POPULATION-BASED LEARNING OF LOAD BALANCING POLICIES FOR A DISTRIBUTED COMPUTER SYSTEM

Pankaj Mehra
Information Sciences, MS 269-3
NASA - Ames Research Center
Moffett Field, CA 94035-1000
mehra@ptolemy.arc.nasa.gov

Benjamin W. Wah
Coordinated Science Laboratory
University of Illinois, Urbana-Champaign
Urbana, IL 61801
wah@manip.crhc.uiuc.edu

ABSTRACT

Effective load-balancing policies use dynamic resource information to schedule tasks in a distributed computer system. In this paper, we present a novel method for automatically learning such policies. At each site in our system, we use a comparator neural network to predict the relative speedup of an incoming task using only the resource-utilization patterns obtained prior to the task's arrival. Outputs of these comparator networks are broadcast periodically over the distributed system, and the resource schedulers at each site use these values to determine the best site for executing an incoming task. The delays incurred in propagating workload information and tasks from one site to another, as well as the dynamic and unpredictable nature of workloads in multiprogrammed multiprocessors, may cause the workload pattern at the time of execution to differ from patterns prevailing at the times of load-index computation and decision making. Our load-balancing policy accommodates this uncertainty by using certain tunable parameters. We present a population-based machine-learning algorithm that adjusts these parameters in order to achieve high average speedups with respect to local execution. Our learning algorithm overcomes the lack of task-specific information by learning to compare the relative speedups of different sites with respect to the same site, rather than attempting to predict absolute speedups. We present experimental results based on the execution of benchmark programs on a network of Sun workstations connected by a local area network. Our results show that our load-balancing policy, when combined with the comparator neural network for workload characterization, is effective in exploiting idle resources in a distributed computer system.

1. INTRODUCTION

Workload indices and policy parameters are key components of load-balancing policies, which seek to improve the performance of distributed computer systems by exploiting idle resources. In this paper, we present an automated policy-learning system that integrates automated learning of workload indices and automated tuning of policy parameters.

Research supported by National Aeronautics and Space Administration Grants NCC 2-481 and NAG 1-613, and a grant from Sumitomo Electric Industries, Yokohama, Japan.

Proceedings of Computing in Aerospace 9 Conference, AIAA, October 19-21, 1993, San Diego, California.

Our distributed system comprises a network of interconnected sites, each having private resources (such as CPU and memory) and possibly disks; all sites share the network resource. We assume that the sites are homogeneous in the sense that programs compiled at one site in the system can be executed at any other. We further assume that resources such as network and disks can be shared transparently among the processes at different sites; other resources, such as CPU and memory, can only be accessed by local processes. We do allow for *configurational heterogeneity*, *i.e.*, sites having different processor speeds, memory space, or disk space. As a result, the load-balancing policy at each site must be tuned with respect to its local configuration information as well as delays that affect the migration and execution of tasks at remote sites.

We assume that our load-balancing policy has no control on the background workload at each site, which may be caused either by operating-system functions or by tasks that, for some reason, cannot be migrated. Our load-balancing policy does not use information on the tasks to be scheduled since such information is usually unavailable before the tasks are executed. Local execution of an incoming task is the reference case, relative to which all load-balancing decisions are evaluated. Since we assume independent tasks, our objective is to maximize the average speed-up, computed on a task-by-task basis, over local execution. We further assume that our load-balancing policies are dynamic, preemptive, decentralized, given the lack of task information and the dynamic and unpredictable nature of resource usage.

Figure 1 shows the parameterized policy considered in this paper. The sender-side rules (SSRs) are evaluated at s, the site of arrival of a task. *Reference* can be either *0* or *MinLoad*; the other parameters — δ, θ_1, and θ_2 — take non-negative real values. A remote destination, r, is picked randomly from *Destinations,* a set of sites whose load indices fall within a small

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

Figure 1. The load-balancing policy studied

neighborhood of *Reference*. If *Destinations* is the empty set, or if the last SSR fails, then the task is executed locally at *s*; otherwise, site *r* is requested to receive the task. Upon receiving that request, site *r* applies its receiver-side rule (RSR). If the RSR succeeds, the task is migrated; otherwise, the task is executed locally at *s*.

Our policy is a generalization of several well-known decentralized dynamic load-balancing policies [2, 7, 15]. It uses primitive workload measures (*Load*, denoting the load at each site) as well as an abstract measure (*MinLoad*, denoting the smallest *Load* value). Our focus in this paper is on methods for tuning the parameters of the policy at each site.

The design of effective load-balancing policies involves three components: measurement and regeneration of low-level resource-utilization information for carrying load-balancing experiments, the design of workload measures that indicate each site's load, and the design of load-balancing policies that determine both the conditions under which tasks are migrated and the destinations of incoming tasks. In this paper, we summarize the design of the first two components [8, 9] and present a population-based learning algorithm for tuning policy parameters. We describe the architecture of SMALL (*S*ystematic *M*ethod for *A*utomated *L*earning of *L*oad-balancing policies), our learning system, in Section 2. Section 3 presents the various phases of SMALL's operation: data collection, load-index learning, policy learning, and application. Our learning system is described in Section 4, and experimental results are shown in Sections 5 and 6.

2. ARCHITECTURE OF SMALL

Figure 2 shows our learning system. The three key components of SMALL include a workload generator (DWG), a trainable load-index function (comparator network), and a mechanism for tuning the parameters of given load-balancing policies (TEACHER). DWG resides partly inside the operating-system kernel and partly at the process level, and forms a layer below the given load-balancing system. The learning system 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.

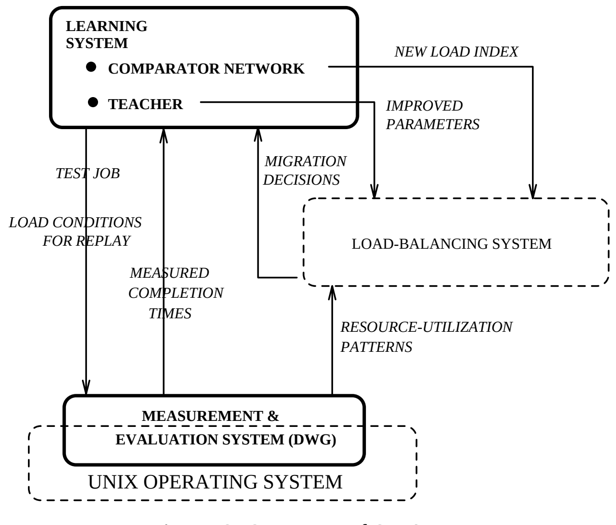


Figure 2. Overview of SMALL

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 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 set of site-specific policies.

The load-balancing system implements the policy shown in Figure 1, 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 [1], such functions are provided by *shells*. We also assume that the underlying operating system provides support for initiating processes and for detecting their termination.

3. 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 these workloads. The learning phase can be subdivided into the learning of indices and the tuning of policy parameters. Figure 3 shows a schematic view of the overall operation of SMALL.

Our method is based on *off-line learning*, in which data collected ahead of time are used for modifying a load-balancing policy 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.

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

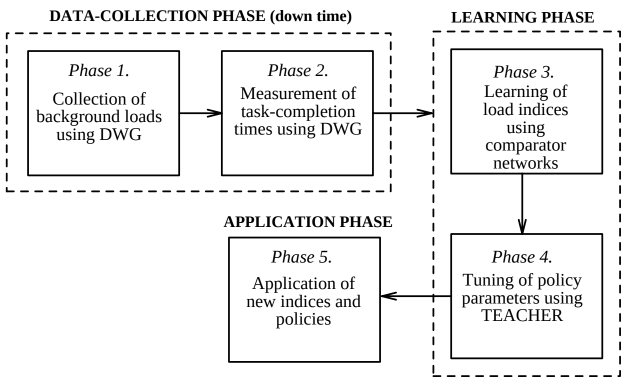

DATA-COLLECTION PHASE (down time)
LEARNING PHASE
Phase 1. Collection of background loads using DWG
Phase 2. Measurement of task-completion times using DWG
Phase 3. Learning of load indices using comparator networks
APPLICATION PHASE
Phase 5. Application of new indices and policies
Phase 4. Tuning of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

Figure 3. The five phases of SMALL's operation

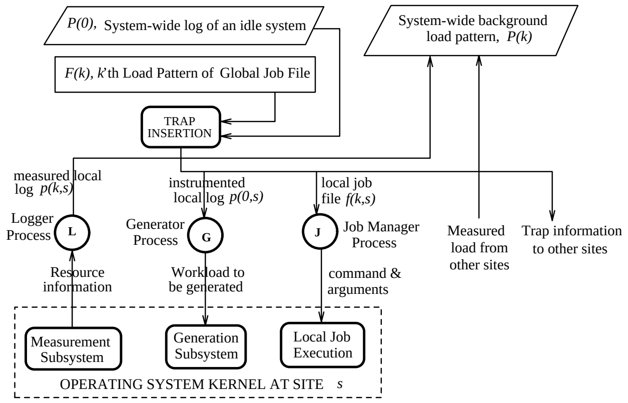

P(0), System-wide log of an idle system
System-wide background load pattern, P(k)
F(k), k'th Load Pattern of Global Job File
TRAP INSERTION
measured local log p(k,s)
instrumented local log p(0,s)
local job file f(k,s)
Logger Process
Generator Process
Job Manager Process
Measured load from other sites
Trap information to other sites
Resource information
Workload to be generated
command & arguments
Measurement Subsystem
Generation Subsystem
Local Job Execution
OPERATING SYSTEM KERNEL AT SITE s

Figure 4. Phase 1: Collection of background-load patterns

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 policies 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 summarizes the various phases of SMALL's operation.

3.1. *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 4 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).

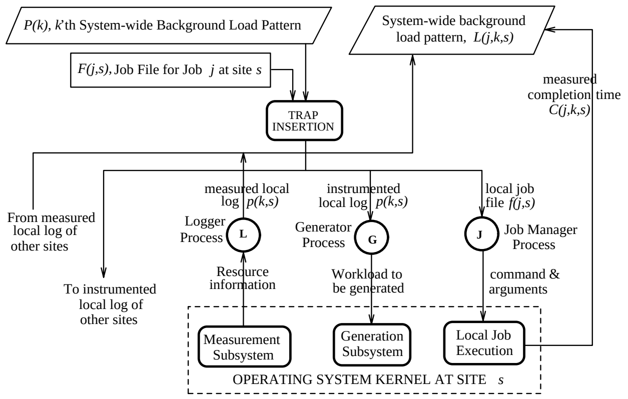

P(k), k'th System-wide Background Load Pattern
System-wide background load pattern, L(j,k,s)
F(j,s), Job File for Job j at site s
TRAP INSERTION
measured completion time C(j,k,s)
From measured local log of other sites
measured local log p(k,s)
instrumented local log p(k,s)
local job file f(j,s)
Logger Process
Generator Process
Job Manager Process
To instrumented local log of other sites
Resource information
Workload to be generated
command & arguments
Measurement Subsystem
Generation Subsystem
Local Job Execution
OPERATING SYSTEM KERNEL AT SITE s

Figure 5. Phase 2: Measurement of completion times

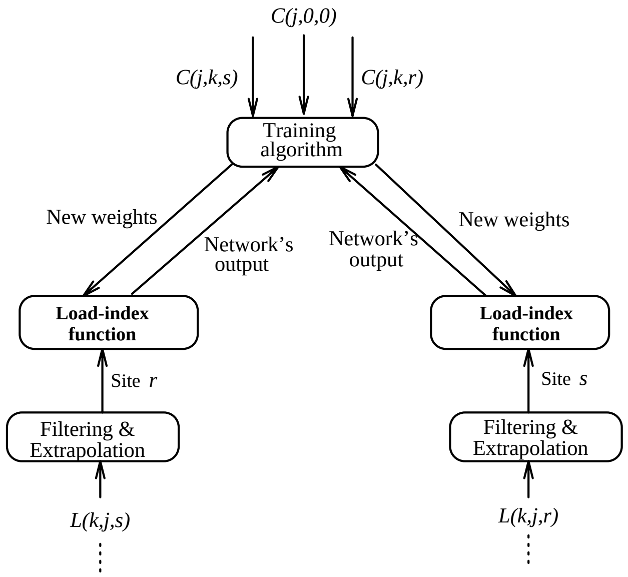


Figure 6. Phase 3: Load-index learning

Off line, this log is instrumented by certain trap-insertion routines. 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 s is labelled $f(k,s)$), which are then given to the job-manager processes (labelled J) at each site. Likewise, local instrumented logs (the log for site s is labelled $p(0,s)$) 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 $p(k,s)$ for site s) are 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 5) 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 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 $L(k,j,s)$ in Figure 5, 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 5, of job j when it is executed at site s under load pattern k.

3.2. *Off-line Learning of Load Indices*

Figure 6 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. We solve a ''learning-to-compare'' problem rather than a ''learning-to-predict'' problem because we cannot predict the completion time of an incoming task, and we are only interested to find a site that allows the task to be completed sooner.

Information flows for a typical learning trial are shown in Figure 6. Each trial involves a randomly selected job (j), a randomly selected load pattern (k), and a randomly selected pair of sites (s and r). First, the raw utilization patterns recorded earlier by DWG are smoothed using low-pass filters. Smoothing helps in detection of trend; the smaller the cutoff frequency of the low-pass 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 $C(j,k,s)$ and $C(j,k,r)$, 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, $\dfrac{C(j,k,s) - C(j,k,r)}{C(j,0,0)}$, where $C(j,0,0)$ is the completion time of job j on a chosen idle file server. Our configuration, shown in Figure 6, simultaneously trains multiple comparable functions; we refer to this as *the comparator configuration*.

Details of the experiments can be found elsewhere [9]. 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.

3.3. *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 1. 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

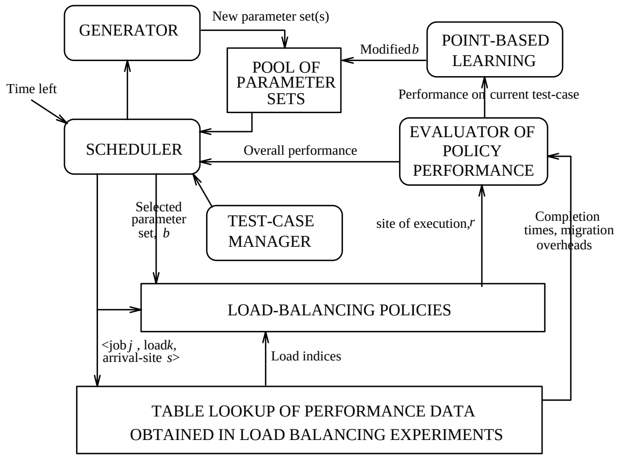

GENERATOR
New parameter set(s)
POOL OF PARAMETER SETS
Modified b
POINT-BASED LEARNING
Time left
Performance on current test-case
SCHEDULER
Overall performance
EVALUATOR OF POLICY PERFORMANCE
Selected parameter set, b
TEST-CASE MANAGER
site of execution, r
Completion times, migration overheads
LOAD-BALANCING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

Figure 7. Phase 4: Policy learning

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 7 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 1. 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 7), 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, b. It then uses the information contained in the test-case to compute the load index of each site at the time of job j's arrival. Thus, given the values of policy parameters, as well as the values of primitive and abstract decision metrics, one can determine the policy's decision for the chosen test job. This decision involves the choice of a remote destination (r) where the incoming job j should be executed.

Given the information contained in the test-case, and knowing the decision (r) of the policy, the performance of the policy parameterized by b 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 b for the given test case can be calculated.

The *Evaluator of Policy Performance* (Figure 7) 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.* [13, 14]. Parts of our learning algorithm are similar to those used in genetics-based learning except that tests of each population are expensive and that test results are noisy.

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 Section 4. 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.

3.4. *Application Phase*

With the new load-index functions and the selected policy parameters in place, the scheduling system is now ready for application. Figure 8 shows the information flows during the application phase. At the site of arrival (s) of an incoming job, the sender-side rules of site s are instantiated. This figure shows the case when a remote destination (site r) is selected by the sender-side rules of site s. A request is made to site r, which instantiates its receiver-side rules. If the receiver-side rules at site r succeed, then the site of arrival (s) initiates a job transfer by remotely invoking the job-execution facilities at site r.

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 illustrated in the previous subsection.

3.5. *Overheads in Learning*

Table I summarizes the times required by the different phases of SMALL. The typical times reported in the rightmost column are based on our 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

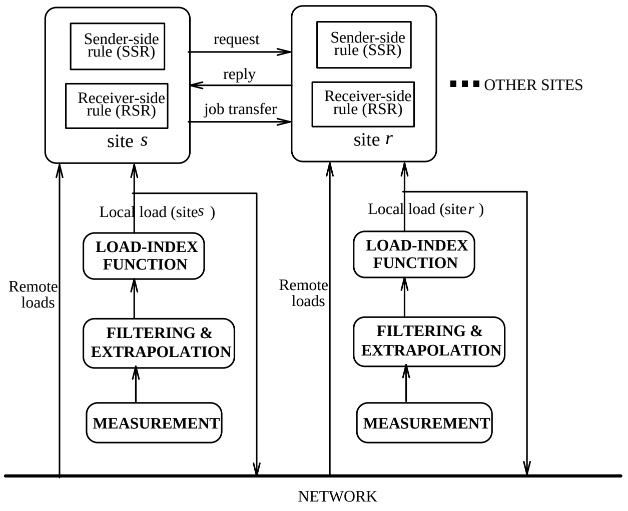


Figure 8. Phase 5: Application phase

Table I
Times required by different phases of SMALL's operation
(Values used in estimating typical times are: $n = 10$, $p = 24$, $d = 4$, $l = 15$ minutes, $N = 60$, $c = 0.064$ seconds, $e = 6$, $T = 15$ minutes, $s = 12$, $f = 1{,}000$, $k = 4$, and $t = 0.2$ seconds.)

Phase	Formula for Time Needed	Typical Time
I	$l(p+1)$	6¼ hours
II	$n \times d \times p \times l$	10 days
III	$N \times d \times p \times c \times e\,(p+d-2)$	15 hours
IV	$s \cdot max\,(T, f \times b \times t)$	3 hours
V	New sites, new configuration, or OS upgrade: 10 days (on-line data collection) 15 hours (off-line retraining of comparator networks) 3 hours (off-line tuning of policy parameters) Removal of a site, or new load-balancing software 3 hours (Offline tuning of policy parameters)	

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, $N \times p\,(p-1)d$ and $N \times p \times d\,(d-1)$, where N is the number of checkpoints in a benchmark program. (A checkpoint in a program can be considered as a point of time at which the program is stopped and migrated.) 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 b 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 b sets are likely to be good. Also, the time spent on poor parameter sets is usually negligible as 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 the remaining sections of this paper, we present our approach to the automated tuning of load-balancing-policy parameters using integrated (population-based and point-based) learning. Details for the other components are not included due to space limitation [8, 9].

4. INTEGRATED LEARNING OF LOAD-BALANCING POLICIES

4.1. The Policy-Learning Problem

Figure 1 shows the sender-side and receiver-side rules for the load-balancing policies considered in this paper. There is one set of rules per site. The rules are 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: threshold θ_2. Altogether, there are 4 parameters per site, or $4N$ parameters for the whole system. In this paper, we use the term *heuristic method (HM)* [13] to denote a set of system-wide parameters.

Figure 9 shows our representation of HMs. Later, 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.

The load-balancing policies are applied in a distributed fashion. When a job arrives at site s, 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 $\delta(i)$-neighborhood of *Reference*(i). Out of these, one destination (say, r) is picked at random. Depending upon the value of *Reference*(s), the SSRs compare either the absolute or the relative value of *Load*(s) against the threshold $\theta_1(s)$. If *Load*(s) is large enough, then the rule for sending succeeds, and the SSRs invoke the RSRs at site r; otherwise, the job is executed locally at site s. The RSRs at site r compare *Load*(r) against threshold $\theta_2(r)$. The request from the SSRs at site s is turned down if *Load*(r) is too high; in this case, the job is executed locally at site s. Otherwise, the request is accepted, and the remote site (s) 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.

Randomization in selection of remote destinations avoids instability in distributed scheduling algorithms [11]. 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.

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 9. Representation of one set of load balancing policies for N sites

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 is 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 *d* sites. We use the word *test* in this paper 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* [10]: 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.* [13, 14], whose TEACHER (TEchniques for Automated Creation of HEuRistics) system [6, 13] provides a domain-independent approach to rational scheduling of limited learning time between generation and testing.

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.

4.2. Architecture and Operation of TEACHER

The architecture of TEACHER as applied to learning load balancing policies is shown in Figure 7. 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 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.

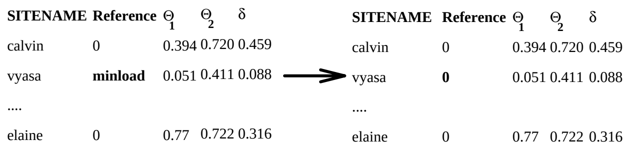
Within each generation, tests need to be allocated rationally between different HMs. The traditional method of solving this problem is to apply decision-theoretic sampling techniques developed for the ranking and selection problems [3, 5, 12]. However, these techniques have several limitations: (i) they 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 policies that overcome the aforementioned limitations.

Besides supporting generation and testing 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.

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.

4.3. Operators for Population-Based Learning

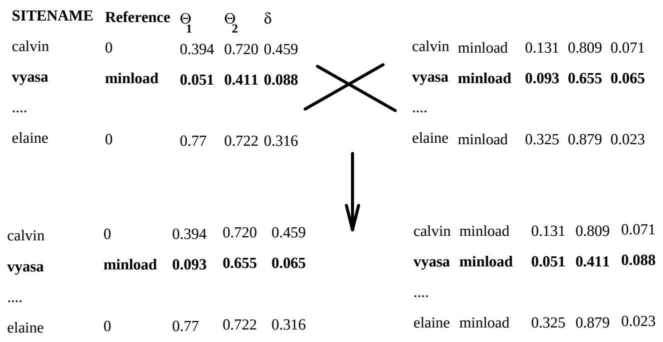
Figures 10 and 11 illustrate the genetic operators used in the generation of new HMs from existing ones. The initial HMs can either be generated randomly or provided by the designer. There are three operators in our system: (i) *Mutation*: the *Refer-*



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

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

Figure 10. An example of mutation applied to an HM



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

calvin	minload	0.131	0.809	0.071
vyasa	**minload**	**0.093**	**0.655**	**0.065**
....				
elaine	minload	0.325	0.879	0.023

calvin	0	0.394	0.720	0.459
vyasa	**minload**	**0.093**	**0.655**	**0.065**
....				
elaine	0	0.77	0.722	0.316

calvin	minload	0.131	0.809	0.071
vyasa	**minload**	**0.051**	**0.411**	**0.088**
....				
elaine	minload	0.325	0.879	0.023

Figure 11. An illustration of the crossover-1 operation

ence 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.

Following the customary practice in genetic search [4], 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.

4.4. 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.

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

Adjustment of $\theta_1(s)$:

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

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

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

Figure 12. Rules for point-based learning

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 12.

5. 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 is shown in Figure 7. 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 is 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. In order 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 and 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.

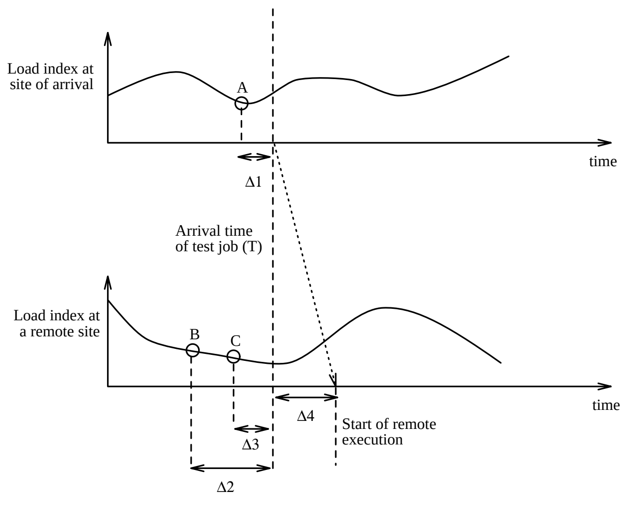
5.1. Cloning for Simulating Larger Systems

All our sites are assumed to be architecturally homogeneous; however, we do allow configurational heterogeneity. As is explained earlier, 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.

5.2. Modeling of Overheads and Delays

Figure 13 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 $\Delta 1$ and $\Delta 3$ depend upon the interval of computation of load-index function. Such delays affect the currency of both the local and the remote load indices. The delay $\Delta 2$ includes both the delay in computing the load index at the remote site and the delay in propagating that value to site of arrival. Finally, the delay $\Delta 4$ represents the overhead of migration.



Load index at site of arrival
A
time
Δ1
Arrival time of test job (T)
Load index at a remote site
B
C
time
Δ4
Start of remote execution
Δ3
Δ2

Figure 13. Overheads and delays in load balancing

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 comparator network computing it had 40 hidden units, then the computation of this function requires 1680 (40×40 + 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 up to the decision point, the effects of delays due to computation and communication of load indices can be simulated by using older values of load indices, rather than their values at time T.

In the next section, we discuss results on several learning experiments involving various combination of these factors.

6. 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 is 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 II shows the values of *M* and *I* in the left two columns. For each combination of *M* and *I* values, a different set of data are prepared as follows. To 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 methods described in Section 3.

TEACHER was allowed 30 minutes learning time for each experiment on a Sparcstation 10, Model 20. Twelve experiments were performed, one for each parameter set described in Table II. The best policy found by TEACHER after 30 minutes was fully evaluated on all the test-cases. The average speed-up of that policy and the corresponding standard deviation are shown in the third and fourth columns of Table II. 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 14a shows a contour plot of the best policy learned. This contour plot was created by finding the probability of achieving certain speedup over local execution (represented in the *X*-axis) for a checkpoint at a specific time in seconds from the start of a job (represented in the *Y*-axis). A total of 10 jobs and 24 patterns were used in computing these probabilities. 19 contours, each connecting *X-Y* points having equal cumulative probability of speedup, are shown; they divide up the space into 20 regions of equal (5%) probability. This figure shows that the best speed-up values are achieved for jobs that are neither too long nor too short. The quality of our indices drops with the length of the job; therefore, long jobs cannot be speeded up by one-time placement; they need 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.

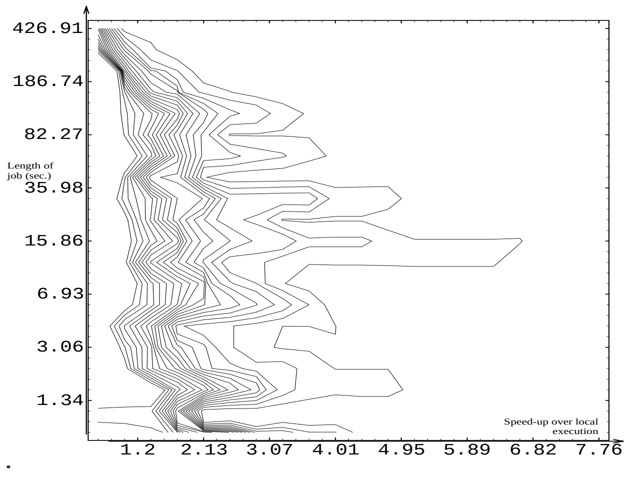
Figure 14b shows the behavior of policies learned under high migration overheads but small interval of load-index computation. Relative to Figure 14a, high speed-up values are less probable when migration overhead is significant. Figure 14c shows the case when migration overheads are small but the interval 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 14a, but they are much more likely than in the case of high migration overheads.

7. CONCLUSIONS

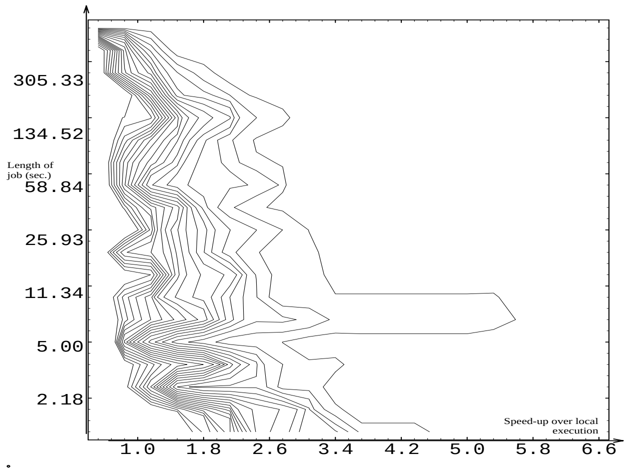
We conclude that (i) TEACHER discovers policies with significant speed-ups over local execution under a variety of assumptions about overheads and delays; (ii) the quality of policies learned by TEACHER is adversely affected by both large migration overheads and long intervals of load-index computation; and (iii) migration overhead is the dominant factor affecting the quality of policies learned.

Table II
Performance of policies learned under different assumptions

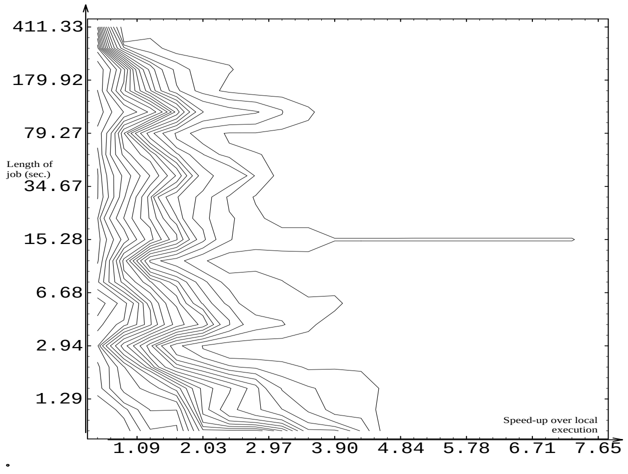
M (ms)	I (ms)	Average Speedup	Std. Dev. Speedup
100	100	1.73743	0.226114
	500	1.49916	0.219608
	1000	1.50908	0.203219
500	100	1.50156	0.179138
	500	1.53047	0.152211
	1000	1.6004	0.17311
2000	100	1.2397	0.0694498
	500	1.15947	0.112387
	1000	1.17821	0.124843
5000	100	1.25181	0.128558
	500	1.19149	0.14072
	1000	1.31023	0.134078



(a) *M*=100, *I*=100, *P*=20



(b) *M*=5000, *I*=100, *P*=20



(c) *M*=100, *I*=1000, *P*=20

Figure 14. Contour plot of cumulative probability of speed-up

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.

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.

8. REFERENCES

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

[2] 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.

[3] 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.

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

[5] 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.

[6] 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.

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

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

[9] P. Mehra and B. W. Wah, ‘‘Automated Learning of Workload Measures for Load Balancing on a Distributed System,’’ *Proc. Int’l Conference on Parallel Processing,* pp. III-263-III-270, CRC Press, Aug. 1993.

[10] H. Simon, *The Sciences of the Artificial,* The MIT Press, Cambridge, MA, 1982.

[11] J. A. Stankovic, ‘‘Stability and Distributed Scheduling Algorithms,’’ *Trans. on Software Engineering,* vol. SE-11, pp. 1141-1152, IEEE, Oct. 1985.

[12] 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.

[13] 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.

[14] B. W. Wah, A. Aizawa, and A. Ieumwananonthachai, ‘‘Real-Time Learning of Heuristics,’’ *Trans. on Knowledge and Data Engineering*, IEEE, (under revision) 1993.

[15] 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.