hand, the placement of dependent tasks requires (indirect) operators for matching, sorting, and assignment, which combine into structured solutions. For efficiency reasons, dependent tasks can be scheduled using static strategies in which the complete mapping of all the jobs must be determined before executing any one of them [200].* □

10) Nondeterminism

Several problems discussed earlier — standard-of-comparison and learning-while-searching — are related to the basic problem of nondeterminism in problem solving. When a problem solver is forced to choose between alternatives, it must attempt to reduce the nondeterminism by limiting the number of alternatives actually explored. The challenge of nondeterminism is for the strategy learner to discover heuristic rules that allow it to choose the 'best' alternative without having to explore all of them. In knowledge-lean environments, alternatives cannot be compared because there is insufficient information. Further, exploration of an alternative may have an unpredictable effect on the external state, especially for time-varying problem variables. In this case, the learning system may sustain nondeterminism by introducing a controlled element of randomness in decision making. Only one randomly selected alternative is explored on each visit to a state but several different alternatives may be explored if each state is visited several times. Such strategies, also known as stochastic strategies, are popular in strategy-learning systems for knowledge-lean environments [14, 196].

Example 1 (contd.). *Learning to Balance a Pole. Instead of computing the value of the decision variable directly, the controller* PS *computes the probabilities of the left and the right moves at each decision point [5]. The actual assignment of a value to the decision variable is performed randomly using these probabilities. This approach is easily extended to the case with multiple alternative actions at each decision point [140]. As the probabilities of selection become more and more biased in favor of positive alternatives, nondeterminism is reduced.* □

Table XVI summarizes issues relevant to the tasks listed at the beginning of Section B. The next section surveys methods for resolving each of these issues.

D. Techniques for strategy learning

Strategy-learning techniques have been developed by researchers working in many different areas: machine learning, neural networks, cognitive science, and decision theory. In order to clearly bring out their commonalities and differences, it is important to understand how these techniques may be used for

Table XVI
Examples of issues in strategy learning

Example Task	Relevant Issues
Load Balancing (Example 1)	''Minimum-completion-time'' objective ill-defined, measurable over intervals [115]; Standard-of-comparison problem due to lack of absolute evaluations [85]; SCA over process-migration rules [87]; TCA over interacting sequences of placement operations, complicated by a violation of the Markovian property [115]; Prediction of future values of status variables [74]; Capturing the effects of past states and decisions by learning abstract decision metrics (load averages) [115].
Steering a Ship (Example 2)	SCA on preconditions of operators [43]; TCA over solution structures comprising indirect operators [45]; Prediction of trajectory under the influence of time-varying parameters as well as the controller's actions [9]; General problem-solving and learning capabilities for diverse scenarios [101]; Reduction of nondeterminism by macro-operator formation [99].
Pole Balancing (Example 3)	''Balanced-pole'' objective ill-defined but measurable over instants [5]; Within-trial learning warranted by episode length and asynchronous timing of feedback [5]; TCA over sequences of balancing operations; history maintenance and eligibility computation simplified by the Markovian property [173]; SCA over probabilities of applying ''move-left'' and ''move-right'' operations [14]; Prediction of future feedback signals necessary for within-trial learning [175].
Towers of Hanoi (Example 4)	General problem solving warranted by recursive transformation of the original goal into subgoals [102]; SCA over heuristics for move selection [43]; TCA over explicitly stored structured solutions due to indirect operators; Explicit modeling of hard constraints [8]; Controlling nondeterminism by macro-operator formation [99].

resolving the issues presented in the previous section. (See Table XV for a quick overview of the methods presented in this section.) The rest of this section presents an *issue-based survey* of strategy-learning techniques.

1) Tackling Ill-Posed Objectives

A strategy-learning task is ill-posed when it lacks either a complete and accurate specification of its objective function or an appropriate way to combine multiple objectives. Consequently, it does not have

a performance standard for choosing among the operators available at each decision point. The methods we describe in this section use past experience with problem solving to either model the objective function or learn other strategic information that will help the problem solver to compare the relative utilities of alternative moves.

Any of the methods described in this section can be adopted for learning while searching, provided that it addresses the tradeoff between exploration and convergence. While exploration demands that the system should sample a large number of states (usually by having a stochastic strategy), convergence demands that it should converge quickly to the true model of the objective function, and thence to the strategy that optimizes the function. A number of approaches for addressing this tradeoff rationally are now available in the literature [16, 72].

There are two general techniques for making a problem with an ill-defined objective function well-posed: learning from absolute evaluation, and learning from relative evaluation. Both assume that the objective function is ill-defined and measurable over instants. Only the latter addresses the standard-of-comparison problem.

Learning from absolute evaluation. This technique assumes that the feedback signal F is directly related to the (unknown) objective function. It is useful for learning *evaluation functions*, which can be used for predicting the true (or expected) objective-function value of states. We now describe two methods: the first is applicable when feedback is immediate, prescriptive, and available for each state; the second, when it is immediate, evaluative, and (possibly) unavailable for some states.

a) Learning the Objective-Function Value of States. This approach treats states as independent data points and associates an evaluation with each state [37, 106]. This is equivalent to assuming a Markovian representation for states. For example, in the game of checkers, the value of a board position is a function of the current board configuration irrespective of the moves made to reach that configuration. Therefore, several researchers [149, 157, 158] have designed systems that learn evaluation functions for this game.

Another approach associates evaluations with state sequences, and performs time-series regression [129, 159] of measured objective-function values using recorded values of problem variables. Most existing work on strategy learning considers only problems satisfying the Markovian property; therefore, this approach has not been used in the literature. However, in learning strategies for load balancing (Example 1), if the job descriptors were available for prediction, completion-time-based objective functions can be characterized in terms of the job descriptors' current values as well as the time series corresponding to status variables.

b) Modeling the Reward-Generation Mechanism. When external feedback is available only intermittently, an internal model — which characterizes feedback as a function of current and past values of problem variables — can be used for generating an internal feedback. Such a feedback signal can be used for evaluating the states not immediately followed by an external feedback signal [125, 137]. This approach works with evaluative feedback. However, in order to learn strategies from such feedback signals, the learner should know the relative importance of current and future signals, as well as whether feedback is to be minimized or maximized. Since learning from evaluative feedback is called reinforcement learning, internal generation of missing reinforcement is termed *secondary reinforcement learning* [125].

Example 3 (contd.). *Learning to Balance a Pole. Barto, et al. [14] describe an approach using neural networks to learn a model of external feedback. Their approach performs learning while searching. It learns to predict the 'pole-fallen' signal, but, more importantly, it is able to provide evaluations for states not evaluated by external feedback. Anderson [5] extends this work further using the more powerful multi-layered networks of nonlinear sigmoidal units [154] to perform nonlinear regression.* □

Learning from relative evaluation. The approaches described above do not address the standard-of-comparison problem [1] for learning from evaluative feedback in knowledge-lean environments. The methods we describe next can be used when neither the sign nor the absolute value of evaluation is important. Methods that learn from absolute evaluation can be thought of as *value-based*: they induce a

model that can predict the objective-function value for any given state. The methods described below can then be thought of as *direction-based*: they estimate the slope of the objective function. These methods assume only that the relative magnitude of evaluations is proportional to the relative goodness of solutions. They learn a restricted class of *dominance relations* [146, 201], which can be applied to either a state and its siblings, or a state and its ancestors, in order to select states for further exploration.

a) Using Past States as Points of Reference. Certain methods target improvement rather than optimization, and are useful when the degree of optimality of a solution cannot be computed easily. In this case, the current state is evaluated against either an incumbent (the best evaluation so far) or an average of recent values [68]. Thus, even though the original objective was ill-defined, the goal of learning is well-defined, namely, to improve upon the incumbent.

Example 3 (contd.). *Learning to Balance a Pole. An interesting approach to this problem is illustrated by the work of Morgan, et al. [134] who use structurally local, immediate, evaluative feedback. They generate positive feedback for the part of the decision maker favoring 'move right' and negative feedback for the part favoring 'move left,' when the pole is to the right of vertical; and vice versa. The implicit goal of each part is to minimize feedback, which is generated when the pole either changes the side to which it leans, or changes significantly its angle of incline. When the feedback is received, the system uses only the immediately preceding state as a point of reference in order to improve the balance.* □

b) Using Alternative States as Points of Reference. For certain problems, especially those with objective functions measurable only over intervals, it is not clear how to define a point of reference using past states. However if relative evaluations for alternative moves can be obtained, then the strategy learned should choose the move that has the best (relative) evaluation. Producing relative evaluations requires a mechanism that learns to compare consistently. For example, Tesauro [179] uses neural networks for learning to perform pairwise comparison of board configurations in learning strategies for the game of backgammon.

Example 1 (contd.). *Learning Strategies for Load Balancing Completion time of jobs are important in determining the objective-function values for the load-balancing problem. Completion time is a function of both the time series generated by status variables and the current value of job-descriptor variables. The latter are unavailable for prediction, but their common effect can be ignored in comparing alternative operators that represent the migrations of the same job to alternative sites for execution. In order to perform a sensible comparison of alternatives, a reference state — characterized by no load on this site — is used as a common point of reference. Regression or time-series regression, whichever is appropriate, is used for learning to compare the relative completion times of the same job under different loads. (Also see Chapter IV.)* □

2) Solving the Credit-Assignment Problem

Exact and efficient algorithms for structural credit assignment are available for a variety of decision structures; however, approaches to the temporal credit-assignment problem tend to be ad hoc and biased.

Solutions to structural credit assignment. The eventual goal of SCA is to modify the decision processes so that the error between the actual and the expected outputs is reduced. Methods to achieve this goal need to consider the representations of solutions and strategies. Existing SCA algorithms adopt one of the following three representation techniques.

a) Explicit Representation of Structured Solutions. When the decision variables are dynamic and the problem solver's operators indirect, solutions are structured and consist of interdependent operator applications. This is particularly true of symbolic problem variables because decision rules are instantiated differently in different contexts. SCA for such tasks uses explicit representation of solutions, and distributes feedback backwards from the more recent decisions to the less recent ones. An example of such techniques is goal regression (pp. 300 of [64]), which is particularly suitable for learning from prescriptive feedback in knowledge-rich environments. In knowledge-lean environments, an approximation technique called experimental goal-regression [147] is more appropriate. Both techniques use the regression step, wherein new subgoals are derived from the unsatisfied preconditions of an operator, whose known

postconditions match the current goal. If operator application leads to success, the preconditions of the operator are generalized either analytically (as in Explanation-Based Learning, or EBL [43, 126, 133]) or experimentally as in experimental goal-regression.

Example 5. *Learning to Solve Simultaneous Equations. Porter and Kibler [147] use experimental goal-regression in learning heuristics for solving simultaneous linear equations. The objective is to minimize the number of terms in the equations. The problem solver has operators that, for example, combine terms containing the same variable, combine constant terms, cancel zero-valued terms, substitute one equation into another, or replace an equation by its difference with another equation. The symbolic nature of terms and the indirect nature of operators give rise to structured solutions.*

In this system, strategies are represented as preconditions for applying operator; these preconditions are initially set to some general test that always succeeds. Therefore, the problem solver initially finds solutions to given instances by search. When a solution is found, its specific facts become the preconditions for applying the operators involved. Parts of this solution structure may be found useful for solving another problem instance in the future; when that happens, the preconditions are generalized to accommodate both instances. The feedback signals merely indicate the applicability of operators, and SCA uses them to modify the existing strategy so that operators involved will apply correctly to a greater number of future instances. □

A problem that complicates SCA with symbolic variables is the combinatorial complexity of the interactions among the variables to be considered. In this case, strategy learning can benefit from *factoring* the set of problem variables into groups of related variables; the simplified algorithm is called independent credit assignment (ICA) [64]. ICA is useful for learning from structurally local feedback.

b) Explicit Representation of Decision Structures. When solutions lack structure, a commonly used method for SCA is to represent explicitly, in richly connected networks, all possible ways in which decision-making rules can be combined. SCA in such structures consists of recording the level of activity for various components at the time of making a decision and then modifying either the components, their interconnections, or both.

Example 3 (contd.). *Learning to Balance a Pole. Anderson [5] uses a feedforward neural network [154] to represent explicitly the decision structure of his pole-balancing system. Even though multiple decisions are made using the same network, all information used for credit assignment is stored as eligibility values with weights of the network. Weights are modified in the direction of more positive feedback, which constitutes a change of strategy.* □

c) SCA with Decision Procedures. Sometimes structural credit assignment is not possible because decisions are made in algorithms, and even though the algorithms may have modifiable parameters, the role of these parameters in decision making is not explicit. For example, in learning strategies for load balancing (Example 1), if the migration policy were implemented procedurally, and if the thresholds and weights used by it were available for modification, then some way other than traditional credit assignment must be found for altering the policy. One possibility is to model the effect of various parameters on (measured) performance. Using this model, SCA may be carried out as before. Another possibility is to have a population of candidate strategies that can be tested on different sets of instances. The best one can be selected based on measured average performance. Systems that simultaneously test multiple alternative strategies are said to be *population based* [185]. Population-based learning has been found to be a viable alternative to structural credit assignment, and its applicability is characterized mainly by the nature of its episodes. It has been applied to learn new heuristics for process mapping [87, 88] and for designing suitable neural-network configurations [184].

Solutions to temporal credit assignment. TCA is responsible for disentangling the interrelationships between decisions and feedback. When solutions are structured, the decisions are causally related to one another. Explicit representation of such causal relationships can, therefore, simplify TCA; however, forming such representations requires complete and accurate knowledge of the consequences of applying each operator. In knowledge-rich learning environments, such knowledge is available to the learning system in the form of causal models. In knowledge-lean environments, there are two possibilities: the first is to determine heuristically causal connections between decisions [105]; the second, to learn causal models in the course of problem solving [174]. While the first of these approaches introduces errors and biases into

the learning process, the second incurs overhead during learning as well as problem solving. In general, the quality of knowledge-lean TCA procedures is somewhat suspect. Where possible, TCA should be avoided in knowledge-lean learning environments.

When problem variables vary with time, TCA requires not only a causal model but also a *temporal model* specifying the natural dynamics of the external state. Existing systems [95, 135] often make simplifying assumptions about these dynamics in order to simplify the model. For instance, Dean and Kanezawa [40, 41] classify operators into those that support the truth of a problem variable and those that do not. In their model, the probability of a Boolean variable being true increases exponentially towards one following a supporting action, and decays exponentially to zero following an interfering action. Exponential or memoryless dynamics are tantamount to assuming Markovian representations.

TCA resolves the causal and temporal dependences between decisions and feedback. For example, in the pole-balancing task, there is a causal dependence between a balancing decision and the state following it. There is also a temporal dependence, namely, the effect of a decision on a state decays exponentially with time. Typically, the causal component is used in an all-or-none fashion, to determine which decisions are at all eligible to receive feedback. On the other hand, the temporal model is used in the actual division of feedback because it provides a numerical eligibility value for each decision. Existing TCA algorithms follow three general schemes for representing and for resolving decision interactions and are described next.

a) Explicit Representation of Causal Dependence. Whenever a problem solver produces structured solutions, *i.e.*, a number of different operators contribute to the rewarded state(s) of the external environment, TCA can use the causal dependence to identify candidate decisions for receiving a share of the feedback signal. In certain cases [199], the depth of a decision in the causal chain, as well as its distance from the current state, are used for computing a numerical eligibility value.

Example 5 (contd.). *Learning to Solve Simultaneous Equations. A solution to a given set of equations comprises numerous operators applied one after another. Initially applicable operators, such as substitution and differencing, lead to states in which later operators, such as operators for combining variables and constants, can be applied. The causal dependences between earlier and later operators do not necessarily apply to all pairs of operators in the sequence. Structured representations, such as trees and graphs, are better at capturing the necessary causal relationships. More importantly, unlike the aforementioned sequence, these do not imply any unnecessary dependencies. The feedback for a solution can be distributed either equally to all decisions that are causally connected to it [132], or proportionately based on the distance from the decision to the final state [147].* □

b) Explicit Representation of Decision Structures. In systems with common algorithms for SCA and TCA [4, 160], learning can be simplified if the solutions do not have an internal structure. As noted above, such systems represent their decision structure explicitly, which includes all plausible interactions among decision rules. The algorithms proposed for computing the eligibilities of parameters in these rules are variants of Sutton's TD procedure [175], which have been shown to be rational for tasks satisfying the Markovian property.

Example 3 (contd.). *Learning to Balance a Pole. Barto, et al. [14] use a simple neural network as the PS component of their learning system for pole balancing. Neural networks without feedback are explicit decision structures. (Certain other control architectures [90] employ recurrent networks, which cannot be considered explicit decision structures because the full context of decision making is not stored in such a structure.) Because of the Markovian nature of the representation, the temporal dependence between decisions and feedback follows an exponentially decaying pattern. Sutton [175] has shown that, for Markovian representations, eligibility can be updated incrementally. Since no additional information from past states and/or decisions is required, the eligibility values stored with each weight contain sufficient information for TCA.* □

c) Scoping of Decisions. This approach is characterized by a dynamic history of past decisions [115] and is especially suitable for tasks having either multiple or structured objectives and time-varying problem variables. Like the first approach above, it keeps an explicit record of decision making; and like the second, it computes numerical eligibility values. However, instead of associating eligibility values

with decision parameters, it associates them with decisions themselves.

The essential information with each decision is its scope: if a feedback signal falls within a decision's scope, then this decision is eligible to receive a portion of the signal. Two types of scope information are used.

Causal Scope can be used for separating decisions pertinent to each component or objective when multiple objectives are measured or evaluated, or when several components of an objective function can be computed independently. When feedback results from the natural dynamics of the external state, causal scoping can prevent decisions from receiving a share of such feedback.

Example 1 (contd.). *Learning Strategies for Load Balancing. The decision to migrate a job has the effect of reducing the load at the originating site and of increasing the load at the destination site. Only the decisions made using information about the load at either the source or the destination site are* in the causal scope *of a migration decision. When a job finishes, its completion time is evaluated against the case without load balancing, and an evaluation of the scheduling decisions generated based on such evaluation. A decision is eligible to receive a share of the feedback when its causal scope includes placement or migration decisions regarding the completed job.* □

Temporal Scope (or Extent). A state or a decision is said to be *in the temporal extent* of another (earlier) decision if the effects of the latter persist until the occurrence of the former. When the external state of the environment is fully controlled by the problem solver (closed-world assumption), decisions can have an infinite extent. More often though, decisions have a finite temporal scope and can be discarded from the history of past decisions without affecting learning. In addition, the learner can model the persistence of each decision (the degree to which a decision affects a feedback signal that occurs within the decision's extent). Such a temporal model may take several forms. When objective functions evaluate states independently (usually for Markovian representations), and when decisions affect the external state immediately, a temporal model specifying an exponentially decaying persistence is quite adequate. In more complex environments, where objective functions are evaluated over intervals, or where decisions take effect after an initial delay, more complicated temporal models may be needed.

Example 1 (contd.). *Learning Strategies for Load Balancing. Migration of a job causes load changes at both the site of origin and the destination site. The effects are felt while the job is executing; after a job finishes execution, future decisions are not affected by the migration decision in question. In this sense, migration decisions have finite temporal scopes. Within its temporal scope, a decision's persistence initially increases while the job is being migrated, then stays high during the job's lifetime, and drops quickly to zero again upon completion. When a migration decision is determined using causal scope to be a candidate for receiving feedback, it is given a portion of the feedback signal proportional to its persistence at the time of feedback.* □

3) Prediction

Prediction problems are inherent to tasks having time-varying problem parameters. A number of approaches, each with its own advantages and disadvantages, are available in the literature. Various factors affecting the choice between these include i) on-line versus off-line learning, ii) numeric versus symbolic variables, and iii) dynamic versus static decision variables.

a) Temporal Difference Methods

Temporal difference (TD) methods originated in early literature on dynamic programming and problem solving, and methods related to learning were studied formally by Sutton [175]. The essential idea of these on-line methods is to rewrite the total error of prediction as a sum of differences between successive predictions. If the problem solver uses Markovian representations, numeric problem variables, and direct objectives, then the error of prediction can be minimized by reducing the error between successive predictions [176, 181].

Example 3 (contd.). *Learning to Balance a Pole. A popular class of strategy-learning algorithms, known as ACE/ASE [14, 124], use TD methods to predict the discounted sum of all future evaluations at each time step while simultaneously using the difference between successive predictions to modify the predictor. (In the computation of this 'discounted' sum [84], immediate evaluations are given exponentially greater weight than their successors.)*

The predicted sum of future evaluations can be used instead of missing external evaluations as the internal feedback, thereby solving the TCA problem. The error of prediction is divided between prediction decisions using a recency weighting [175]: older predictions incur exponentially smaller errors than recent predictions when computing the current difference between successive predictions. This is a justifiable heuristic for Markovian representations and instant-evaluated feedback, both of which are true of the pole-balancing task. □

Time-series analysis. This refers a large class of approaches that build models using sequential periodic (therefore, synchronous) observations of time-varying parameters, and predict based on these models. Time-series methods have been developed by researchers in statistics, neural networks, and control. In general, these are useful for predicting future states of learning tasks having numeric problem variables and static decision variables.

Time-series models developed by statisticians [97, 191] comprise i) trends: linear, polynomial, and exponential dependence on time; ii) periodic components: as found by frequency-domain analysis of a signal; and iii) random or unmodeled dynamics of environmental processes. Statistical time-series analysis techniques tend to be off-line and require more complex computations than on-line ones, such as the TD method described above.

Nonlinear recurrent neural networks [192] can be used for time-series prediction. However, they have not been popular in strategy-learning systems due to the unresolved problems of instability, poor convergence, and chaotic behavior. Somewhat more success has been obtained with TDNNs (time-delay neural networks) [103], which are non-recurrent but examine simultaneously several delayed values of problem variables along with their current values. Both recurrent and non-recurrent neural networks are capable of on-line learning.

While the methods described above are applicable to prediction in the absence of prior knowledge, tasks with numeric time-varying parameters, Markovian representation, dynamic decision variables, and knowledge-rich environment can be solved using Kalman filtering [119]. Kalman filtering is an example of state-space methods developed by control theorists [73]. Its predictions as well as its model of

environmental dynamics can be updated on-line in straightforward steps.

Example 1 (contd.). *Learning Strategies for Load Balancing. The status variables in load balancing, namely, the levels of utilization on various resources, vary with time and often exhibit periodic (stationary with time) behavior as well as local trends (nonstationary behavior). These characteristics are due to the fact that the background load is generated by a process population whose rate of change is slower than the sampling rate of status variables. Statistical time-series methods such as ARIMA modeling [25] are useful for identifying trends in these series, which can then be used in making short-term predictions of future load.* □

Projection. Methods of temporal projection were developed as part of temporal-reasoning systems in AI. These employ a technique called forward chaining: the current state of the world and a set of rules that fire therein generate new facts that characterize a possible next state of the world. Using these new worlds, one can continue to reason and to get a two-step-ahead prediction, and so on [41, 79]. Heuristics of preference [40, 95, 135] are needed for narrowing down the space of possible worlds. Time, being a continuous quantity, does not easily lend itself to modeling with discrete event-based symbolic representations except when problem variables do not change outside the control of the problem solver. Projection methods are, therefore, applicable to symbolic (but not time-varying) problem variables and dynamic decision variables. Dean and Siegle [42] have recently proposed approaches that can reason symbolically about natural dynamics specified using differential equations.

Strategy-learning systems for symbolic problem variables and dynamic decision variables in time-varying environments commonly use reactive rather than pro-active approaches. Such methods are known as reactive planning [2, 66, 145, 162] and learning from failure [5, 35, 36]. In some sense, these approaches ignore the natural dynamics of environmental processes and predict (using chaining) only the causal dynamics of the process.

Example 2 (contd.). *Learning to Steer a Ship. Although the ship-steering task involves decision-making with time-varying problem variables, there is little time for the problem solver to search in the space of solutions. Instead, the problem solver uses a causal model that relates its control inputs and resultant states. The model is* learned from

observation *by relating patterns in recent control actions with patterns (trajectories) in state variables following these actions. It is then used in chaining to determine action sequences that will reach a desired state from the current state.* □

4) Exploiting the Markovian Property

Markovian representations often accompany objective functions that are either well-defined or ill-defined and measured over instants. Some of these well-defined functions satisfy the path-independence axiom; for these, in the presence of deterministic strategies, immediate feedback, and complete knowledge of state transitions (knowledge-rich environment), the optimal solution can be computed using dynamic programming [21, 195]. Similarly, for ill-defined functions, the objective of maximizing the (possibly discounted) sum of future evaluations is also amenable to dynamic programming [84]. Variants of this procedure exist for knowledge-lean environments as well as for problem solvers with stochastic strategies. In this section, we discuss three different dynamic-programming formulations for strategy learning. Note that these formulations are limited by their requirement of a finite number of states and their high complexity in the presence of a large state space.

Dynamic programming (DP). Approaches based on pure DP are suitable for knowledge-rich environments, symbolic or discrete problem variables, and Markovian representations. When the assumptions of DP are satisfied, the optimal strategy can be computed given the objective function (defined either as utility or as cost) and a causal model of the environment. However, without additional heuristics to restrict the space of possible solutions, DP is too complex to be practical. One way to harness this complexity is by learning dominance relations between states [202], which can be used to prune moves leading to suboptimal states.

Example 4 (contd.). *Learning Strategies for the Towers of Hanoi Problem. Dynamic programming can be applied when the objective is to find a solution that transforms the initial state to the goal state in the minimum number of moves. Since operators applied result in deterministic changes to the problem state, DP can be used. (For*

probabilistic transitions, the SDP algorithm described next can be used.) The first step is to set up the functional equations of dynamic programming [86], which define recursively the optimum cost of the final state(s) in terms of the optimum costs of the preceding states. In the Towers of Hanoi problem, there are two possible preceding states: either the smallest disk was moved from the leftmost tower or the middle tower. The optimum cost of each of these states can again be computed using similar recursive equations. Depending on which precedingstate achieves the minimum cost in the recurrence, the optimal solution can be constructed. The major drawback of dynamic programming is that it requires computations exponential in the number of problem variables. □

Dynamic programming can aid strategy learning by generating complete optimal solutions to certain specific instances; one can then use credit-assignment procedures [104] or generalization techniques [9] to generalize these solutions to new goal states as well as new instances.

Stochastic dynamic programming. When a problem solver's strategy is stochastic, so is the evaluation of the solution. In this case, it is not possible to say whether one policy is better than another for a given instance; instead, one needs to consider the expected evaluation of the policy rather than its exact evaluation [15, 153]. Methods analogous to deterministic DP can be used to compute the optimal strategy. Stochastic dominance relations between states [146] compare their expected evaluations rather than the exact ones, and help to limit the complexity of SDP. Their use leads directly to a method for refining strategies called *policy iteration* [84]. However, like DP, SDP also requires extensive knowledge of the environment and a well-defined objective function.

Example 6. *Learning the Optimal Route to a Goal around Barriers on a Grid. This is the route-finding task discussed by Barto, et al. [15] in their review of exact and approximate methods for SDP. The performance task has an ill-defined but measurable objective of reaching a target location on a two-dimensional grid, starting from a given location on the grid. It also has an ill-defined constraint that the path should not cross certain initially unknown barrier locations. The environment produces an immediate feedback signal whose value is* -1 *for all states except the goal state. The problem solver has four operators (whose preconditions are unknown but effects are known): one each for moving up, down, left, and right. The problem solver's state includes information about the current location on the grid. The state space is Markovian; and the operators, direct. The problem solver's strategy*

is stochastic, associating a probability of applying an operator with each state. Equivalently, the strategy defines a Markov chain on the state space, where transitions are possible from each state to one of its four neighbors. No transitions are possible out of the goal state.

With each strategy, SDP associates an evaluation function that allows one to compute, for each state, the expected number of steps to the target using the current strategy. As in deterministic DP, one can translate the optimality of strategies into the optimality of their respective evaluation functions [153]. SDP also permits one to set up a recurrence equation for computing the optimal path length with a k-step lookahead in terms of the optimal length for $(k-1)$-step lookahead. Once the optimal evaluation function is computed by solving this recurrence, the optimal strategy simply picks the action that optimizes this evaluation function at each step. □

Heuristic dynamic programming (HDP). HDP [16, 193, 194] is applicable in knowledge-lean environments where a complete model of state transitions is unavailable. As discussed before, DP derives an optimal strategy by first deriving an optimal evaluation function from a partial enumeration of the search space. HDP, on the other hand, works for problems with ill-defined objective functions by estimating an optimal evaluation function and an optimal strategy. In the absence of extensive knowledge of state transitions, the learned evaluation function and strategy can only approximate the optimal ones.

Example 3 (contd.). *Learning to Balance a Pole. The basic idea of HDP is to use a generalization of temporal difference methods to predict the sum of all future reinforcements. At any point in time, the prediction process is adapted so it will predict correctly the sum of the next state's prediction (using the current predictor), the external feedback received in that state, and an appropriate negative constant (to keep the sum of future reinforcements finite) [194]. The prediction at the next state using the current set of weights becomes the apportioned feedback for the current decision. The necessary strategy modifications for SCA can then be carried out using this feedback.* □

5) Constraint Handling

Problem solvers take two distinct approaches to constraints: i) by satisfying them explicitly, and ii) by incorporating them into the solution procedures. Accordingly, the strategy-learning systems can

modify either the operator-selection procedure or the objective function in order to prefer moves that generate valid states rather than those that generate invalid ones.

Constraint satisfaction. Constraint-satisfaction procedures for learning with symbolic (discrete) problem variables are typified by explanation-based learning (EBL) methods in which generalization is performed by retaining the structure of a known solution and by relaxing the conditions under which the structure applies. In this case, constraint satisfaction can reduce the space of solutions significantly. General techniques for finding feasible solutions for constrained problems with discrete-valued problem variables are surveyed by Nadel [138]. Techniques based on truth maintenance can be used to enforce symbolic constraints during decision making and learning. (See reference [112] for an overview.)

Constraint-satisfaction approaches for learning with continuous variables induce a model of the feasible region and use it to perform constrained optimization during both decision-making and learning. Examples include explicit search for feasible solutions, and subsequent training of decision makers using such solutions as examples [120]. Methods for constrained optimization of continuous functions are reviewed by Walsh [188] and use projection of infeasible solutions to the nearest point in the feasible region. Such methods are applicable only with well-defined constraints, but allow great flexibility for the strategy learner.

Example 4 (contd.). *Learning Strategies for the Towers of Hanoi Problem. The constraint, that if a disk A is on top of another disk B then A must be smaller than B, is part of the domain theory. New states are generated by applying operators; those that violate this constraint are simply inconsistent with the domain theory. Upon detecting that the state resulting from the application of an operator meets all other constraints except this one, the problem solver may postpone applying this operator until the constraint can be satisfied. In the meanwhile, the problem solver may attempt to satisfy this constraint by setting up alternative subgoals [9].* □

Constraint incorporation. When the problem variables are numeric, constraints can be incorporated into the objective functions as penalty terms. Jordan [90] describes one approach for incorporating constraints into the SCA process and another for incorporating constraints into the cost function. His

approaches work for a variety of numerical constraints in both knowledge-rich (constraints are explicit) and knowledge-lean (constraints are available implicitly by random sampling) environments. Similar approaches are discussed under the rubric of knowledge compilation. An example is the test-incorporation approach of Dietterich and Bennett [46]. The drawback of methods in this class is that different relative importance given to the objective function and constraint terms leads to different optimal solutions.

Example 6 (contd.). *Learning the Optimal Route to a Goal around Barriers on a Grid. In this case, the constraints are ill-defined in the sense that their violation can be detected only when the problem solver attempts to move into the barrier. The net effect of constraint violation is the increased cost of solutions because infeasible moves cause no change in state and are, therefore, wasted. Thus, by optimizing the cost function, the problem solver learns to avoid barriers automatically.* □

6) Managing a History of Recent Decisions

Available solutions to this problem use either the episode structure of problem instances [43, 132] or the Markovian property [5, 177] to limit the amount of information stored. When the problem variables do not vary with time, the state of the external environment does not change outside the control of the problem solver; in this case, the episode ends when the final feedback signal related to this instance is received. This feedback is used either for modifying the decision process via credit assignment, or for assessing the importance of the current solution path. In some cases, entire solution paths, or generalizations thereof, are retained for solving similar problem instances in the future. Examples include analogical learning systems [30, 75] that retrieve and transform deductively old solutions for solving new problem instances.

When the problem variables vary with time but their distributions are stationary, a learning system may still be able to converge to an optimal control strategy in either an absolute sense (for deterministic variation) or an average sense (for stochastic variation). One may consider such convergence as the end of

an episode. Frequently though, either the length of episodes or the nonstationarity in the environment preclude storage of complete solutions. When the Markovian property is violated, new states and decisions must be added or removed continuously from the history. No general approaches for such dynamic management of history are known.

Example 1 (contd.). *Learning Strategies for Load Balancing. As discussed in Section 4.2.2, load-balancing decisions have finite temporal scopes. Because the learning task is non-Markovian, it is essential to keep decisions around until feedback is received. Decisions whose temporal scope has expired are no longer eligible for feedback and can be deleted [116].* □

7) General Problem Solving and Learning

Systems for general problem solving requires symbolic-reasoning capabilities, irrespective of the nature of problem variables. Such systems are capable of general strategy learning, although extensive amounts of explicitly stated task-specific knowledge are needed. The techniques used by such systems include knowledge-compilation methods, such as macro-operator formation [9], chunking [107, 152], and procedure learning [141]. The archetypical system in this class is SOAR [172] which can learn strategies associated with arbitrary problem spaces defined by objectives and constraints using symbolic variables.

Example 7. *SAGE.2: A General Learning System. Langley [104] demonstrates his SAGE.2 system on the Towers of Hanoi (Example 4) and five other learning tasks. This program employs some general-purpose learning rules that allow it to characterize good solutions in many domains. For instance, one such rule takes a complete solution tree and characterizes the states on the path from the initial state to the goal state as good, while characterizing all other states as bad. Yet another general learning technique specializes preconditions of operator-application rules so that a rule will fire only when its firing does not result in a bad state. The same general rule is shown useful in learning strategies for several diverse tasks.* □

The techniques described in this section resolve most of the issues raised in Section C. Tables XV and XVI together describe how to handle the issues relevant to our running examples. Consider, for

instance, the load-balancing problem. Knowing that its objective function is ill-defined and measurable over intervals and that absolute evaluations are unavailable, we should apply time-series regression using an alternative state as a point of reference in order to address the ill-defined nature of its objective function. For the ship-steering task, which requires symbolic prediction of the ship's trajectory, the temporal-projection methods of AI are the appropriate technique.

To summarize, we have presented in Sections B through D a general method for identifying the issues and techniques relevant to any given strategy-learning task. We have described the applications in terms of the various attributes of its performance task, problem solver, and learning environment (Section B). We have identified the relevant issues based on the nature of the learning task (Section C). Finally, we have discussed appropriate technique(s) for addressing these issues (Section D).

E. Architecture of strategy-learning systems

This section reviews four strategy-learning architectures by describing their internal structure and operation. Their basic information flow follows the pattern shown in Figure 37. Depending on the assumptions made, an architecture may be more suitable for a given learning scenario than another.

1) Dietterich and Buchanan's Model

Learning systems developed in the areas of cognitive science and artificial intelligence tend to have knowledge-rich learning environments. The common features of such systems are represented well by Dietterich and Buchanan's model [13, 45], which is suitable for learning tasks with well-defined objectives and prescriptive feedback (Figure N). Other models of learning systems, such as those proposed by Langley [104] and Smith *et al*. [166], also fit into this general framework.

The problem solver of this model solves given instances using a causal model relating states, operators, and effects. Strategies are usually represented as preconditions of either operators or operator

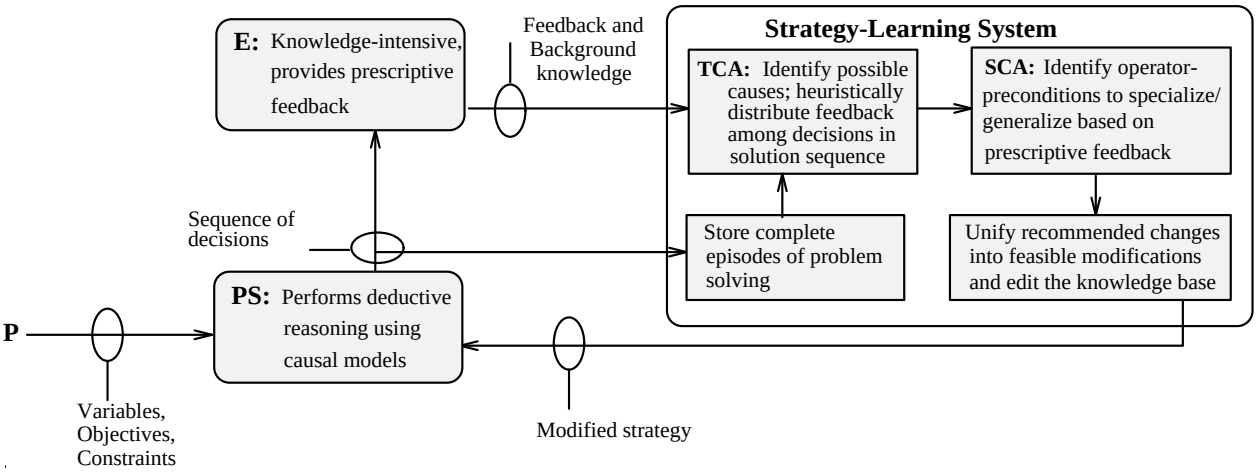


Figure 41. Dietterich and Buchanan's model of learning systems

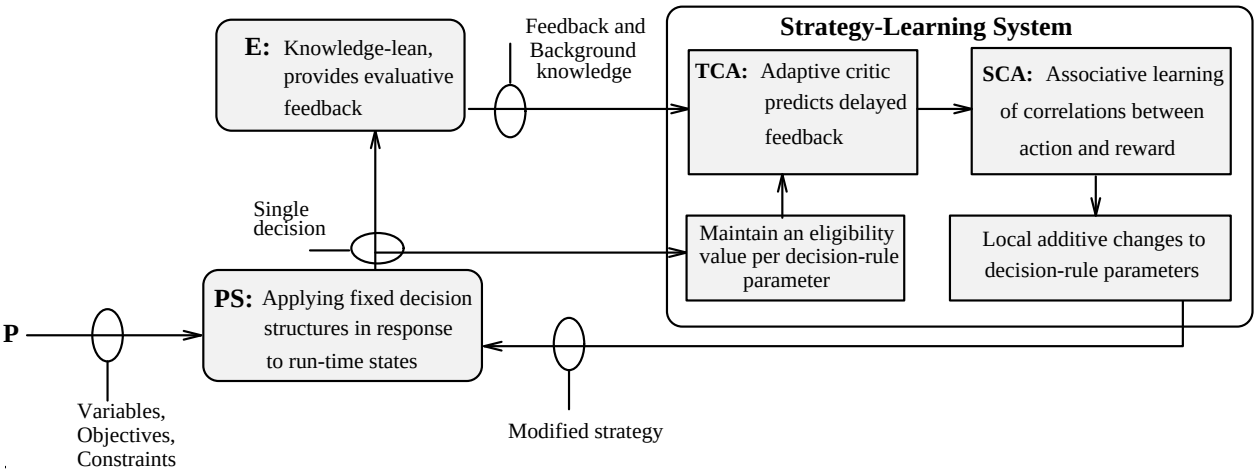


Figure 42. Minsky's model of learning systems

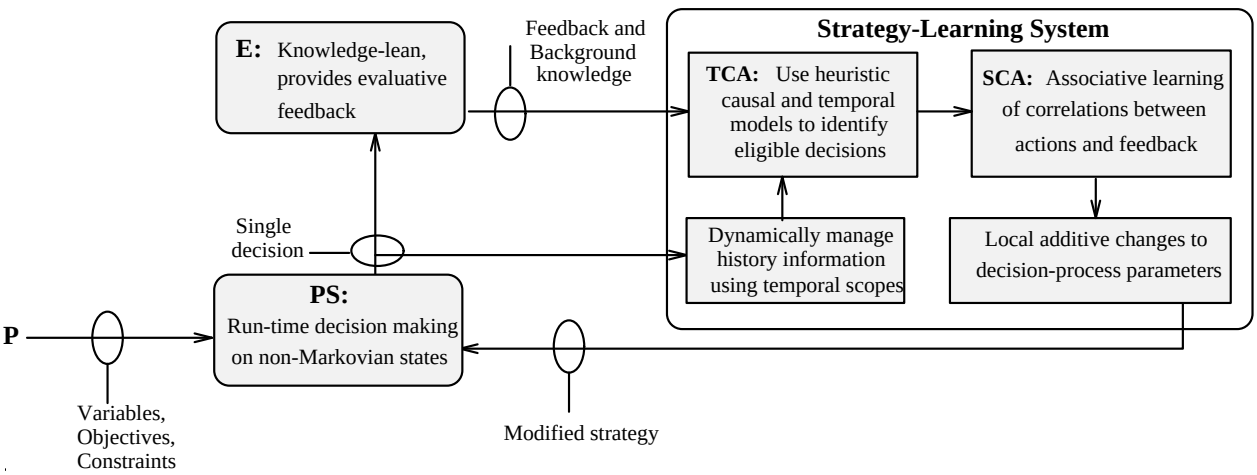


Figure 43. The hybrid point-based model of learning systems

sequences. The learning system contains a critic and a learning element, the former handling the semantic problems such as credit assignment, and the latter, syntactic problems such as parsing, unification, and eventual modification of the knowledge base. The credit-assignment modules make extensive use of stored knowledge both to relate the external feedback to objectives and to determine the preconditions of operators responsible for the current feedback. The learning system uses an internal model of the problem solver to edit strategies.

Retention of complete solutions permits the use of symbolic learning techniques, such as syntactic generalization of stored solutions and formation of macro-operators. Syntactic generalization techniques are particularly apt when the problem solver employs static strategies, and the feedback is delayed, prescriptive, and synchronous,

Syntactic generalization and credit assignment based on abductive reasoning permit systems based on this model to generalize substantially from each example. Rule schemata, which can be derived using only a few episodes of successful problem solving, capture the common structure of observed solutions; their preconditions are the maximally general conditions under which the operators of the common structure may be applied.

The chief advantages of this model are i) its generality — powerful symbolic representations allow the application of the same general learning principles to multiple performance tasks and problem solvers; and ii) its ease of debugging — extensive use of knowledge during credit assignment produces justifications for each generalization or specialization step. Its disadvantages are that i) it works only for problems with well-defined objectives; ii) it works only with prescriptive feedback; iii) it relies on prior knowledge of causal models; and iv) it cannot handle time-varying problem variables. Dietterich and Buchanan’s model is efficient in that it requires a small number of examples, but inefficient in the time required to learn from each.

2) Minsky's Model

Learning tasks in the areas of control and decision theory tend to have knowledge-lean learning environments. Techniques developed in these areas generally employ statistical methods to identify models, and estimate their missing parameters. Knowledge-lean environments usually produce evaluative feedback which, in the presence of time-varying problem variables, makes TCA the principal problem. The common characteristics of systems employing statistical methods for learning are well represented by Minsky's model [125] (Figure 42). This model is characterized by its use of feedback predictors (also known as *adaptive critics* or *secondary reinforcement devices*), which learn to produce internal feedback whenever the external feedback is delayed.

Systems based on Minsky's model generally operates continuously using dynamic strategies, and respond to ongoing changes in the external environment. In these systems, it is hard to identify episodes, and is nearly impossible to retain complete solutions. As a result, the eligibility of decision-making parameters is computed on-line without storing an explicit performance trace. Existing examples of this architecture invariably assume Markovian representations; hence, they lack the mechanisms for maintaining a history of old states and decisions, as well as the machinery for the associated TCA problems. Since eligibility values are computed on-line and correspond directly to the parameters of decisions making, this model provides efficient solutions to the credit-assignment problem, often integrating both SCA and TCA in the same learning algorithm.

Statistical methods are useful as ways to bootstrap learning without prior knowledge, but such (associative) learning techniques require a large number of problem-solving episodes. The amount of information extracted from each example is small relative to Dietterich and Buchanan's model. That causes slowness of learning, which is an important factor behind several recent proposals for hybrid architectures [177, 196].

Accurate and efficient estimation of strategic knowledge using statistical techniques requires resolution of the exploration-convergence dilemma. Systems based on Minsky's model work only with problem solvers employing stochastic strategies: the randomness of such strategies is a natural way to implement search or exploration of alternatives; convergence, on the other hand, can be achieved by reducing such randomness.

The advantages of Minsky's model are that: i) it is suitable for learning in knowledge-lean environments; ii) it can handle ill-defined objective functions; iii) it can learn from delayed, evaluative feedback; iv) it solves efficiently the credit-assignment problem; and v) it learns dynamic strategies that work better than static ones for problems with time-varying parameters. The disadvantages of this model are i) its reliance on Markovian representations, and the associated lack of mechanisms for using and for storing past decisions; ii) its difficulty of debugging due to the use of complex statistical models; and iii) its lack of generality due to poor transfer of learning across performance tasks. As a result, learning with Minsky's model requires a large number of examples, each taking a small amount of time.

3) Hybrid Point-Based Learning Model

The hybrid point-based learning model [114-116] (Figure 43) was motivated by non-Markovian strategy-learning tasks in knowledge-lean environments. Minsky's model is not directly applicable to such problems because it lacks both the knowledge and the reasoning mechanisms necessary for distributing credit among explicitly stored past decisions. Usually, knowledge-lean environments are accompanied by ill-defined objectives and evaluative feedback, both of which preclude the use of Dietterich's model.

In the hybrid model, the states and decisions are recorded in a dynamically managed history as decisions are made. Temporal models, containing information about persistence and temporal scopes, are used in history management as well as in temporal credit assignment. Causal models are used only to identify candidate decisions during TCA. Relative to Dietterich and Buchanan's model, the causal model

used here can be heuristic and less detailed. Unlike Minsky's model, the hybrid model cannot combine solutions for TCA and SCA: instead of associating eligibility values with modifiable parameters of the problem-solver's strategies, it associates eligibility values with individual decisions in the history.

When feedback becomes available, the candidate decisions are first identified using the heuristic causal model; the eligibilities of individual decisions are then used for proportional assignment of credit/blame among decisions. SCA is performed on the explicitly stored portions of solutions. Examples of systems employing the hybrid model include Samuel's Checker Player [158], Widrow, *et al*.'s truck-backer-upper system [143], and learning systems based on genetic algorithms [24, 167].

The hybrid model compromises the generality of Dietterich's model and the efficiency of Minsky's model. It is like Minsky's model with additional functionality for maintaining history information and performing TCA on stored decisions. Such added functionality allows the hybrid model to work with problem solvers employing non-Markovian representations. Alternatively, the hybrid model can be viewed as Dietterich and Buchanan's model with additional functionality for tackling time-varying problem variables, ill-defined objective functions, dynamic strategies, and evaluative feedback.

4) TEACHER: A Population-Based Learning Model

All the models described above employ point-based search in the space of strategies, using credit assignment to modify the incumbent strategy based on feedback, and either stochastic strategies or explicit perturbations to explore the search space. The fundamental idea in TEACHER [185] (Figure 44) is to use population-based methods for probing the strategy space at several points simultaneously. Starting with an initial pool of candidates, TEACHER admits several ways of generating new candidates: i) *grammar-based*, in which problem-solving strategies are generated as leaves of a phrase-structured grammar; ii) *perturbation-based*, in which random or systematic perturbations are applied to the incumbent strategy to obtain new candidate strategies; and iii) *performance-based*, in which credit assignment is

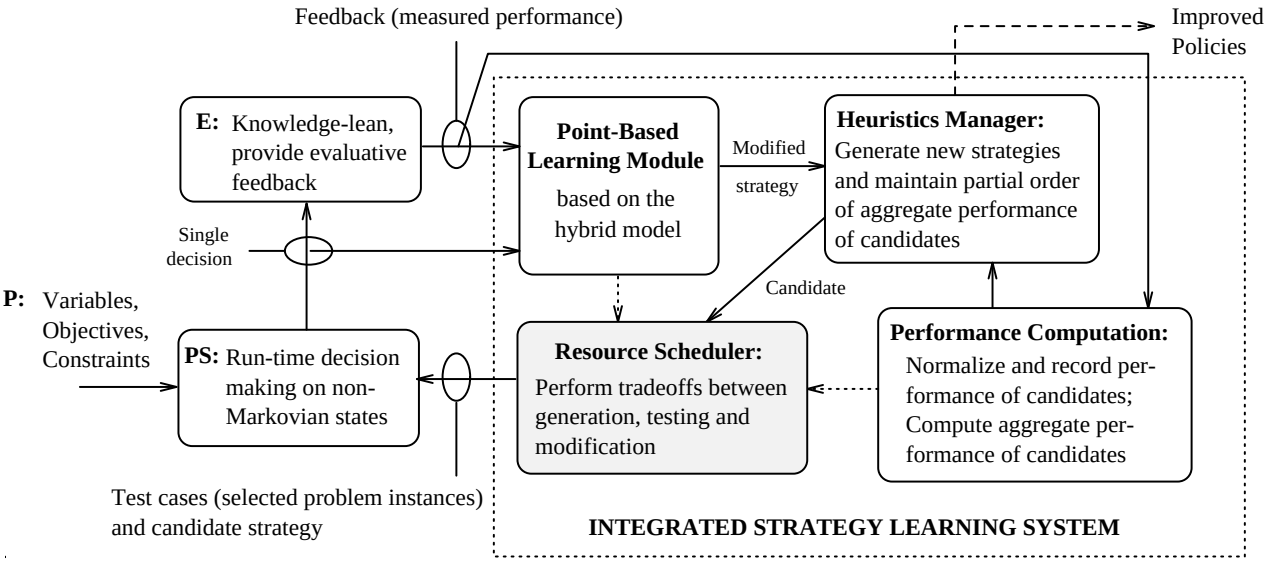


Figure 44. The TEACHER model of learning systems

used either for modifying the incumbent or for updating the probability of selecting one of the candidates generated.

TEACHER is characterized by its resource scheduler, which rationally divides learning time between generation, testing, and modification. TEACHER honors deadlines on learning time. In order to make rational and effective use of the limited time, it uses the theory of sequential selection from statistical decision theory [67]. However, this theory applies under certain restrictive assumptions on the distributions of objective-function values over the populations of problem instances.

Three significant advantages of TEACHER accrue from its use of a population-based approach: i) parallelism — a number of different strategies can be tested in parallel; ii) global search of strategy space — while point-based methods explore only some neighborhood of the initial strategy, converging to the nearest locally best strategy, TEACHER performs a multi-pronged search and is more likely to hit a globally best strategy given enough time; and iii) it is able to learn with decision procedures whereas point-based methods fail when not given access to the internal model of the problem solver's decision process. Thus, TEACHER has all the power of the hybrid model plus the added functionality for resource-constrained learning.

Table XVII
Architectural models characterized by their approach

Model†	Issue	Approach
D	Credit Assignment	Knowledge-based algorithms using ICA (Sec. 4.2.1); explicitly represented structured solutions; well-defined objective functions
	Prediction	Projection of symbolic variables using explicit causal model
	General Problem Solving, Non-Markovian Representations, Indirect Operators, Multiple General Objectives, Storing Past Decisions	Using powerful symbolic representations, prior knowledge of general purpose problem-solving techniques, static strategies in problem solvers, and explicit storage of structured solutions
M	Ill−Posed Objective Functions	Interleaved learning of instant-evaluated objective functions and dynamic strategies; learning from absolute evaluation and (delayed) evaluative feedback.
	Credit Assignment	Complex TCA problems in knowledge-lean environments solved using on-line learning of reward-generation mechanism; SCA interleaved with TCA; not suitable for non-Markovian representations
	Prediction	Learning to predict future feedback as a function of current and recent inputs by temporal difference methods; projection to future states not attempted due to lack of causal models
	Dynamic Decision Making	Using dynamic programming and its variants to learn dynamic strategies from evaluative feedback for Markovian representations
	Nondeterminism	Controlled using stochastic strategies
H	Ill−posed Objective Functions	Off-line learning of interval-evaluated objective functions from absolute or relative evaluation
	Credit Assignment, Storing Past Decisions, Non-Markovian Representations	Off-line learning of causal and temporal models, explicit storage of past decisions (with size limited by temporal scopes), and combination of causal and temporal scopes; valid even for non-Markovian representations
T	Credit Assignment	Maintenance of a population of competing strategies, and use of credit assignment only to alter probability of selection but not necessarily to improve the incumbent strategy; avoidance of structural credit assignment entirely, hence, suitable for decision procedures
	Resource Constraints	Using statistical theory of sequential selection to resolve the exploration-convergence dilemma
	Efficiency	Amenable to parallel implementation on multiprocessors

†D: Dietterich and Buchanan's Model (Fig. 41); M: Minsky's Model (Fig. 42); H: Hybrid Model (Fig. 43); TEACHER Model (Fig. 44)

Table XVII summarizes the characteristic features of the four architectural models. Table XVIII reviews their applicability to the strategy-learning tasks described at the end of Section B.

F. Practicality of strategy learning

In this section, we examine practical issues in the design of strategy-learning systems. A practical system should be easy to debug, efficient at learning, able to tackle unseen instances, and able to produce

Table XVIII
Examples of strategy-learning tasks and architectures

Example	Model†	Comments
Load Balancing	D	Knowledge-based critic needs a well-defined performance standard, which is not available. This model lacks mechanisms for predicting future values of time-varying status variables because it assumes that nothing changes outside the problem solver's control.
	M	The adaptive critic attempts to compute eligibility on-line, which does not work for non-Markovian representations. The adaptive performance element is efficient for learning stochastic strategies from evaluative feedback.
	H	The model is suitable for non-Markovian states and time-varying status variables. It is difficult to find a persistence model that properly accounts for feedback delays.
	T	The model obviates the credit-assignment problem. Population-based learning allows multi-pronged performance-directed search in strategy space. If background knowledge were available, it can be used in intelligent generation of new strategies.
Ship Steering	D	A good match because of well-defined objective function, static strategies, and knowledge of operator semantics.
	M	The model is unsuitable because of non-Markovian representations, complex decision rules, and deterministic strategies.
	H	Unnecessary overhead is incurred due to the additional functionality for handling dynamic strategies and dynamic history management using temporal scopes.
	T	The functionality for simultaneous testing of multiple strategies is not needed for this problem. Sufficient background knowledge is available so that point-based learning using Dietterich and Buchanan's model requires only a few problem-solving episodes.
Pole Balancing	D	Knowledge-based learning is not applicable to the ill-defined objective function and time-varying problem variables.
	M	A perfect match for this problem due to the Markovian representation, ill-defined objective function measurable at instants, knowledge-lean environment, and evaluative feedback.
	H	This model works but fails to exploit the Markovian property for efficient solutions to SCA and TCA.
	T	Point-based learning suffices since the Markovian property is satisfied and TCA can be solved efficiently.
Towers of Hanoi	D	Perfect match for this model due to its ability to handle multiple, general objectives. Syntactic generalization techniques of this model work well for constrained generalization of observed solutions.
	M	The model lacks mechanisms for handling hard constraints and learning from complete solutions.
	H	Since the Markovian property is satisfied, the extra functionality of this model is not used.
	T	Lengthy test-cases preclude the application of population-based learning.

†D: Dietterich and Buchanan's Model (Fig. 41); M: Minsky's Model (Fig. 42); H: Hybrid Model (Fig. 43); T: TEACHER Model (Fig. 44).

strategies that generate solutions of acceptable quality in a reasonable amount of time. These issues have traditionally been either ignored or solved using *ad hoc* techniques. They represent important dimensions along which strategy-learning will evolve in the future. Table XIX illustrates such practical issues using the examples developed in Section B.

Table XIX
Examples of practical issues in strategy learning

Example	Issue	Comments
Load Balancing	Resource Constraint	The problem is too complex to test all the strategies exhaustively; rational division of time between competing placement heuristics is essential.
	Debugging	It is difficult to design and debug a system to predict completion time using statistical methods.
	Efficiency	Stochastic strategies and evaluative feedback require large number of instances but less time per instance.
	Quality	Optimality is provably intractable; the design should aim only to improve speed-up over a reference case (such as one without load balancing).
	Generality	It is difficult to extend the strategies learned and tested on a small network to larger networks; there is no generalization across problems.
Steering a Ship	Resource Constraint	Symbolic events and complex rules limit episode size, effectively removing resource limitations on learning.
	Debugging	Explanatory trace from knowledge-rich strategy learner simplifies debugging.
	Efficiency	Knowledge of causal models and temporal variations reduces the number of learning episodes but increases the time per episode.
	Quality	Quality is judged by the solution length; *chunking* (collapsing rule sequences) produces near-optimal solutions.
	Generality	With only a few task-specific rules, the system is extremely brittle; with additional rules describing common-sense physical laws, generality can be increased at the cost of efficiency.
Pole Balancing	Resource Constraint	Given deadlines, learning should limit exploration and converge faster, at the risk of learning a suboptimal strategy.
	Debugging	Strategies learned using statistical (associative learning) methods are difficult for humans to interpret and modify.
	Efficiency	Learning requires too many trials; the amount of work per trial is small; therefore, this application is suitable for on-line learning.
	Quality	Asymptotic convergence to the optimal strategy is guaranteed when convergence is sufficiently slow.
	Generality	Generalization to similar instances is spontaneous; to new problems, difficult because of inflexible decision structures.
Towers of Hanoi	Resource Constraint	Episodes of this provably complex problem can be lengthy; these force consideration of the exploration-convergence trade-off.
	Debugging	Symbolic representations are analyzed easily by humans, but not necessarily by the algorithms used by the problem solver.
	Efficiency	Symbolic generalization requires relatively few trials, but each trial may be quite complex.
	Quality	Quality is judged by the number of states explored relative to the incumbent strategy.
	Generality	Powerful heuristics are problem-specific but not instance-specific.

1) Implementation issues

Designers of learning systems must take into account realistic constraints on time and other computational resources in developing, debugging, and testing computer programs for strategy learning. This section reviews the prevalent approaches to these problems.

a) *Resource Constraints*. Table XIX shows that resources for strategy learning might be subjected to a variety of constraints. For example, in the load-balancing task (Example 1), a large amount of testing time is required for each test case, which comprises multiple jobs and testing a strategy on multiple test cases. Such learners must sample the space of alternatives intelligently, ensuring that alternatives yielding high-quality solutions are given more time while simultaneously attempting to explore a large number of alternatives to gain confidence in the strategies learned. At an abstract level, this may force the strategy learner to first form estimates of eventual execution cost and solution quality, and then trade between these to learn a good strategy with high confidence.

The TEACHER architecture of Wah *et al*. [87, 164, 184] addresses resource-constrained strategy learning; it includes a resource scheduler that trades between quality and cost. Formal models for the design of resource-constrained rational learning agents, using decision analysis to estimate the utility and cost of alternatives, have been proposed by Etzioni [54], and Russell *et al*. [155, 190].

b) Analysis and Debugging. No general-purpose debugging tools for strategy-learning systems are available yet. Even so, it is easier to design knowledge-based learning systems. These systems often produce intelligible traces of their behavior, which include justifications for proposed actions and explanation of unanticipated errors. Their representations are human readable, simplifying their interpretation and debugging. Dietterich and Buchanan's architecture, owing to its use of knowledge-based learning, is the winner on this issue. This architecture can be used in learning strategies for both the ship-steering (Example 2) and the Towers of Hanoi (Example 4) problems (see Table XIX). However, it is difficult to evaluate statistically the performance of knowledge-based systems on account of their brittleness (poor generalization). Complex analytical techniques are needed for proving that such systems can indeed learn strategies for all the instances of a problem.

On the other extreme, systems based on Minsky's model do not usually produce detailed explanations of their behavior. This architecture can be used in learning strategies for load balancing (Example 1) and pole balancing (Example 3). (See Table XIX.) The performance of such systems can, however, be

analyzeded using techniques from decision analysis and computational learning theory.

c) Parallel Implementation. Two of the four architectures described in the previous section — Minsky's model and the TEACHER model — lend themselves to parallel implementation. Systems based on Minsky's model, especially those based on neural networks [4, 173], are characterized by fine-grained local computations and can be carried out efficiently on regular massively parallel computers [38]. Systems based on TEACHER yield a somewhat coarser grain of parallelism, which is suitable for implementation on distributed-memory systems [89].

2) Evaluation of strategy-learning systems

We have seen in the foregoing that the choice of a learning system depends upon various aspects of the given strategy-learning task. However, problems can be reformulated; problem-solving strategies and operators, redesigned; and the feedback structure of the learning environment, changed. Realistically, therefore, an application engineer may have a choice between several designs for a strategy-learning system. One may compare strategy-learning systems on the basis of their efficiency, quality, and generality.

a) Efficiency. The efficiency of a strategy-learning system is defined by both the number of instances needed for learning and the time taken per instance. For general-purpose learning systems based on Dietterich and Buchanan's model, inefficiency is a problem due to the need for symbolic manipulations (which are slower than numerical ones), as well as the need for matching new instances against a database of goal-indexed solutions. These systems take too much time per instance but are able to learn using fewer instances. Table XIX illustrates this fact for the ship steering (Example 2) and the Towers of Hanoi (Example 4) tasks. On the other hand, systems based on Minsky's model use stochastic strategies to learn from evaluative feedback; they require too many training instances to converge, but require only simple computational steps per instance. Table XIX illustrates this property for the load-balancing (Example 1) and the pole-balancing (Example 3) tasks. In between are systems that learn from prescriptive feedback, which apply error-correction or gradient descent to compute changes to the various strategy

parameters; they need less time per sample than knowledge-based learners, and fewer samples than stochastic ones.

Recent attempts to overcome inefficiency include the incorporation of look-ahead mechanisms in stochastic search [177, 196], as well as reductions in the cost of symbolic matching [76, 127]. Systematic improvements in efficiency can also be obtained by goal-directed partitioning of the state space into regions of similar utility [150]. Moreover, one can learn *dominance rules* that prune out suboptimal illustrates this property for the load-balancing (Example 1) and the pole-balancing (Example 3) tasks. In between are systems that learn from prescriptive feedback, which apply error-correction or gradient descent to compute changes to the various strategy parameters; they need less time per sample than knowledge-based learners, and fewer samples than stochastic ones.

Recent attempts to overcome inefficiency include the incorporation of look-ahead mechanisms in stochastic search [177, 196], as well as reductions in the cost of symbolic matching [76, 127]. Systematic improvements in efficiency can also be obtained by goal-directed partitioning of the state space into regions of similar utility [150]. Moreover, one can learn *dominance rules* that prune out suboptimal alternative strategies [54, 202].

b) Quality. The quality of strategy learning is judged by the quality of solutions produced by the strategies learned. For optimization problems, the quality of solutions is judged by the degree of optimality as well as the length of the solution; that of *satisficing* problems (where the objective is a truth-valued function), by the length of the solution alone. Table XIX shows that the load-balancing (Example 1) and the pole-balancing (Example 3) tasks are of the first type; and the ship-steering (Example 2) task, of the second type. Techniques such as dynamic programming and its variants can often produce optimal or near-optimal strategies for learning from evaluative feedback when the Markovian property holds. In general, there is a trade-off between quality and efficiency of strategy learning: producing strategies of high quality requires large sample sizes. This trade-off is critical to strategy-learning systems under resource constraints, where learning must be combined with intelligent sampling in order to efficiently

learn strategies of high quality.

A key problem in assessing the relative quality of alternative strategies is that of normalization [185]. One possibility is to compare alternatives on a case-by-case basis; this approach yields a large number of comparisons that need to be aggregated in order to produce a scalar performance metric. Another is to compare the aggregate performance of one alternative on a large number of cases against the aggregate performance of another on the same set of cases. Of these two, the first is a better indicator of expected relative improvement in solving an unseen random problem instance. When both cost and quality need to be compared, it is best to compare them separately. In optimization problems, one may be interested in finding the best solution within certain bounds on cost; in satisficing problems, one may only be interested in a solution of satisfiable quality in the shortest possible time. Both these desiderata can be accommodated by a normalization scheme that separately normalizes cost and quality on a case-by-case basis.

c) Generality. Every learning system is expected to generalize what it learns during training to instances it has not seen before. Strategy-learning systems for general problem solving must generalize not only to new instances but also to new problems. There is a sharp divide on this account between systems based on Minsky's model on the one hand, and those based on Dietterich and Buchanan's model on the other. While the former generalize spontaneously to new instances, their generality to new problems is severely limited by their static representation; on the other hand, the latter are brittle in terms of generalizing to new instances, even though they are somewhat better at generalizing to new problems. Table XIX shows that typical symbolic problem solvers, such as the ones in the ship-steering (Example 2) and the Towers-of-Hanoi (Example 4) tasks, are capable of generalizing across tasks provided that they are given sufficient general knowledge.

Designers of strategy-learning systems need to address practical considerations such as resource constraints, ease of debugging, and fast learning. It is likely that future systems will employ hybrid reasoning and representation techniques in order to combine the benefits of knowledge-based and statistical

methods.

G. Chapter summary

A strategy-learning task is a triple comprising a performance task, a problem solver, and a learning environment. Performance tasks are characterized by their variables, objectives, and constraints; problem solvers, by their representations, operators, and strategies; and learning environments, by their feedback and knowledge-intensity. Strategy-learning tasks drawn from diverse fields can be classified using this taxonomy.

The issues of interest to the designers of a strategy-learning system are functions of the structure of its learning task. Complex learning tasks are characterized by ill-posed objective functions, delayed feedback, violation of the Markovian property, dynamic decision making, time-varying problem variables, and knowledge-lean domain.

A clean separation of issues from applications allows us to address techniques for strategy learning independent of the specific learning systems. General-purpose approaches, such as dynamic programming, regression, and time-series analysis, are shown to be useful. Non-Markovian tasks, for which no general purpose techniques are known, require new approaches to update dynamically history information and perform rational credit assignment on stored solutions. The concepts of causal and temporal scopes are introduced as prerequisites for general-purpose credit-assignment procedures.

The numerous issues and approaches can be abstracted into four general architectures: Dietterich and Buchanan's model, Minsky's model, the hybrid model and the TEACHER model. The first is useful for learning static strategies in knowledge-rich environments; the second, for learning dynamic strategies for Markovian problems in knowledge-lean environments; the third, for learning dynamic strategies for non-Markovian problems; and the fourth, for resource-constrained learning of procedurally encoded strategies using generate-and-test approaches.

To develop more powerful strategy-learning systems, we need to address practical issues regarding implementation, such as limited resources and efficacy of design, and those regarding evaluation, such as methods for characterizing the quality, efficiency, and generality of the techniques. Some issues are specific with specific techniques, such as the slowness of reinforcement learning, the difficulty in generalizing the structure of strategies in explanation-based learning, and methods for nonlinear prediction and regression.

CHAPTER VI.

POPULATION-BASED LEARNING OF LOAD-BALANCING POLICIES

Experience is one thing you can't get for nothing.

- Oscar Wilde

A. The policy-learning problem

Figure 3 (Chapter I) shows the sender-side and receiver-side rules for the load-balancing policies considered in this thesis. There is one set of rules per site. The rules are parameterized; the number of parameters depends upon the number of sites in the distributed system. Let *N* denote the number of sites. The sender-side rules (SSRs) of each site have 3 parameters: two thresholds — θ_1 and δ — and the `Reference` parameter. The receiver-side rules (RSRs) of each site have 1 parameter: the threshold θ_2. Altogether, there are 4 parameters per site, or 4*N* parameters for the whole system. In this chapter, we use the term *heuristic method (HM)* [185] to denote a set of system-wide parameters.

Figure 45 shows our representation of HMs. Later in the chapter, we show that genetic operators, such as mutation and crossover, can be applied to such representations in order to generate new HMs from a population of existing HMs.

Reference(1), $\theta_1(1)$, $\delta(1)$, $\theta_2(1)$
Reference(2), $\theta_1(2)$, $\delta(2)$, $\theta_2(2)$
...
Reference(*N*), $\theta_1(N)$, $\delta(N)$, $\theta_2(N)$

Figure 45. Representation of heuristic methods

The load-balancing policies are applied in a distributed fashion. When a job arrives at a site *i*, it triggers that site's SSRs. Using the information about the local and remote load indices, the SSRs determine a set of possible destinations, which includes all sites whose load indices lie within a δ(i)-neighborhood of `Reference`(i). Out of these, one destination (say, *j*) is picked at random. Depending upon the value of `Reference`(*i*), the SSRs compare either the absolute or the relative value of `Load`(*i*) against the threshold $\theta_1(i)$. If `Load`(*i*) is large enough, then the rule for sending succeeds, and the SSRs invoke the RSRs at site *j*; otherwise, the job is executed locally at site *i*. The RSRs at site *j* compare `Load`(*j*) against the threshold $\theta_2(j)$. The request from the SSRs at site *i* is turned down if `Load`(*j*) is too high; in this case, the job is executed locally at site *i*. Otherwise, the request is accepted, and the remote site (*i*) is given permission to migrate its job. Thus, successful migration requires, first, that the SSRs at the site of arrival succeed, and, second, that the RSRs at the chosen remote destination also succeed. Also see Figure 11 and the discussion of the policy-learning and application phases in Chapter II.

Randomization in selection of remote destinations avoids instability in distributed scheduling algorithms [171]. The threshold θ_1 in SSRs prevents migration when the expected gains in completion time are not large enough to offset the overheads of migration. The threshold θ_2 in RSRs is less critical, but it can help counter bad decisions by senders who overestimate their own load.

Because of the measurements conducted during the application 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, 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. As described above, first the SSRs are evaluated at the site of arrival; then, if necessary, the RSRs are evaluated for some remote destination. 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.

The total completion time of a migrated task equals the sum of the overhead and the remote completion time. The ratio of total completion time to the local completion time gives the speed-up with respect to local execution. Such speed-ups are the basic units of an HM's performance.

We organize the data collected ahead of time into test-cases. Each test-case carries information about one incoming job, its time of arrival, and the load indices of different sites at the time of arrival; it also carries information about the measured completion time of that job at each of the N sites. We use the word *test* in this chapter to denote the evaluation of a selected HM on one test-case. Since our experiments are performed off-line, 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 quite large. We must search this space intelligently and in limited time (i.e., 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* [165]: 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.* [185, 186], whose TEACHER (TEchniques for Automated Creation of HEuRistics) system [88, 185] provides a domain-independent approach to rational scheduling of limited learning time between generation and testing.

B. Brief overview of TEACHER

We use the TEACHER system for the policy-learning phase of SMALL. This section first describes TEACHER and then the interface between it and SMALL.

1) Architecture and operation of TEACHER

Figure 46 shows the overall architecture of TEACHER. The key component of TEACHER is the scheduler, which decides whether to generate new HMs 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 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 quality (sample mean) of an HM may be high, our confidence in that quality will be low. The advantage of continuing to test existing HMs is that our estimate of their quality

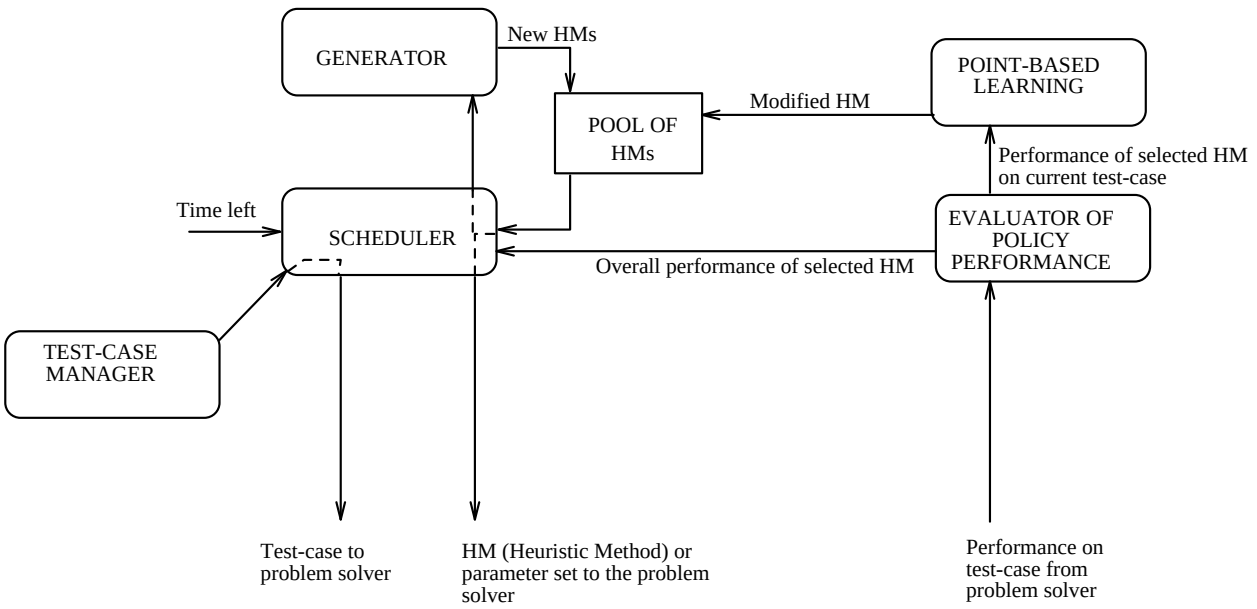


Figure 46. The architecture of TEACHER

improves with the number of tests. TEACHER divides up its total learning time into *generations*. 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 for the application phase.

Within each generation, tests need to be allocated rationally between different HMs. The traditional method of solving this problem is to apply decision-theoretic sampling techniques developed for the ranking and selection problems [20, 77, 183]. However, these techniques have several limitations: (i) they only work with a small (and known) number of populations, whereas the number of possible HMs is large and unknown ahead of time; (ii) they do not accommodate generation of new HMs; and (iii) they optimize the probability of finding the best population, rather than finding a population whose mean is within a certain range of the optimal mean. TEACHER offers a suite of test-allocation strategies that overcome the aforementioned limitations.

Besides supporting generation and tesing of HMs, TEACHER also accommodates point-based methods, in which an HM is refined each time it is tested; such refinement is based on prior knowledge of relationships between the problem solver's inputs and outputs. Since we have only limited prior knowledge of the load-balancing domain, refinement by point-based learning can only be applied to some of the parameters.

For operational details of the policy-learning phase, please refer to Chapter II. The rest of this section describes the domain-dependent functions of SMALL, which are required by the TEACHER system. These include (i) genetic operators for generation of new candidates; (ii) an algorithm for point-based modification of thresholds; and (iii) mechanisms for simulating the problem solver, thereby providing performance information useful for evaluation of test-cases.

2) Operators for population-based learning

Figures 47 and 48 illustrate the genetic operators used in the generation of new HMs from existing ones. The initial HMs can either be generated randomly or provided by the designer. There are three operators in our system: (i) *Mutation*: the `Reference` value of a randomly chosen site is changed from `0` to `MinLoad`, or vice versa; (ii) *Crossover-1*: given a pair of HMs, two new HMs are generated by interchanging the substrings (of the old HMs) containing the parameters of a randomly chosen site; (iii) *Crossover-half*: similar to Crossover-1, except that the exchange involves $N/2$ sites.

SITENAME	**Reference**	Θ_1	Θ_2	δ		**SITENAME**	**Reference**	Θ_1	Θ_2	δ
calvin	0	0.394	0.720	0.459		calvin	0	0.394	0.720	0.459
vyasa	**minload**	0.051	0.411	0.088	→	vyasa	**0**	0.051	0.411	0.088
....						....				
elaine	0	0.77	0.722	0.316		elaine	0	0.77	0.722	0.316

Figure 47. An example of mutation applied to an HM

SITENAME	**Reference**	Θ_1	Θ_2	δ						
calvin	0	0.394	0.720	0.459		calvin	minload	0.131	0.809	0.071
vyasa	**minload**	**0.051**	**0.411**	**0.088**	×	**vyasa**	**minload**	**0.093**	**0.655**	**0.065**
....						....				
elaine	0	0.77	0.722	0.316		elaine	minload	0.325	0.879	0.023

↓

calvin	0	0.394	0.720	0.459	calvin	minload	0.131	0.809	0.071
vyasa	**minload**	**0.093**	**0.655**	**0.065**	**vyasa**	**minload**	**0.051**	**0.411**	**0.088**
....					....				
elaine	0	0.77	0.722	0.316	elaine	minload	0.325	0.879	0.023

Figure 48. An illustration of the crossover operation

Following the customary practice in genetic search [71], 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.

3) Algorithm for point-based learning

Since the completion times and overheads for all possible destinations are known ahead of time, the learning system knows the optimal decision for every decision point. Therefore, it 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 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 behavior 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 behavior 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 Figure 49.

Adjustment of δ:

- If 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 (i) 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 (i) in order to exclude the sub-optimal site.
- IF (optimal destination not in set of possible destinations) THEN
  $\delta(i) \leftarrow \delta(i) + \eta \cdot \varepsilon_1$
  ELSE
  $\delta(i) \leftarrow \delta(i) - \eta \cdot \varepsilon_2$.

Adjustment of $\theta_1(i)$:

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

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

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

Figure 49. Rules for point-based learning

C. Data Preparation

The sole purpose of the data-preparation phase of policy learning is to replace an actual problem solver with a table-lookup routine, as shown in Figure 10. Given the names of 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.

The advantage of off-line data collection using DWG is really obvious here. The learning system may try hundreds of different policies, but with respect to each given test-case, these policies must fall

into *N* equivalence classes, each corresponding to one destination. All policies within an equivalence class are indistinguishable with respect to the given test-case. Each test-case contains information about one test-job and one background-load pattern; therefore, in order to to have information sufficient for evaluating all possible policies over all the test-cases, we should have data about the completion times of all the test-jobs run under all possible loading patterns and on each of the possible destinations. Such data are collected during the second phase of data collection; they can be used for evaluating any set of policy parameters considered by the policy-learning system.

An even greater advantage of our learning method is the facility it affords us for simulating (i) scaled-up systems; (ii) the effect of migration overheads; (iii) the effect of packet-transmission delays; and (iv) the effect of periodic computation of load averages. The following subsections address these issues in greater detail.

1) Cloning: A method for simulating larger systems

All our sites are assumed to be architecturally homogeneous; however, we do allow configurational heterogeneity. As explained in Chapter II, only configurationally distinct sites need to be measured, for configurationally identical sites behave identically. 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 of simulating new sites is called *cloning*: it creates load patterns distributed in space from load patterns distributed in time. Test jobs behave identically across different clones under the same loading pattern; therefore, 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. Thus, the ability of the learning system to discover HMs with near-optimal performance can be tested more thoroughly with cloning than without.

2) Modeling of overheads and delays

Figure 50 shows on top a trace of the load-index function at the arrival site of an incoming job and, at the bottom 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 Δ1 and Δ3 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 Δ2 includes both the delay in computing the load index at the remote site and the delay

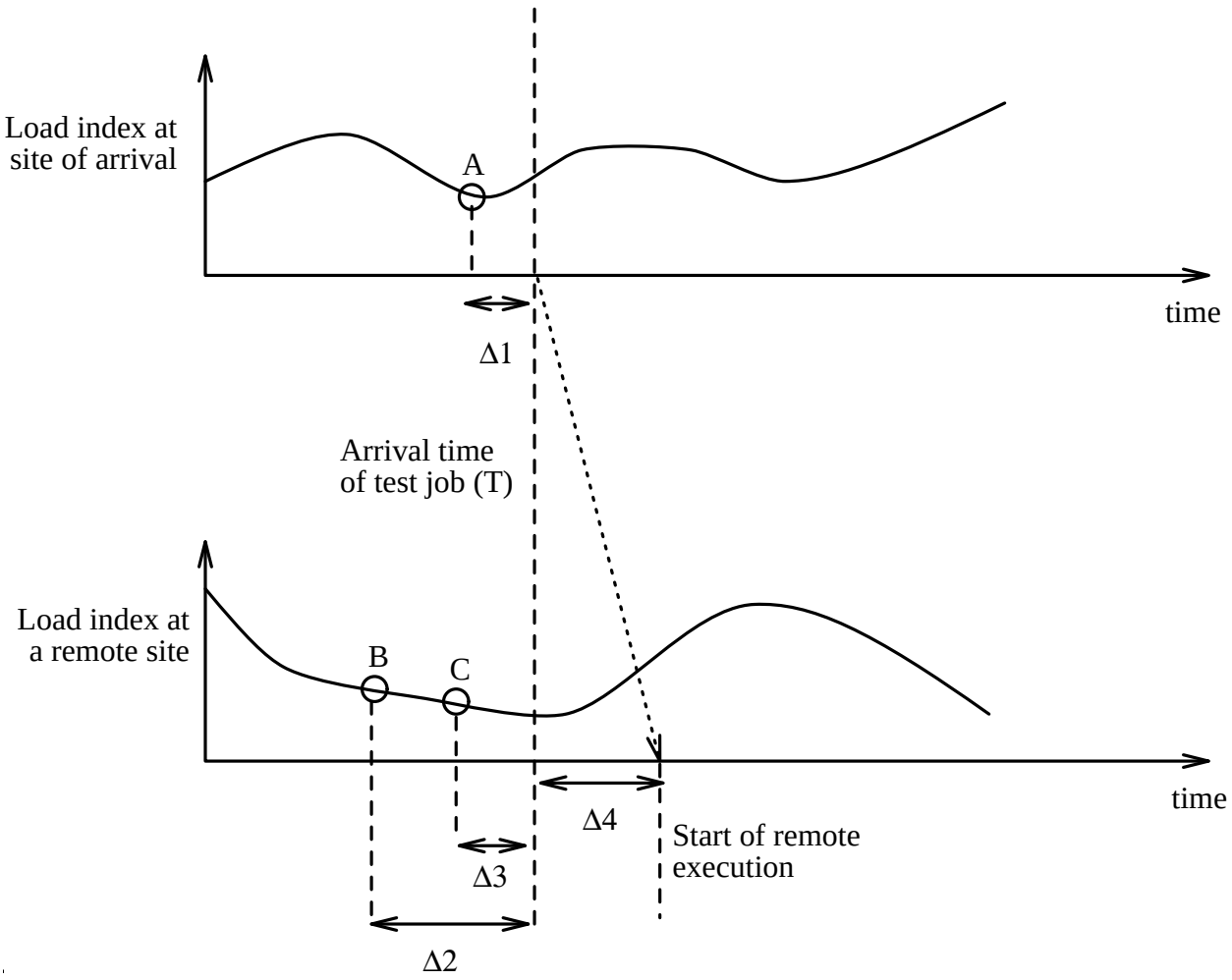


Figure 50. Overheads and delays in load balancing

in propagating that value to site of arrival. Finally, the delay $\Delta 4$ represents the overhead of migration.

It is not enough to add $\Delta 4$ to the completion time of every test job, for in our (off-line) 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 on-line experiment; the difference between on-line (actual) and off-line (simulated) cases is proportional to $\Delta 4$. If $\Delta 4$ is large, the load could have been very different; if small, then less so. The effect of migration overhead is simulated by adding $\Delta 4$ to the age of the remote load index.

Consider a typical load-index function. If the feed-forward 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 $\Delta 1$ and $\Delta 3$ are random variables distributed uniformly over the interval $[0,\tau]$.

One final factor that needs to be considered is $\Delta 2$, the delay including the time for propagation of load values to remote sites. Assuming that load values are transmitted over the network each time they are computed, the propagation delay can be modeled by half the round-trip time of a short packet. Round-trip times can be measured off-line for a network. Delays in propagation 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 upto 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.

We conducted several learning experiments involving various combination of these factors. The next section presents our results.

D. Empirical Results

Using the data for our four-processor system, we created a total of nine clones (3 each for the two diskless clients, 2 for the diskful client, and 1 for the fileserver). 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 on our network, we found it adequate to assume that load information from remote sites can be received within 20 milliseconds (P=20) of being broadcast. We studied four different migration overheads (M): (i) small, 100 milliseconds; (ii) medium, 500 milliseconds; (iii) moderately large, 2000 milliseconds; and (iv) large, 5000 milliseconds. In addition, we considered three different intervals (I) of load-index computation: (i) small, 100 milliseconds; (ii) medium, 500 milliseconds; and (iii) large, 1000 milliseconds. Table XX shows the values of M and I in the left two columns. For each combination of M, I, and P values, a different set of data are prepared as follows. To

Table XX
Performance of policies learned under different assumptions

Migration overhead (M) (ms)	**Interval (I) of load-index computation (ms)**	**Average performance (speed-up over local execution) of policy learned**	**Standard deviation of speed-up**
100	100	1.73743	0.226114
	500	1.49916	0.219608
	1000	1.50908	0.203219
500	100	1.50156	0.179138
	500	1.53047	0.152211
	1000	1.6004	0.17311
2000	100	1.2397	0.0694498
	500	1.15947	0.112387
	1000	1.17821	0.124843
5000	100	1.25181	0.128558
	500	1.19149	0.14072
	1000	1.31023	0.134078

the age of each load index, we add a random delay uniformly distributed between 0 and I. Only for the remote load indices, we add the factors M and P. The load indices are determined using load-average traces similar to those shown in Chapter IV (Figures 31 and 35).

TEACHER is allowed 30 minutes learning time for each experiment on a Sparcstation 10, Model 20. Twelve experiments are performed, one for each parameter set described in Table XX. The best policy found by TEACHER after 30 minutes is fully evaluated on all the test-cases. The average speed-up of that policy and the corresponding standard deviation are shown in the third and fourth columns of Table XX. The best policy is found when both M and I are small. While performance appears to decrease with increasing migration overheads, no such effect is visible for the interval of load-index computation. Figure 51 shows a contour plot of the best policy learned. (The contour plots of this chapter include migration overheads, unlike those of the policy min, in Chapter IV. The axes and contours should be interpreted in a similar fashion as before.) This figure shows that the best speed-up values are achieved for jobs that are neither too long nor too short. As shown in Chapter IV, 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 preemptive 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 52 shows the behavior of policies learned under high migration overheads but small interval of load-index computation. Relative to Figure 51, high speed-up values are less probable when migration overhead is significant. Figure 53 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 XX, but they are much more likely than in the case of high migration overheads.

We conclude (i) that TEACHER discovers policies with significant speed-ups over local execution under a variety of assumptions about overheads and delays; (ii) the quality of policies learned by TEACHER 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.

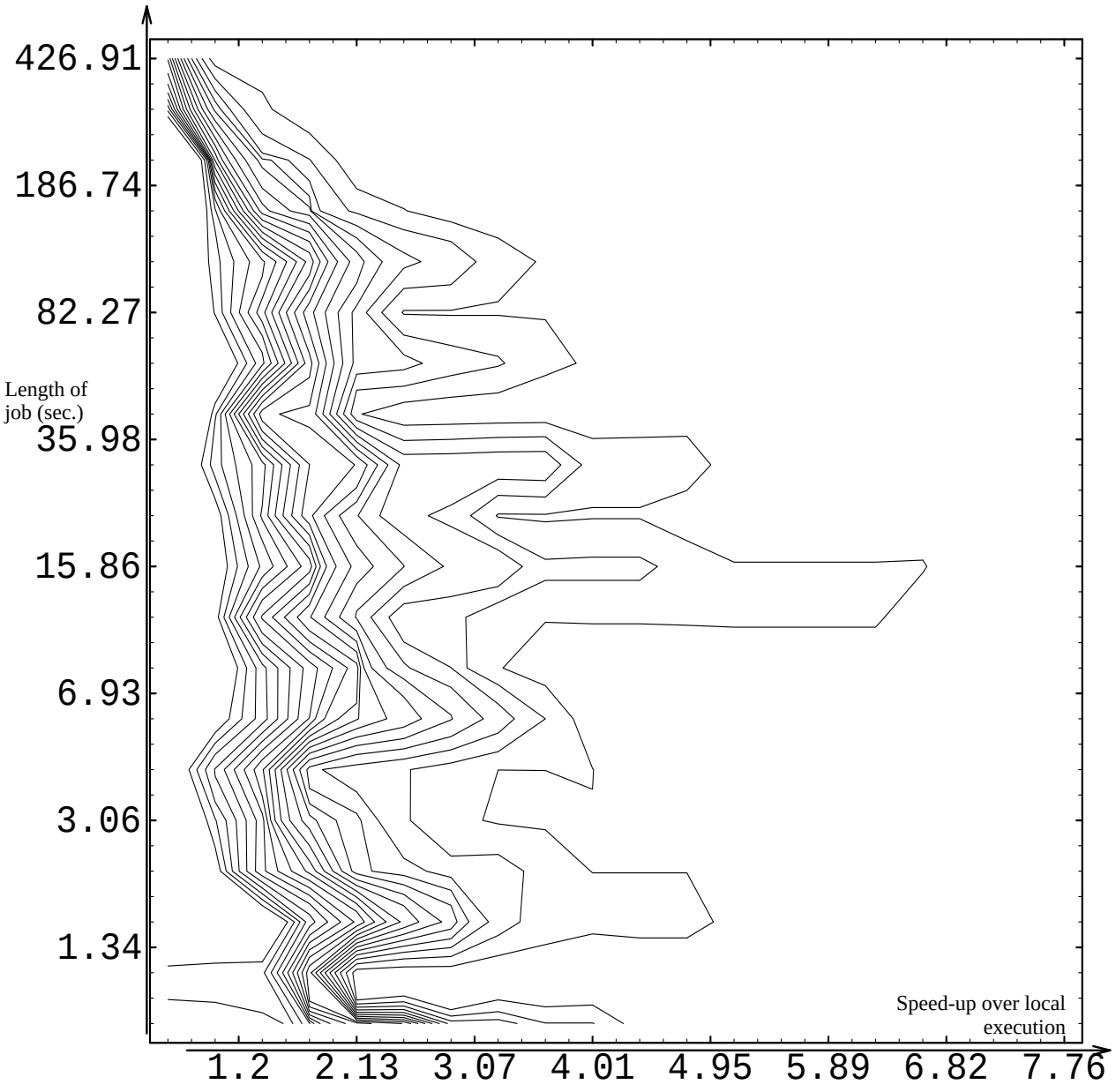


Figure 51. Contour plot of cumulative probability of speed-up; M=100,I=100,P=20

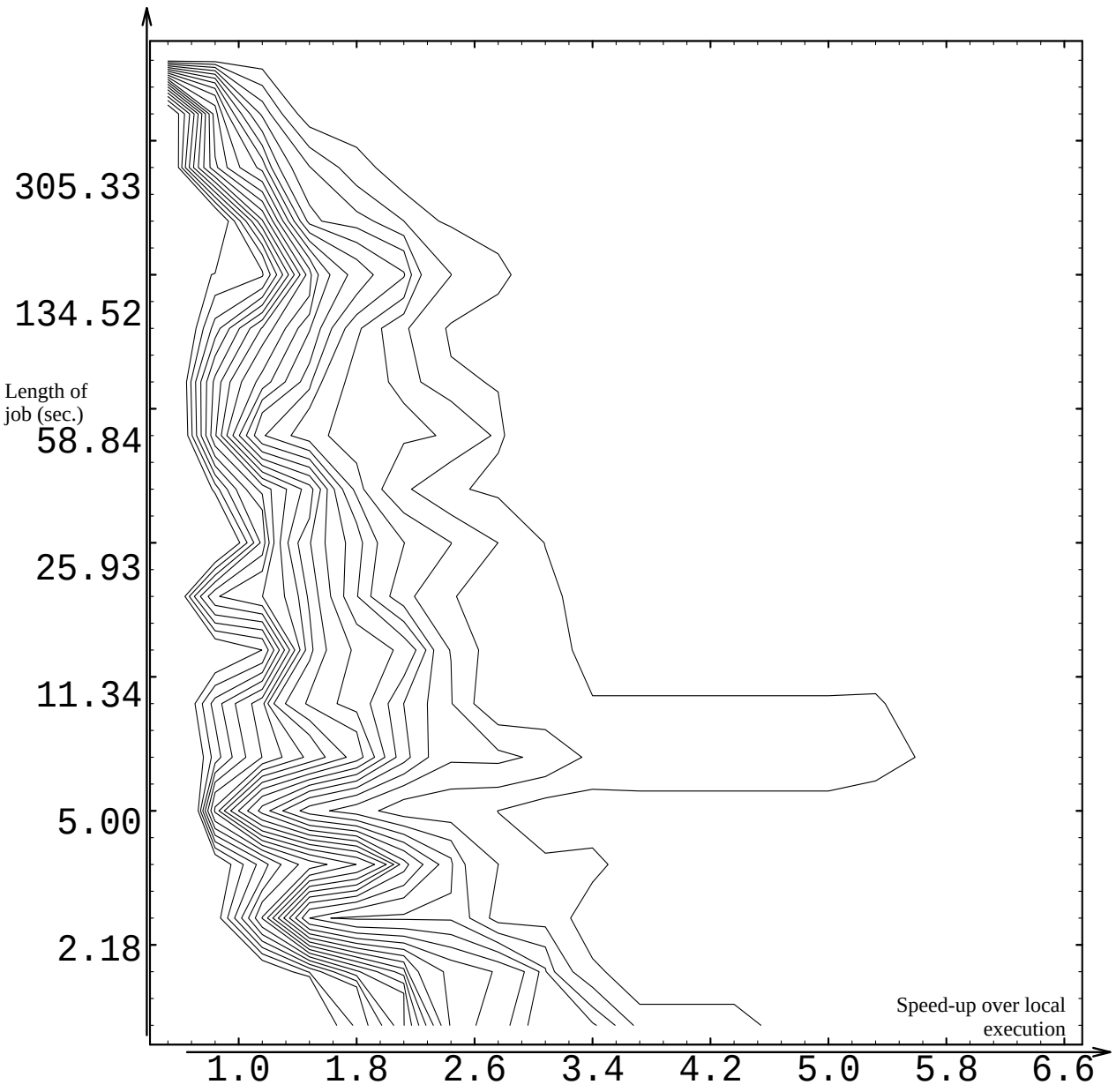


Figure 52. Contour plot of cumulative probability of speed-up; M=5000,I=100,P=20

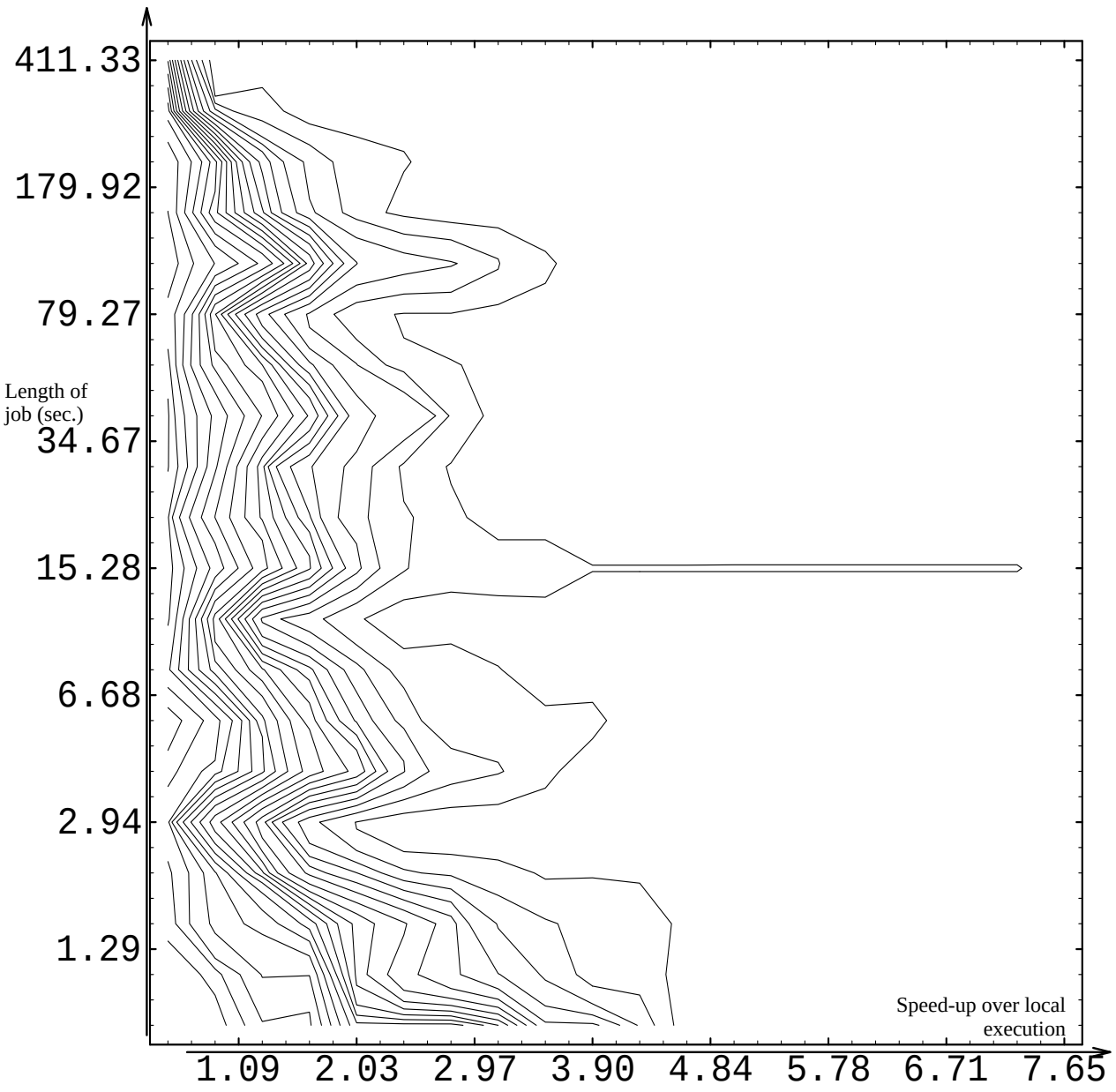


Figure 53. Contour plot of cumulative probability of speed-up; M=100,I=1000,P=20

Finally, 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 TEACHER finds a policy with a speed-up of 1.73, we can appreciate the utility of integrated learning.

E. Chapter Summary

Parameter sets of load-balancing policies can be generated using genetic operators and modified systematically using point-based learning. Selective breeding of well-performing parameter sets at the end of each generation, and rational allocation of tests among the generated parameter sets during each generation, are the key functions supported by the TEACHER system. Using our integrated (population-based cum point-based) approach, good parameter sets for load-balancing policies can be discovered quickly and in an automated fashion.

CHAPTER VII.

CONCLUSIONS

We have successfully demonstrated an automated process for improving the performance of parameterized load-balancing strategies. Our learning system, SMALL, discovers new load indices that can be meaningfully compared across the sites of a configurationally heterogeneous but architecturally homogeneous distributed system.

SMALL's experimentation environment represents a significant departure from the traditional way of conducting load-balancing experiments. The new set-up substantially limits both the duration of each experiment and the number of experiments needed for off-line evaluation of competing policies.

During policy learning, we evaluate each decision in isolation. At an abstract level, we have converted dynamic load balancing from a dynamic decision problem into a static decision problem. This transformation obviates temporal credit assignment, an important unsolved problem in strategy learning.

The centerpiece of our research is DWG, a synthetic workload generator that allows the measurement of task-completion times under precisely controlled and reproducible loading conditions. In order to gain precision, we had to implement DWG inside the operating-system kernel. In doing so, we faced a problem of modeling the interactions between foreground and background workloads. We solved this problem using *dynamic doctoring*, a systematic method for reducing the generated load in the presence of foreground processes while simultaneously honoring the principle of conservation of work.

By posing performance-driven adaptation of load-balancing as a strategy-learning task, we are able to (i) study its characteristics; (ii) isolate clean, general issues; and (iii) either develop, adopt, or borrow general techniques for addressing these issues. An important characteristic of our system is that it was put together using the following general tools that work equally well for related applications.

Comparator neural networks. The comparator configuration represents a natural learning solution to problems involving comparison between functions of multivariate time series. Its precursor, the original comparator network, was a champion at selecting moves for the game of backgammon [180]; however, the original version considered only the current state in making comparisons. Our architecture and learning algorithm are suitable for problems that involve comparison of traces. One such problem arises in population-based design of neural networks [184], where one would like to predict which out of several competing network configurations will converge faster, based only on their error traces. The architecture and training algorithm reported in Chapter IV, when applied to this problem, easily achieved scores exceeding 75% correct, thus providing an effective means for pruning out unpromising configurations. In the future, we plan to apply the comparator architecture for early detection of bottlenecks on massively parallel computers, based on workload traces of different sites.

TEACHER. We have added dynamic load balancing to TEACHER's growing list of strategy-learning applications. In the past, this system was used for tuning the parameters of stereo-vision algorithms [163], and in learning heuristics for mapping communicating processes on distributed systems [88].

DWG. Synthetic workload generation is useful not only in scheduling of independent tasks, but also in object-oriented systems, where the problem is to study the behavior of object-placement policies under a variety of background workloads [94].

Our comprehensive survey of strategy learning shows that learning tasks from diverse domains may sometimes exhibit similar characteristics. General techniques developed for one learning task should carry over to the others having similar components: performance tasks, problem solvers, or learning environments. In the future, we plan to use our experience with load balancing to develop automated mechanisms for learning (i) page-prefetching strategies in distributed memory hierarchies; and (ii) instruction-level scheduling strategies in pipelined supercomputers.

REFERENCES

[1] D. H. Ackley, *A Connectionist Machine for Genetic Hillclimbing*, Kluwer Academic Pub., Boston, MA, 1987.

[2] P. E. Agre and D. Chapman, ‘‘PENGI: An Implementation of a Theory of Activity,’’ *Proc. National Conf. Artificial Intelligence*, pp. 268-272, Morgan Kaufman, Palo Alto, CA, June 1987.

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

[4] C. W. Anderson, *Learning and Problem Solving with Multilayer Connectionist Systems*, Ph.D. Thesis, Univ. of Massachusetts, Amherst, MA, 1986.

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

[6] J. R. Anderson, J. G. Greeno, P. J. Kline, and D. M. Neves, ‘‘Acquisition of Problem-Solving Skill,’’ in *Cognitive Skills and their Acquisition*, ed. J. R. Anderson, Lawrence Erlbaum Associates, Hillsdale, NJ, 1981.

[7] J. R. Anderson, ‘‘Knowledge Compilation: The General Learning Mechanism,’’ in *Machine Learning: An Artificial Intelligence Approach*, ed. Michalski, *et al*., Morgan Kaufmann, 1986.

[8] P. M. Andreae, ‘‘Constraint Limited Generalization: Acquiring Procedures from Examples,’’ *Proc. National Conf. Artificial Intelligence*, pp. 6-10, AAAI, Inc., Austin, TX, 1984.

[9] Y. Anzai, ‘‘Doing, Understanding, and Learning in Problem Solving,’’ in *Production System Models of Learning and Development*, ed. Klahr, *et al*., MIT Press, Cambridge, MA, 1987.

[10] Y. Artsy (ed.), ‘‘Special Issue on Process Migration,’’ *Operating Systems Technical Committee Newsletter*, vol. 3, no. 1, pp. 4-34, IEEE Computer Society, Winter 1989.

[11] M. J. Bach, *The Design of the UNIX Operating System*, Prentice-Hall, Englewood Cliffs, NJ, 1986.

[12] R. Barletta and R. Kerber, ‘‘Improving Explanation-Based Indexing with Empirical Learning,’’ *Machine Learning*, pp. 84-86, Kluwer Academic Pub., Boston, MA, 1989.

[13] A. Barr and E. A. Feigenbaum, *The Handbook of Artificial Intelligence*, vol. 1, 2, and 3, William Kaufmann, Los Altos, CA, 1981, 1982.

[14] A. G. Barto, R. S. Sutton, and C. W. Anderson, ‘‘Neuronlike Adaptive Elements that can Solve Difficult Learning Control Problems,’’ *Trans. on Systems, Man and Cybernetics*, vol. SMC-13, no. 5, pp. 834-846, IEEE, 1983.

[15] A. G. Barto, R. S. Sutton, and C. J. C. H. Watkins, ‘‘Learning and Sequential Decision Making,’’ pp. 539-602 in *Learning and Computational Neuroscience: Foundations of Adaptive Networks*, ed. M. Gabriel and J. Moore, MIT Press, Cambridge, MA, 1990.

[16] A. G. Barto, S. J. Bradtke, and S. P. Singh, *Real-Time Learning and Control using Asynchronous Dynamic Programming*, Tech. report 91-57, Dept. of Computer Sc., Univ. of Massachusetts,

Amherst, MA, 1991.

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

[18] K. Baumgartner and B. W. Wah, ‘‘Computer Scheduling Algorithms: Past, Present and Future,’’ *Information Sciences*, vol. 57 & 58, pp. 319-345, Elsevier Science Pub. Co., Inc., New York, NY, Sept.-Dec. 1991.

[19] K. M. Baumgartner, *Resource Allocation on Distributed Computer Systems*, Ph.D. Thesis, School of Electrical Engineering, Purdue Univ., West Lafayette, IN, May, 1988.

[20] R. E. Bechhofer, ‘‘A Single-Sample Multiple Decision Procedure for Ranking Means of Normal Populations with Known Variances,’’ *Ann. Math. Statist.*, vol. 25, no. 1, pp. 16-39, Institute of Mathematical Statistics, Ann Arbor, MI, March 1954.

[21] R. Bellman and S. Dreyfus, *Applied Dynamic Programming*, Princeton Univ. Press, Princeton, NJ, 1962.

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

[23] R. B. Bodnarchuk and R. B. Bunt, ‘‘A Synthetic Workload Model for a Distributed File Server,’’ *Proc. SIGMETRICS Conf. on Measurement and Modeling of Computer Systems*, pp. 50-59, ACM, 1991.

[24] L. B. Booker, D. E. Goldberg, and J. H. Holland, ‘‘Classifier Systems and Genetic Algorithms ,’’ in *Machine Learning: Paradigm and Methods*, ed. J. Carbonell, MIT press, 1990.

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

[26] W. L. Brogan, *Modern Control Theory*, Prentice-Hall, Englewood Cliffs, NJ, 1985.

[27] R. M. Bryant and R. A. Finkel, ‘‘A Stable Distributed Scheduling Algorithm,’’ *Proc. 1st Int’l Conf. on Distributed Computing Systems*, pp. 314-323, IEEE, 1981.

[28] B. G. Buchanan and T. M. Mitchell, ‘‘Model-Directed Learning of Production Rules,’’ in *Pattern-Directed Inference Systems*, ed. D. A. Waterman and F. Hayes-Roth, Academic Press, New York, NY, 1978.

[29] L-F. Cabrera, ‘‘The Influence of Workload on Load Balancing Strategies,’’ *USENIX Conf. Proceedings*, pp. 446-458, 1986.

[30] J. G. Carbonell, ‘‘Learning by Analogy: Formulating and Generalizing Plans from Past Experiences,’’ in *Machine Learning*, ed. R. S. Michalski, J. G. Carbonell, and T. M. Mitchell, Tioga, 1983.

[31] J. G. Carbonell and Y. Gil, ‘‘Learning by Experimentation,’’ *Machine Learning*, pp. 256-266, Kluwer Academic Pub., Boston, MA, 1987.

[32] B. Chandrasekaran, ‘‘Towards a Taxonomy of Problem Solving Types,’’ *AI magazine*, pp. 9-17, Winter/Spring 1983.

[33] Y. Chauvin, ‘‘A Back-Propagation Algorithm with Optimal Use of Hidden Units,’’ pp. 519-526 in *Proc. Neural Information Processing Systems*, ed. D. Z. Anderson, American Inst. of Physics, New

York, 1988.

[34] C.-T. Chen, *Linear System Theory and Design*, Holt, Rinehart and Winston, Inc., New York, 1970.

[35] S. A. Chien, ‘‘Learning by Analyzing Fortuitous Occurances,’’ *Machine Learning*, pp. 249-251, Kluwer Academic Pub., Boston, MA, 1989.

[36] S. A. Chien, *An Explanation-Based Learning Approach to Incremental Planning*, Ph.D. thesis, Dept. of Computer Science, Univ. of Illinois, Urbana, IL, 1991.

[37] J. Christensen and R. E. Korf, ‘‘A Unified Theory of Heuristic Evaluation Functions and its Application to Learning,’’ *Proc. National Conf. on Artificial Intelligence*, pp. 148-152, AAAI, Inc., 1986.

[38] L.-C. Chu and B. W. Wah, ‘‘Optimal Mapping of Neural-Network Simulations on Message-Passing Multicomputers,’’ *J. of Parallel and Distributed Computing*, vol. 14, no. 3, pp. 319-339, Academic Press, March 1992.

[39] G. Cybenko, ‘‘Approximation by Superpositions of a Sigmoidal Function,’’ *Mathematics of Control, Signals, and Systems*, vol. 2, no. 4, pp. 303-314, Springer International, New York, 1989.

[40] T. Dean and K. Kanazawa, ‘‘Probabilistic Temporal Reasoning,’’ *Proc. National Conf. on Artificial Intelligence AAAI-88*, pp. 524-528, 1988.

[41] T. Dean and K. Kanazawa, ‘‘A Model for Reasoning about Persistence and Causation,’’ *Compuational Intelligence*, vol. 5, no. 3, pp. 142-150, National Research Council Canada, Toronto, 1989.

[42] T. Dean and G. Siegle, ‘‘An Approach to Reasoning about Continuous Change for Applications in Planning,’’ *Proc. 8th Natl. Conf. Artificial Intelligence*, pp. 132-137, AAAI, Inc., Seattle, Washington, 1990.

[43] G. F. DeJong and R. J. Mooney, ‘‘Explanation-Based Learning: An Alternative View,’’ *Machine Learning*, vol. 1, no. 2, pp. 145-176, Kluwer Academic Pub., 1986.

[44] M. Devarakonda and R. K. Iyer, ‘‘Predictability of Process Resource Usage: A Measurement-Based Study of UNIX,’’ *Trans. on Software Engineering*, vol. 15, no. 12, IEEE, Dec. 1989.

[45] T. G. Dietterich and B. G. Buchanan, *The Role of Critic in Learning Systems*, Tech. Rep. STAN-CS-81-891, Stanford Univ., CA, Dec. 1981.

[46] T. G. Dietterich and J. S. Bennett, ‘‘The Test Incorporation Theory of Problem Solving,’’ *Proc. Workshop on Knowledge Compilation*, pp. 145-159, Dept. of Computer Science, Oregon State Univ.,, Sept. 1986.

[47] P. Dikshit, S. K. Tripathi, and P. Jalote, ‘‘SAHAYOG: A Test Bed for Evaluating Dynamic Load-Sharing Policies,’’ *Software — Practice and Experience*, vol. 19, no. 5, pp. 411-435, John Wiley and Sons, Ltd., May 1989.

[48] R. T. Dimpsey and R. K. Iyer, ‘‘Performance Prediction and Tuning on a Multiprocessor,’’ *Proc. Int’l. Symp. Computer Architecture*, pp. 190-199, ACM/IEEE, 1991.

[49] F. Douglis and J. Ousterhout, ‘‘Transparent process migration: Design alternatives and the Sprite implementation,’’ *Software — Practice and Experience*, vol. 21, no. 8, pp. 757-785, 1991.

[50] N. R. Draper and H. Smith, ‘‘An Introduction to Nonlinear Estimation,’’ in *Applied Regression Analysis*, Wiley, New York, 1981.

[51] D. L. Eager, E. D. Lazowska, and J. Zahorjan, ‘‘Adaptive Load Sharing in Homogeneous Distributed Systems,’’ *Trans. on Software Engineering*, vol. SE-12, pp. 662-675, IEEE, May 1986.

[52] D. L. Eager, E. D. Lazowska, and J. Zahorjan, ‘‘The Limited Performance Benefits of Migrating Active Processes for Load Sharing,’’ *Proc. ACM SIGMETRICS Conf. on Measurement and Modeling of Computer Systems*, pp. 63-72, 1988.

[53] G. W. Ernst and M. M. Goldstein, ‘‘Mechanical Discovery of Classes of Problem-Solving Strategies,’’ *J. of the ACM*, vol. 29, no. 1, pp. 1-23, Jan. 1982.

[54] O. Etzioni, ‘‘Embedding Decision-Analytic Control in a Learning Architecture,’’ *Artificial Intelligence*, vol. 49, pp. 129-159, Elsevier, Amsterdam, 1991.

[55] S. E. Fahlman, ‘‘Faster-Learning Variations on Back-Propagation: An Empirical Study,’’ *Proc. Connectionist Models Summer School*, pp. 38-51, Morgan Kaufmann, Palo Alto, CA, 1988.

[56] S. E. Fahlman and Christian Lebiere, ‘‘The Cascade-Correlation Learning Architecture,’’ pp. 524-532 in *Advances in Neural Information Processing Systems 2*, ed. D. S. Touretzky, Morgan Kaufmann, San Mateo, 1990.

[57] J. A. Feldman and R. F. Sproull, ‘‘Decision Theory and Artificial Intelligence II: The Hungry Monkey,’’ *Cognitive Science*, vol. 1, pp. 158-192, Ablex, Norwood, NJ, 1977.

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

[59] D. Ferrari, ‘‘On the foundations of artificial workload design,’’ *Proc. ACM SIGMETRICS conf. on Measurement and Modeling of Computer Systems*, pp. 8-14, 1984.

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

[61] D. Ferrari and S. Zhou, ‘‘A Load Index for Dynamic Load Balancing,’’ *Proc. Fall Joint Computer Conf.*, pp. 684-690, ACM/IEEE, Nov. 1986.

[62] C. Gao, J. W. S. Liu, and M. Railey, ‘‘Load Balancing Algorithms in Homogeneous Distributed Systems,’’ *Proc. Int’l Conf. Parallel Processing*, pp. 302-306, IEEE, Aug. 1984.

[63] J. Geers, ‘‘A New Generation of Benchmarking,’’ *MIPS Magazine*, pp. 92-98, Feb. 1989.

[64] M. R. Genesereth and N. J. Nilsson, *Logical Foundations of Artificial Intelligence*, Morgan Kaufmann, Los Altos, CA, 1987.

[65] M. P. Georgeff, ‘‘Planning,’’ pp. 359-400 in *Annual Review of Computer Science*, Annual Reviews Inc., Palo Alto, CA, 1987.

[66] M. P. Georgeff and A. L. Lansky, ‘‘Reactive Reasoning and Planning,’’ *Proc. National Conf. on Artificial Intelligence*, pp. 677-82, AAAI, Inc., Seattle, Washington, June 1987.

[67] B. K. Ghosh and P. K. Sen (ed.), *Handbook of Sequential Analysis*, Marcel Dekker, Inc., New York, NY, 1991.

[68] M. Gluck, D. B. Parker, and E. S. Reifsnider, ‘‘Learning with Temporal Deivatives in Pulse-Coded Neuronal Systems,’’ in *Proc. Neural Information Processing Systems*, ed. D. Z. Anderson, American Inst. of Physics, New York, 1988.

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

[70] S. D. D. Goggin, K. E. Gustafson, and K. M. Johonson, ‘‘An Asymptotic Singular Value Decomposition Analysis of Nonlinear Multilayer Neural Networks,’’ *Int’l Joint Conf. on Neural Networks*, Seattle, WA, July 1991.

[71] D. E. Goldberg, *Genetic Algorithms in Search, Optimization, and Machine Learning*, Addison-Wesley Pub. Co., 1989.

[72] D. E. Goldberg, ‘‘Probability Matching, the Magnitude of Reinforcement, and Classifier System Bidding,’’ *Machine Learning*, vol. 5, pp. 407-425, Kluwer Academic Pub., Boston, MA, 1990.

[73] G. C. Goodwin and K. S. Sin, *Adaptive Filtering Prediction and Control*, Prentice-Hall, Englewood Cliffs, NJ, 1984.

[74] K. K. Goswami and R. K. Iyer, *Dynamic Load-Sharing Using Predicted Process Resource Requirements*, Tech. Rep. UILU-ENG-90-2224, Coordinated Sci. Lab., Univ. of Illinois, Urbana, 1990.

[75] R. Greiner, ‘‘Learning by Understanding Analogies,’’ *Machine Learning*, pp. 50-52, Kluwer Academic Pub., 1985.

[76] R. Greiner, ‘‘Towards a Formal Analysis of EBL,’’ *Machine Learning*, pp. 450-453, Kluwer Academic Pub., Boston, MA, 1989.

[77] S. S. Gupta and S. Panchapakesan, ‘‘Sequential Ranking and Selection Procedures,’’ pp. 363-380 in *Handbook of Sequential Analysis*, ed. P. K. Sen, Dekker, NY, 1991.

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

[79] S. Hanks, ‘‘Practical Temporal Projection,’’ *Proc. 8th Natl. Conf. Artificial Intelligence*, pp. 158-163, AAAI, Inc., Seattle, Washington, 1990.

[80] G. E. Hinton, ‘‘Connectionist Learning Procedures,’’ *Artificial Intelligence*, vol. 40, pp. 185-234, Elsevier Science Pub., New York, 1989.

[81] P. G. Hoel, S. Port, and C. J. Stone, *Introduction to Stochastic Processes*, Houghton Mifflin Co., Atlanta, GA, 1972.

[82] J. H. Holland, *Adaptation in Natural and Artificial Systems*, Univ. of Michigan Press, Ann Arbor, MI, 1975.

[83] J. H. Holland, ‘‘Properties of the Bucket Brigade Algorithm,’’ *Proc. Int’l. Conf. Genetic Algorithms and Their Applications*, pp. 1-7, The Robotics Inst. of Carnegie-Mellon Univ., Pittsburgh, PA, 1985.

[84] R. A. Howard, *Dynamic Programming and Markov Processes*, Jon Wiley, London, 1960.

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

[86] T. Ibaraki, ‘‘Enumerative Approach to Combinatorial Optimization,’’ *Annals of Operations Research*, Scientific Pub. Co., Basel, Switzerland, 1988.

[87] A. Ieumwananonthachai, A. N. Aizawa, S. R. Schwartz, B. W. Wah, and J. C. Yan, ‘‘Intelligent Mapping of Communicating Processes in Distributed Computing Systems,’’ *Proc. Supercomputing*

91, pp. 512-521, ACM/IEEE, Albuquerque, NM, Nov. 1991.

[88] A. Ieumwananonthachai, A. Aizawa, S. R. Schwartz, B. W. Wah, and J. C. Yan, ‘‘Intelligent Process Mapping Through Systematic Improvement of Heuristics,’’ *J. of Parallel and Distributed Computing*, vol. 15, pp. 118-142, Academic Press, June 1992.

[89] A. Ieumwananonthachai and B. W. Wah, ‘‘Parallel Statistical Selection in Multiprocessors,’’ *Proc. Int’l Conf. on Parallel Processing*, vol. III, pp. 190-194, Pennsylvania State Univ. Press, University Park, PA, Aug. 1992.

[90] M. I. Jordan, ‘‘Supervised Learning and Systems with Excess Degrees of Freedom,’’ *Proc. Connectionist Models Summer School*, pp. 62-75, Morgan Kaufmann, Palo Alto, CA, 1988.

[91] L. Kaebling, ‘‘An Architecture for Intelligent Reactive Systems,’’ in *Reasoning about Actions and Plans*, ed. M. Georgeff and A. Lansky, Morgan Kaufmann, Los Altos, CA, 1987.

[92] L. V. Kale, ‘‘Comparing the Performance of Two Dynamic Load Distribution Methods,’’ *Proc. Int’l Conf. on Parallel Processing*, vol. 1, pp. 8-11, University Park, PA, Aug. 1988.

[93] W-L. Kao and R. K. Iyer, ‘‘A User-Oriented Synthetic Workload Generator,’’ *Proc. 12th Int’l. Conf. on Distributed Computing Systems*, pp. 270-277, IEEE, 1992.

[94] V. Karamcheti and B. W. Wah, *Using the ES-Kit as a Testbed for Load Balancing Experiments*, Research Report CRHC-01-26, Center for Reliable and High Performance Computing, Coordinated Science Laboratory, Univ. of Illinois, Urbana, IL 61801, July 1991.

[95] H. A. Kautz, ‘‘The Logic of Persistence,’’ *Proc. National Conf. on Artificial Intelligence*, p. 401, Morgan Kaufman, 1986.

[96] R. M. Keller, *A Survey of Research in Strategy Acquisition*, Tech. Rep. DCS-TR-115, Dept. of Computer Science, Rutgers Univ., New Brunswick, NJ, May 1982.

[97] M. Kendall and J. K. Ord, *Time Series*, 3rd ed., I Edward Arnold, London, 1990.

[98] D. L. Kiskis, M. H. Woodbury, and K. G. Shin, *Design, Implementation and Application of Synthetic Workload Generators for Real-time Systems*, Tech. Rep. CSE-TR-25-89, Computer Science and Engineering Division, Univ. of Michigan, Ann Arbor, MI, 1989.

[99] R. E. Korf, ‘‘Macro-Operators: A Weak Method for Learning,’’ *Artificial Intelligence*, vol. 26, pp. 35-77, North-Holland, 1985.

[100] T. Kunz, ‘‘The Influence of Different Workload Descriptions on a Heuristic Load Balancing Scheme,’’ *IEEE Trans. Software Engineering*, vol. 17, no. 7, July 1991.

[101] J.E. Laird, P.S. Rosenbloom, and A. Newell, ‘‘Chunking in SOAR: The Anatomy of a General Learning Mechanism,’’ *Machine Learning*, vol. 1, no. 1, pp. 11-46, Kluwer Academic Pub., Boston, MA, 1986.

[102] J. E. Laird, P. S. Rosenbloom, and A. Newell, ‘‘Soar: An Architecture for General Intelligence,’’ *Artificial Intelligence*, vol. 33, no. 1, pp. 1-64, Elsevier Science Pub., New York, 1987.

[103] K. J. Lang and G. E. Hinton, *A Time-Delay Neural Network Architecture for Speech Recognition*, CMU-CS-88-152, Dept. of Computer Science, Carnegie Mellon Univ., Pittsburgh, PA, 1988.

[104] P. Langley, ‘‘Learning to Search: From Weak Methods to Domain-Specific Heuristics,’’ *Cognitive Science*, vol. 9, pp. 217-260, Ablex Pub. Co., Norwood, NJ, 1985.

[105] M. Lebowitz, ‘‘Integrated Learning: Controlling Explanation,’’ *Cognitive Science*, pp. 219-240, Ablex Pub. Co., Norwood, NJ, 1986.

[106] K. F. Lee and S. Mahajan, ‘‘A Pattern Classification Approach to Evaluation Function Learning,’’ *Artificial Intelligence*, vol. 36, pp. 1-25, North-Holland, 1988.

[107] C. Lewis, ‘‘Composition of Productions,’’ in *Production System Models of Learning and Development*, ed. Klahr, *et al*., MIT Press, Cambridge, MA, 1987.

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

[109] J. Long, W. K. Fuchs, and J. A. Abraham, *Compiler-Assisted Static Checkpoint Insertion*, Coordinated Science Laboratory, Univ. of Illinois, Urbana, IL, (submitted) 1992.

[110] M. B. Lowrie and B. W. Wah, ‘‘Learning Heuristic Functions for Numeric Optimization Problems,’’ *Proc. Computer Software and Applications Conf.*, pp. 443-450, IEEE, Chicago, IL, Oct. 1988.

[111] P. Maes, ‘‘How to do the Right Thing,’’ *Connection Science*, vol. 1, no. 3, pp. 291-323, 1989.

[112] D. McAllester, ‘‘Truth Maintenance,’’ *Proc. 8th Natl. Conf. Artificial Intelligence*, pp. 1109-16, AAAI, Inc., Seattle, Washington, 1990.

[113] J. L McClelland and D. E. Rumelhart, *Explorations in Parallel Distributed Processing: A Handbook of Models, Programs, and Exercises*, MIT Press, Cambridge, MA, 1988.

[114] P. Mehra and B. W. Wah, ‘‘Architectures for Strategy Learning,’’ pp. 395-468 in *Computer Architectures for Artificial Intelligence Applications*, ed. B. Wah and C. Ramamoorthy, Wiley, New York, NY, 1990.

[115] P. Mehra and B. W. Wah, ‘‘Learning Load-Balancing Strategies using Artificial Neural Networks,’’ pp. 855-860 in *Intelligent Engineering Systems through Artificial Neural Networks (Proc. Int’l Conf. on Artificial Neural Networks in Engineering)*, ed. C. H. Dagli, *et al*., ASME Press, New York, 1991.

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

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

[118] P. Mehra and B. W. Wah (ed.), *Artificial Neural Networks: Concepts and Theory*, IEEE Computer Society Press, Los Alamitos, CA, 1992.

[119] R. K. Mehra, ‘‘Kalman Filters and their Applications to Forecasting,’’ *TIMS Studies in Management Sciences*, vol. 12, pp. 75-94, North-Holland, Amsterdam, 1979.

[120] B. W. Mel, ‘‘MURPHY: A Robot that Learns by Doing,’’ pp. 544-553 in *Neural Information Processing Systems*, ed. D. Z. Anderson, American Institute of Physics, New York, NY, 1988.

[121] R. S. Michalski, J. G. Carbonell, and T. M. Mitchell (ed.), *Machine Learning: An Artificial Intelligence Approach*, William Kaufmann, Los Altos, CA, 1983, 1985.

[122] I. Miller and J. E. Freund, *Probability and Statistics for Engineers (second edition)*, Prentice-Hall, Englewood Cliffs, NJ, 1977.

[123] R. B. Miller and D. W. Wichern, *Intermediate Business Statistics: Analysis of Variance, Regression, and Time Series*, Holt, Rinehart and Winston, New York, 1977.

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

[125] M. Minsky, ''Steps Toward Artificial Intelligence,'' pp. 406-450 in *Computers and Thought*, ed. E. A. Feigenbaum and J. Feldman, McGraw-Hill, New York, 1963.

[126] S. Minton, J. G. Carbonell, C. A. Knoblock, D. Kuokka, and H. Nordin, ''Improving the Effectiveness of Explanation Based Learning,'' *Proc. Workshop on Knowledge Compilation*, pp. 77-87, Computer Science Dept., Oregon State Univ., 1986.

[127] S. Minton, ''Quantitative Results Concerning the Utility of Explanation-Based Learning,'' *Artificial Intelligence*, vol. 42, pp. 363-391, Elsevier, Amsterdam, 1990.

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

[129] T. W. Mirer, *Economic Statistics and Econometrics*, Macmillan, New York, 1983.

[130] T. M. Mitchell, P. E. Utgoff, B. Nudel, and R. Benerji, ''Learning Problem-Solving Heuristics Through Practice,'' *Proc. 7th Int'l Joint Conf. on Artificial Intelligence*, pp. 127-134, William Kaufman, Los Altos, CA, 1981.

[131] T. M. Mitchell, ''Learning and Problem Solving,'' *Proc. 8th Int'l Joint Conf. on Artificial Intelligence*, pp. 1139-1151, William Kaufman, Los Altos, CA, Aug. 1983.

[132] T. M. Mitchell, ''Toward Combining Empirical and Analytical Methods for Inferring,'' pp. 81-103 in *Artificial and Human Intelligence*, ed. Banerji and Elithorn, Elsevier, New York, 1984.

[133] T.M. Mitchell, R.M. Keller, and S.T. Kedar-Cabelli, ''Explanation-Based Generalization: A Unifying View,'' *Machine Learning*, vol. 1, no. 1, pp. 47-80, Kluwer Academic Pub., Boston, MA, 1986.

[134] J. S. Morgan, E. C. Patterson, and A. H. Klopf, ''Drive-Reinforcement Learning: A Self-Supervised Model for Adaptive Control,'' *Network: Computation in Neural Systems*, vol. 1, pp. 439-448, IOP Pub. Ltd., UK, 1990.

[135] L. Morgenstern and L. A. Stein, ''Why Things Go Wrong: A Formal Theory of Causal Reasoning,'' *Proc. National Conf. on Artificial Intelligence AAAI-88*, pp. 518-523, 1988.

[136] D. J. Mostow, ''Machine Transformation of Advice into a Heuristic Search Procedure,'' pp. 367-404 in *Machine Learning: An Artificial Intelligence Approach*, ed. R. S. Michalski, *et al*., Morgan Kaufmann, Los Altos, CA, 1983.

[137] P. Munro, ''A Dual Back-Propagation Scheme for Scalar Reward Learning,'' *Proc. Ninth Annual Conf. of the Cognitive Science Society*, pp. 165-176, Lawrence Erlbaum Associates, Hillsdale, NJ, 1987.

[138] B. A. Nadel, ''Constraint Saisfaction Algorithms,'' *Computational Intelligence*, vol. 5, pp. 188-224, National Research Council Canada, Toronto, 1989.

[139] A. K. Nanda, H. Shing, T-H. Tzen, and L. M. Ni, ‘‘A Replicated Workload Framework to Study Performance Degradation in Shared-Memory Multiprocessors,’’ *Proc. Int’l. Conf. Parallel Processing*, vol. I, pp. 161-168, IEEE, 1990.

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

[141] R. Neches, ‘‘Learning through Incremental Refinement of Procedures,’’ in *Production System Models of Learning and Development*, ed. Klahr, *et al*., MIT Press, Cambridge, MA, 1987.

[142] D. M. Neves, ‘‘Learning Procedures from Examples and by Doing,’’ *Proc. Int’l Joint Conf. on Artificial Intelligence*, pp. 624-30, Morgan Kaufman, 1983.

[143] D. Nguyen and B. Widrow, ‘‘The Truck Backer-Upper: An Example of Self-Learning in Neural Networks,’’ *Proc. Int’l Joint Conf. on Neural Networks*, vol. II, pp. 357-363, IEEE, 1989.

[144] N. J. Nilsson, *Principles of Artificial Intelligence*, Tioga, 1980.

[145] P. S. Ow, S. F. Smith, and A. Thiriez, ‘‘Reactive Plan Revision,’’ *Proc. Tenth National Conf. on Artificial Intelligence AAAI-88*, vol. 1, pp. 77-82, Saint Paul, MN, 1988.

[146] J. Pearl, *Heuristics--Intelligent Search Strategies for Computer Problem Solving*, Addison-Wesley, Reading, MA, 1984.

[147] B.W. Porter and D.F. Kibler, ‘‘Experimental Goal Regression: A Method for Learning Problem-solving Heuristics,’’ *Machine Learning*, vol. 1, no. 3, pp. 249-286, Kluwer Academic Pub., Boston, MA, 1986.

[148] L. Rendell, ‘‘A General Framework for Induction and a Study of Selective Induction,’’ *Machine Learning*, vol. 1, no. 2, pp. 177-226, Kluwer Academic Pub., Boston, MA, 1986.

[149] L. A. Rendell, *An Adaptive Plan for State-Space Problems*, Tech. Rep. CS-81-13, Univ. of Waterloo, Ontario, Canada, March 1981.

[150] L. A. Rendell, ‘‘A New Basis for State-Space Learning Systems and a Successful Implementation,’’ *Artificial Intelligence*, vol. 20, pp. 369-392, North-Holland, 1983.

[151] E. Rich and K. Knight, *Artificial Intelligence*, McGraw Hill, New York, 1991.

[152] P. Rosenbloom and A. Newell, ‘‘Learning by Chunking: A Production System Model of Practice,’’ in *Production System Models of Learning and Development*, ed. Klahr, *et al*., MIT Press, Cambridge, MA, 1987.

[153] S. Ross, *Introduction to Stochastic Dynamic Programming*, Academic Press, New York, 1983.

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

[155] S. Russell and E. Wefald, ‘‘Principles of Metareasoning,’’ *Artificial Intelligence*, vol. 49, pp. 361-395, Elsevier, Amsterdam, 1991.

[156] E.D. Sacerdoti, ‘‘The Nonlinear Nature of Plans,’’ pp. 206-214 in *Reasoning about Actions and Plans*, ed. Georgeff and Lansky, 1987.

[157] A. L. Samuel, ‘‘Some Studies in Machine Learning Using the Game of Checkers,’’ *IBM J. Research and Development*, vol. 3, pp. 210-229, IBM, 1959.

[158] A. L. Samuel, ‘‘Some Studies in Machine Learing Using the Game of Checkers II--Recent Progress,’’ *J. of Research and Development*, vol. 11, no. 6, pp. 601-617, IBM, 1967.

[159] SAS Institute, Inc., *SAS/ETS User’s Guide, Version 5 Edition*, Cary, NC, 1984.

[160] J. H. Schmidhuber, *Making the World Differentiable,*, Tech. Rep. FKI-126-90, Technical Univ. of Munich, Munich, Germany, 1990.

[161] P. Schooley, ‘‘Learning State Evaluation Functions,’’ *Machine Learning*, pp. 177-179, Kluwer Academic Pub., 1985.

[162] M. J. Schoppers, *Representation and Automatic Synthesis of Reaction Plans*, Ph.D. Thesis, Dept. of Computer Science, Univ. of Illinois, Urbana, IL., 1988.

[163] S. R. Schwartz, *Resource Constrained Parameter Tuning Applied to Stereo Vision*, M.Sc. Thesis, Dept. of Electrical and Computer Engineering, Univ. of Illinois, Urbana, IL, Aug. 1991.

[164] S. R. Schwartz and B. W. Wah, ‘‘Automated Parameter Tuning in Stereo Vision Under Time Constraints,’’ *Proc. Int’l Conf. on Tools for Artificial Intelligence*, IEEE, (accepted to appear) Nov. 1992.

[165] H. Simon, *The Sciences of the Artificial*, The MIT Press, Cambridge, MA, 1982.

[166] R. G. Smith, T. M. Mitchell, R. A. Chestek, and B. G. Buchanan, ‘‘A Model for Learning Systems,’’ *Proc. 5th Int’l Joint Conf. on Artificial Intelligence*, pp. 338-343, William Kaufman, Los Altos, CA, Aug. 1977.

[167] S. F. Smith, ‘‘Flexible Learning of Problem Solving Heuristics through Adaptive Search,’’ *Proc. Int’l Joint Conf. on Artificial Intelligence*, pp. 422-5, Morgan Kaufman, 1983.

[168] K. Sreenivasan and A. J. Kleinman, ‘‘On the Construction of a Representative Synthetic Workload,’’ *Communications of the ACM*, vol. 17, no. 3, pp. 127-132, ACM, 1974.

[169] N. S. Sridharan and J. L. Bresina, *Exploration of Problem Reformulation and Strategy Acquisition: A Proposal*, Tech. Rep. LCSR-TR-53, Lab. Computer Science Research, Rutgers Univ., New Brunswick, NJ, March 1984.

[170] W. Stallings, *Data and Computer Communications*, Macmillan Pub. Co., New York, 1988.

[171] J. A. Stankovic, ‘‘Stability and Distributed Scheduling Algorithms,’’ *Trans. on Software Engineering*, vol. SE-11, pp. 1141-1152, IEEE, Oct. 1985.

[172] D.M. Steier, J.E. Laird, A. Newell, and P.S. Rosenbloom, ‘‘Varieties of Learning in SOAR: 1987,’’ *Machine Learning*, pp. 300-311, Kluwer Academic Pub., Boston, MA, 1987.

[173] R. S. Sutton, *Temporal Credit Assignment in Reinforcement Learning*, Ph.D. Thesis, Univ. of Massachusetts, Amherst, MA, Feb. 1984.

[174] R. S. Sutton and B. Pinette, ‘‘The Learning of World Models by Connectionist Networks,’’ *Proc. Seventh Ann. Conf. Cognitive Science Soc.*, pp. 54-64, Lawrence Erlbaum Associates, Hillsdale, NJ, 1985.

[175] R. S. Sutton, ‘‘Learning to Predict by the Methods of Temporal Differences,’’ *Machine Learning*, vol. 3, pp. 9-44, Kluwer Academic Pub., Aug. 1988.

[176] R. S. Sutton, ‘‘Convergence Theory for a New Kind of Prediction Learning,’’ *Proc. 1988 Workshop on Computational Learning Theory*, pp. 421-422, Morgan Kaufmann, Palo Alto, CA, 1988.

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

[178] G. Tesauro, ‘‘Connectionist Learning of Expert Backgammon Evaluations,’’ *Machine Learning*, pp. 200-206, Kluwer Academic Pub., Boston, MA, 1988.

[179] G. Tesauro and T. J. Sejnowski, ‘‘A Parallel Network that Learns to Play Backgammon,’’ *Artificial Intelligence*, vol. 39, pp. 357-390, Elsevier Science Pub., New York, 1989.

[180] G. Tesauro, ‘‘Neurogammon Wins Computer Olympiad,’’ *Neural Computation*, vol. 1, no. 3, pp. 321-323, MIT Press, Cambridge, MA, 1989.

[181] G. Tesauro, ‘‘Practical Issues in Temporal Difference Learning,’’ *Machine Learning*, vol. 8, no. 3/4 (Special Issue on Reinforcement Learning), pp. 257-278, Kluwer Academic Publishers, Boston, MA, 1991.

[182] K. H. Thearling, *Techniques for Automatic Test Knowledge Extraction from Compiled Circuits*, Ph.D. thesis, Dept. of Electrical Engineering, Univ. of Illinois, Urbana, IL, 1990.

[183] Y. L. Tong and D. E. Wetzell, ‘‘Allocation of Observations for Selecting the Best Normal Population,’’ pp. 213-224 in *Design of Experiments: Ranking and Selection*, ed. T. J. Santner and A. C. Tamhane, Marcel Dekker, New York, NY, 1984.

[184] B. W. Wah and H. Kriplani, ‘‘Resource Constrained Design of Artificial Neural Networks,’’ *Proc. Int’l Joint Conf. on Neural Networks*, vol. III, pp. 269-279, IEEE, June 1990.

[185] B. W. Wah, ‘‘Population-Based Learning: A New Method for Learning from Examples under Resource Constraints,’’ *Trans. on Knowledge and Data Engineering*, vol. 4, no. 5, pp. 454-474, IEEE, Oct. 1992.

[186] B. W. Wah, A. Aizawa, and A. Ieumwananonthachai, ‘‘Real-Time Learning of Heuristics,’’ *Trans. on Knowledge and Data Engineering*, IEEE, (accepted to appear) Dec. 1992.

[187] B. Walker, G. Popek, R. English, C. Kline, and G. Thiel, ‘‘The LOCUS Distributed Operating System,’’ *Proc. Nineth Symp. on Operating System Principles*, pp. 49-70, ACM, 1983.

[188] G. R. Walsh, *Methods of Optimization*, Wiley, London, England, 1977.

[189] C. J. C. H. Watkins, *Learning from Delayed Rewards*, Ph.D. Thesis, King’s College, London, May 1989.

[190] E.H. Wefald and S.J. Russell, ‘‘Adaptive Learning of Decision-Theoretic Search Control Knowledge,’’ *Machine Learning*, pp. 408-411, Kluwer Academic Pub., Boston, MA, 1989.

[191] W. W. S. Wei, *Time Series Analysis*, Addison-Wesley, Redwood City, CA, 1990.

[192] A. S. Weigend, B. A. Huberman, and D. E. Rumelhart, ‘‘Predicting the Future: A Connectionist Approach,’’ *Int’l. J. of Neural Systems*, vol. 1, no. 3, p. 209, World Scientific Pub., New Jersey, 1990.

[193] P. J. Werbos, ‘‘Consistency of HDP Applied to a Simple Reinforcement Learning Problem,’’ *Neural Networks*, vol. 3, pp. 179-189, Pergamon Press, Elmsford, NY, 1990.

[194] P. J. Werbos, ‘‘A Menu of Designs for Reinforcement Learning over Time,’’ pp. 67-96 in *Neural Networks for Control*, ed. Miller et al., MIT Press, Cambridge, MA, 1990.

[195] D. White, *Dynamic Programming*, Oliver and Boyd, Edinburgh, UK, 1969.

[196] S.D. Whitehead and D.H. Ballard, ‘‘A Role for Anticipation in Reactive Systems that Learn,’’ *Proc. 6th Int’l. Workshop on Machine Learning*, pp. 354-357, Morgan Kaufmann, San Mateo, CA, 1989.

[197] B. Widrow, N. K. Gupta, and S. Maitra, ‘‘Punish/Reward: Learning with a Critic in Adaptive Threshold Systems,’’ *Trans. Systems, Man, and Cybernetics*, vol. SMC-3, no. 5, pp. 455-465, IEEE, 1973.

[198] B. Widrow and S. D. Stearns, *Adaptive Signal Processing*, Prentice-Hall, Englewood Cliffs, NJ, 1985.

[199] S. W. Wilson, ‘‘Hierarchical Credit Allocation in Classifier Systems,’’ in *Genetic Algorithms and Simulated Annealing*, ed. L. Davis, Pitman, London, 1987.

[200] J. C. Yan and S. F. Lundstrom, ‘‘The Post-Game Analysis Framework--Developing Resource Management Strategies for Concurrent Systems,’’ *Trans. on Knowledge and Data Engineering*, vol. 1, no. 3, pp. 293-309, IEEE, Sept. 1989.

[201] C. F. Yu, *Efficient Combinatorial Search Algorithms*, Ph.D. Thesis, School of Electrical Engineering, Purdue Univ., West Lafayette, IN, Dec. 1986.

[202] C. F. Yu and B. W. Wah, ‘‘Learning Dominance Relations in Combinatorial Search Problems,’’ *Trans. on Software Engineering*, vol. SE-14, no. 8, pp. 1155-1175, IEEE, Aug. 1988.

[203] S. Zhou, ‘‘An Experimental Assessment of Resource Queue Lengths as Load Indices,’’ *Proc. Winter USENIX Conf.*, pp. 73-82, 1987.

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

[205] S. Zhou, X. Zheng, Z. Jingwen, and P. Delisle, *UTOPIA: A Load Sharing Facility for Large, Heterogeneous Distributed Computer Systems*, Tech. report CSRI-257, Computer Systems Research Institute, University of Toronto, April 1992.

VITA

Pankaj Mehra received his B.Tech. degree in Computer Science and Engineering from the Indian Institute of Technology, Delhi, in 1986. He is currently a candidate for the Ph.D. degree in Computer Science at the University of Illinois at Urbana-Champaign. After completing his doctoral dissertation, he will take up a position as a Computer Scientist with Recom Technologies, Inc., working on the Parallel Systems Diagnosis and Visualization Project within the Computational Research Systems Branch at NASA-Ames Research Center.

Mr. Mehra is the co-editor of *Artificial Neural Networks: Concepts and Theory*, a tutorial published by the IEEE Computer Society Press in 1992. He has presented papers at various conferences on distributed computing, artificial intelligence, machine learning, and neural networks. His research interests include computer-workload characterization, computational learning theory, and intelligent resource-scheduling algorithms.

AUTOMATED LEARNING OF LOAD-BALANCING STRATEGIES FOR A DISTRIBUTED COMPUTER SYSTEM

Pankaj Mehra, Ph.D.
Department of Computer Science
University of Illinois at Urbana-Champaign, 1993
B. W. Wah, Advisor

Workstations interconnected by a local-area network are the most common examples of distributed systems. The performance of such systems can be improved via load balancing, which migrates tasks from the heavily loaded sites to the lightly loaded ones. Load-balancing strategies have two components: load indices and migration policies. This thesis presents SMALL (Systematic Method for Automated Learning of Load-balancing strategies), a system that learns new load indices and tunes the parameters of given migration policies. The key component of SMALL is DWG, a *dynamic workload generator* that allows off-line measurement of task-completion times under a wide variety of precisely controlled loading conditions. The data collected using DWG are used for training *comparator neural networks*, a novel architecture for learning to compare functions of time series. After training, the outputs of these networks can be used as load indices. Finally, the load-index traces generated by the comparator networks are used for tuning the parameters of given load-balancing policies. In this final phase, SMALL interfaces with the TEACHER system of Wah, *et al.* in order to search the space of possible parameters using a combination of point-based and population-based approaches. Together, the components of SMALL constitute an *automated strategy-learning system* for performance-driven improvement of existing load-balancing software.

AUTOMATED LEARNING OF LOAD-BALANCING STRATEGIES FOR A DISTRIBUTED COMPUTER SYSTEM

Pankaj Mehra, Ph.D.
Department of Computer Science
University of Illinois at Urbana-Champaign, 1993
B. W. Wah, Advisor

Workstations interconnected by a local-area network are the most common examples of distributed systems. The performance of such systems can be improved via load balancing, which migrates tasks from the heavily loaded sites to the lightly loaded ones. Load-balancing strategies have two components: load indices and migration policies. This thesis presents SMALL (Systematic Method for Automated Learning of Load-balancing strategies), a system that learns new load indices and tunes the parameters of given migration policies. The key component of SMALL is DWG, a *dynamic workload generator* that allows off-line measurement of task-completion times under a wide variety of precisely controlled loading conditions. The data collected using DWG are used for training *comparator neural networks*, a novel architecture for learning to compare functions of time series. After training, the outputs of these networks can be used as load indices. Finally, the load-index traces generated by the comparator networks are used for tuning the parameters of given load-balancing policies. In this final phase, SMALL interfaces with the TEACHER system of Wah, *et al.* in order to search the space of possible parameters using a combination of point-based and population-based approaches. Together, the components of SMALL constitute an *automated strategy-learning system* for performance-driven improvement of existing load-balancing software.

To the people of India

ACKNOWLEDGEMENTS

I would like to thank my advisor, Professor Benjamin Wah, for his advice and support. If this thesis amounts to something, it is probably so because of his perfectionism and attention for detail. I would also like to thank current and former members of my committee — Professors Andrew Barron, Geneva Belford, Ravi Iyer, Kwei-Jay Lin, Tony Ng, and Larry Rendell — for their comments and suggestions on my work. Further thanks are due to Drs. Jerry Yan and Henry Lum for helpful discussions that shaped my research during the summers of 1987 through 1990, when I was visiting NASA-Ames Research Center.

I would also like to thank my current and former colleagues — Albert Yu, Kate Baumgartner, Mark Gooley, Lon-Chan Chu, Kumar Ganapathy, Arthur Ieumwananonthachai, Vijay Karamcheti, Tanay Karnik, Akiko Aizawa, and Chin-Chi Teng — for helpful discussions about ideas presented in this thesis. I have gained valuable insights about machine learning and neural networks from discussions with several friends, colleagues, and faculty members at the University of Illinois: Subutai Ahmad, Bob Horst, Darrell Hougen, Gerry Tesauro, Raj Seshu, Chris Matheus, Larry Rendell, P. R. Kumar, Kevin Buescher, Bob Stepp, Steve Chien, Steve Omohundro, Ray Mooney, and Ryszard Michalski. I wish to especially thank Rich Sutton for indulging me in long-distance discussion through e-mail.

The workload-generation experiments reported in this thesis were conducted on SUN 3/50, SUN3/60, and SUN 3/260 workstations running the SunOS operating system. Comparator networks were simulated using the Rochester Connectionist Simulator. The contour plots of Chapter III were drawn using Mathematica 2.1. The TEACHER software used for tuning policy parameters was written by Arthur Ieumwananonthachai, based on earlier versions by Steve Schwartz. Tanay Karnik provided initial versions of the programs for linear and exponential extrapolation, which are used in Chapter IV for extracting trends from filtered load patterns. The filters were designed using Mathematica 2.1. I also used the UNIXSTAT programs, written by Gary Perlman, for computing statistics and analyzing data.

To the many friends who helped me cope with the pressures and frustrations of graduate school — Munindar Singh, Sanjay and Vinita Bhansali, Harish Kriplani, Abhay and Tanuja, Kanth and Shaku, Samir Mathur, S. Keshav, Inderpal Mumick, Milt Epstein, Ram Charan and family, George and Caroline Badger, Uncle Myron and Aunt Emmie — thanks for your encouragement and support! I wish to thank Donna Guzy not only for her help with paperwork and travel arrangements but also for her delicious cakes and pies.

Among my family members, I first wish to thank my mother and father, whose love and sacrifices have brought me this far. I want to thank my sisters Suman and Juhi, whose loving care once saved my life, and my wife Ranjana, who has suffered with me through the worst of it. Other friends and relatives — Pratap and Prabha Mehra, Professor J. N. Kapur, and Y. A. Shetty — have motivated, supported and encouraged me through the past six years.

I wish to acknowledge the support of National Aeronautics and Space Administration, under Contract NCC-2-481, the National Science Foundation, under grant MIP 88-10584, and Sumitomo Electric Industries, Yokohama, Japan. Finally, I wish to thank the people of India, whose unselfish support of academic excellence and scholarship paved my way to a higher education. It is to them that I dedicate this thesis.

TABLE OF CONTENTS

Chapter Page

I. LOAD BALANCING AS A STRATEGY-LEARNING TASK 1
- A. Load Balancing in Distributed Computer Systems 1
- B. Scope of this Thesis 5
- C. State of the Art 7
- D. Automated Learning of Load-Balancing Strategies 11
- E. Contributions of this Thesis 19
- F. Thesis Outline 21

II. SMALL: A SYSTEM FOR LEARNING LOAD-BALANCING STRATEGIES 22
- A. Architecture of SMALL 22
- B. Operation of SMALL 24
- C. Summary 34

III. DWG: A TOOL FOR SYNTHETIC WORKLOAD GENERATION 37
- A. Design Goals 37
- B. Previous Work 39
- C. Architecture of DWG: A Dynamic Workload Generator 42
- D. Operation of DWG 47
- E. Evaluation, Parameterization and Tuning of Generation Mechanisms 52
- F. Chapter Summary 63

IV. COMPARATOR NEURAL NETWORKS FOR LEARNING LOAD INDICES 66
- A. Design Goals 66
- B. The Comparator Network: Architecture and Learning Algorithms 73
- C. Data Preprocessing 77
- D. Empirical Results 81
- E. Evaluation of Load Indices Learned using Comparator Networks 84
- F. Chapter Summary 93

V. STRATEGY LEARNING: A SURVEY OF ISSUES AND APPROACHES 95
- A. Introduction 95
- B. Structure of strategy-learning tasks 99
- C. Issues in strategy learning 120
- D. Techniques for strategy learning 132
- E. Architecture of strategy-learning systems 152
- F. Practicality of strategy learning 159
- G. Chapter summary 166

VI. POPULATION-BASED LEARNING OF LOAD-BALANCING POLICIES 168
A. The policy-learning problem 168
B. Brief overview of TEACHER 171
C. Data Preparation 175
D. Empirical Results 179
E. Chapter Summary 184

VII. CONCLUSIONS 185

REFERENCES 187

VITA 199

LIST OF TABLES

Table	Page
I. Typical performance-indicating variables available in UNIX-like operating systems	8
II. Times required by different phases of SMALL's operation	35
III. Benchmark programs used in evaluation and tuning of DWG	54
IV. Parameter sets for doctoring rules	56
V. Signed errors due to generation for different parameter sets	58
VI. Unsigned errors due to generation for different parameter sets	59
VII. Significance test for coefficient of rank-correlation (I)	85
VIII. Significance test for coefficient of rank-correlation (II)	86
IX. Components of strategy-learning tasks	98
X. Examples of strategy-learning tasks	98
XI. Components of a performance task	103
XII. Examples of performance tasks in strategy learning	111
XIII. Examples of problem solvers in strategy-learning tasks	115
XIV. Examples of environments for strategy learning	119
XV. Issues, tasks, and approaches of strategy learning	121
XVI. Examples of issues in strategy learning	133
XVII. Architectural models characterized by their approach	159
XVIII. Examples of strategy-learning tasks and architectures	160
XIX. Examples of practical issues in strategy learning	161
XX. Performance of policies learned under different assumptions	179

LIST OF FIGURES

Figure	Page
1. Our model of distributed systems.	2
2. Classification of load-balancing strategies	3
3. The load-balancing policy considered in this thesis	6
4. Comparison between experimentation environments	17
5. Overview of SMALL	23
6. The five phases of SMALL's operation	25
7. Operation of SMALL: Collection of background-load patterns	27
8. Operation of SMALL: Measurement of completion times	28
9. Operation of SMALL: Load-index learning	29
10. Operation of SMALL: Policy learning	32
11. Operation of SMALL: Application phase	34
12. Workload generation without feedback	40
13. Workload generation with feedback using real workloads	40
14. Workload generation with feedback using synthetic processes	41
15. Workload generation with feedback using dynamic doctoring	42
16. Architectural description of DWG	43
17. Format of DWG log files	46
18. The need for *dynamic doctoring* of generated load	51
19. Measuring the accuracy of generated loads	53
20. Contour plots of cumulative probability of error with nine different parameter sets	61
21. Contour plot of cumulative probability of error for the selected parameter set (7)	62
22. Comparison of true and generated resource-utilization patterns (Expt. #17)	64
23. Comparison of true and generated resource-utilization patterns (Expt. #15)	65
24. The objective function for load-index learning	72
25. A typical episode in training a comparator neural network	73
26. Schematic of a feed-forward neural network	75
27. Effects of lowpass filters of order 7 with different cutoff frequencies	79
28. Trend extraction via linear and exponential fitting	80
29. Learning curves for within-site comparisons	83
30. Learning curves for across-site comparisons	83
31. Performance of all test-jobs under load #23; policies: opt, min	88
32. Performance of all test-jobs under load #23; policies: opt, min, rnd, and max	89
33. Performance of all test-jobs under load #11; policies: opt, min, rnd, and max	90
34. Contour plot of cumulative probability of speed-up; policy: min; no overheads or delays	91
35. Raw resource-utilization patterns and their load index	93
36. Classification of strategy-learning tasks	98
37. Typical information flow in a strategy-learning system	100
38. Attributes for classifying performance tasks	103

39. Components of problem solvers and their properties 111
40. Components of a learning environment 115
41. Dietterich and Buchanan's model of learning systems 153
42. Minsky's model of learning systems 153
43. The hybrid point-based model of learning systems 153
44. The TEACHER model of learning systems 158
45. Representation of heuristic methods 168
46. The architecture of TEACHER 171
47. An example of mutation applied to an HM 173
48. An illustration of the crossover operation 173
49. Rules for point-based learning 175
50. Overheads and delays in load balancing 177
51. Contour plot of cumulative probability of speed-up; M=100,I=100,P=20 181
52. Contour plot of cumulative probability of speed-up; M=5000,I=100,P=20 182
53. Contour plot of cumulative probability of speed-up; M=100,I=1000,P=20 183