GAMMON: A Load Balancing Strategy for Local Computer Systems with Multiaccess Networks

KATHERINE M. BAUMGARTNER, MEMBER, IEEE, AND BENJAMIN W. WAH, SENIOR MEMBER, IEEE

Abstract—**This paper investigates an efficient load balancing strategy, GAMMON (global allocation from maximum to minimum in constant time), for distributed computing systems connected by multiaccess local area networks. The broadcast capability of these networks is used to implement an identification procedure at the applications level for the maximally and the minimally loaded processors. The search technique has an average overhead which is independent of the number of participating stations. An implementation of GAMMON on a network of SUN workstations is described. Its performance is found to be better than other known methods.**

Index Terms—**Broadcast, collision detection, dynamic programming, load balancing, multiaccess networks.**

I. INTRODUCTION

LOAD balancing uses communication facilities in a distributed computing system to support remote job execution in a user-transparent fashion in order to improve resource utilization and reduce response time. A decision to load balance a job is made if the job is likely to be finished sooner when executed remotely than when executed locally. Load balancing has been found to be essential because a job will almost always be waiting for service at one processor while another processor is idle in a system with ten or more processors [15].

Load balancing decisions can be made in a centralized or in a distributed manner. A *centralized* decision implies that status information is collected, and decisions to load balance are made at one location. An example would be a system with a job scheduler at one location that collects jobs and dispatches them to stations for processing. Theoretical studies on centralized load balancing have been made by Chow and Kohler [5] and Ni and Hwang [17]. The disadvantage of centralized scheduling is the overhead of collecting processor status information and jobs. When this overhead is large, scheduling decisions are frequently based on inaccurate and outdated status information. In contrast, a *distributed* load balancing scheme does not limit the scheduling intelligence to one processor. It avoids the bottleneck of collecting status information and jobs at a single site and allows the scheduler to react quickly to dynamic changes in the system state.

Load balancing can also be classified as state-dependent or probabilistic [5]. A decision based on the current state of the system is *state-dependent*. A decision is *probabilistic* if an arriving job is dispatched to the processors according to a set of branching probabilities which are collected from previous experience or are based on system characteristics. In the case that the branching probabilities are derived from the service rates of processors, the strategy is called *proportional branching* [5]. It was found that a probabilistic strategy for a single job class performed better than a proportional branching strategy with a single arrival stream [17]. An optimal probabilistic algorithm for multiple job classes was found to be easier to implement than state-dependent strategies. An optimal probabilistic load balancing algorithm with multiple arrival streams has also been shown [19]. Other research on load balancing include studies characterizing state-dependent load balancing, determining appropriate state information [8], proposing efficient algorithms [4], [7], [13], [15], [25], and topology-dependent strategies [6], [10]–[12], [21], [22].

State-dependent load balancing is implemented on the Purdue Engineering Computer Network, which is a system of computers connected by a hybrid of Ethernet and point-to-point links [9]. The load balancing decisions are distributed: each processor decides whether to send its jobs for remote execution. A processor polls other processors for status information about their loads, decides which processor has the lowest load, and sends the job for remote processing if the turnaround time is shorter.

Some results of these previous studies are as follows.

1) A network with load balancing performs better than a network without load balancing.

2) State-dependent load balancing strategies perform better than probabilistic strategies, but have higher overhead.

3) Probabilistic strategies are sometimes insensitive to dynamic changes in system load and may result in suboptimal performance.

4) Load balancing decisions considering the state of the source only do not have the potential for performance improvement that decisions considering the state of the destination do [25].

5) Extensive state information is not needed for effective load balancing and can be detrimental to system performance [7].

6) Status information used in a state-dependent decision

Manuscript received September 21, 1988; revised April 10, 1988. This work was supported by the National Aeronautics and Space Administration under Contract NCC 2-481 and the National Science Foundation under Grant DMC 85-19649.

K. M. Baumgartner is with Digital Equipment Corporation, Maynard, MA 01754.

B. W. Wah is with Department of Electrical and Computer Engineering and the Coordinated Sciences Laboratory, University of Illinois at Urbana-Champaign, Urbana, IL 61801.

IEEE Log Number 8928532.

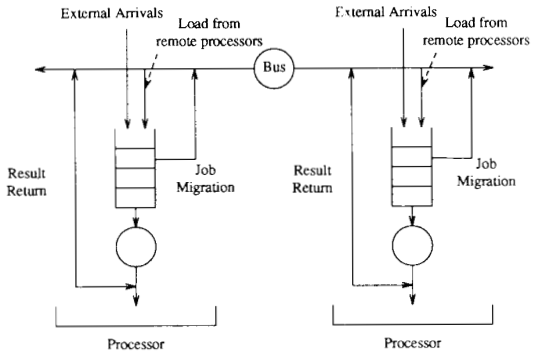


Fig. 1. Queueing diagram of a system of processors connected by a broadcast bus.

must be readily available. Decisions based on outdated or inaccurate status information could degrade performance.

7) Load balancing increases network load which can impede message transmissions.

This study considers load balancing on local computer systems connected by multiaccess networks. These networks have a broadcast bus topology that allows only one job or message to be sent across the network at a time. Response time is the amount of time elapsed from job submission to job completion and is an indication of the processor load. Due to the constraint of sending one job at a time across the bus, an efficient load balancing strategy is to send a job from the processor with the maximum load to the processor with the minimum load when the overhead of sending these jobs and identifying the participating stations is small. This paper proposes a strategy for load balancing that can be implemented at the applications level on *existing* systems. The strategy uses an efficient technique to identify the minimally and the maximally loaded processors with constant average overhead. The strategy is called GAMMON: global allocation from maximum to minimum in constant time.

The organization of this paper is as follows. The section following this introduction gives an overview of GAMMON. Section III shows a window protocol that can be used for distributed extremum search on bus networks and that requires hardware modification to existing network interfaces. Section IV extends this protocol for implementation on existing systems without hardware changes. The implementation of GAMMON is described in Section V, and concluding remarks are drawn in Section VI.

II. Global Scheduling Strategy

A model of the system under consideration is shown in Fig. 1. There are multiple identical processors connected by a broadcast bus. Each processor can have arrivals external to the system or from the bus. Jobs are modeled as independent tasks. If jobs are migrated to a processor across the bus, the results must be returned to the originating processor when execution is completed. Moreover, the queue at each processor is finite: only a limited number of jobs may be waiting for execution.

A good load balancing procedure should avoid the occurrence of the idle-while-waiting condition, as well as any state which makes idle-while-waiting more likely. In a batch processing system, idle-while-waiting will not occur when there is at least one job at each processor at any time. Hence, the likelihood of idle-while-waiting can be minimized if jobs are evenly distributed. In a multiprogrammed system, assuring that processors are busy is not sufficient to minimize the occurrence of the idle-while-waiting condition; it is important to distribute all available jobs evenly in order to have a reasonable response time for every job.

The strategy discussed here uses the queue length of active jobs at a processor as a metric to indicate workload. A queue length imbalance will make the idle-while-waiting condition more likely. Hence, load redistribution is needed when there is significant difference between queue lengths, such that the estimated total overhead of migrating a job and queueing delay at a remote processor and later returning results is less than the delay a job would experience at its source processor.

An important point here is that the number of jobs at a processor, while frequently a good reflection of load, is not always adequate. Other factors that may contribute to the workload at a processor include physical differences of processors (such as speed or size of main memory), paging activity, and the ratio of processing activities and input/output activities in jobs. Future work will involve investigating a more inclusive measure of processor load [24].

An ideal redistribution of jobs, given that the metric used is the queue length at each processor, is to have equal number of jobs at each processor. Since a single bus connects all the computers, only one job can be migrated at any one time. The best strategy is to take a job from the maximally loaded processor, and send it to the minimally loaded processor. Such a strategy is both source- and sink-initiated. Furthermore, it requires the minimal amount of status information transferred.

Three basic scheduling operations are required for this redistribution on a bus network: identification of the maximally and the minimally loaded processors, job migration, and result return. Migrating jobs and returning results are straightforward because existing communication facilities can be used. However, identifying processors with the load extremes efficiently is more difficult. Such an operation should have very low complexity, preferably independent of the number of processors connected to the bus. Any centralized scheduling algorithm, such as polling, is not suitable here. Efficient algorithms are studied in Sections IV and V.

The three basic scheduling operations, in addition to regular message transfers, must be prioritized in order to achieve the best performance. Regular message transfer is assigned the highest priority, since it is the original purpose of the network. The priorities of the remaining tasks are determined by considering the relative overheads in terms of the additional total system delay (the sum of the delays of all jobs) incurred.

First, the relative priority of identifying the ith maximally/minimally-loaded-processor pair and migrating the job between the $(i - 1)$th pair is determined. Two cases are considered. The first is when there are idle processors. If job migration is done first, then the migrated job can begin execution immediately upon arrival at the destination processor, so its delay is increased by the time required to send it

across the network. In contrast, if the max/min identification is done first, then the same job will incur the delay of that operation as well. Clearly, migrating the job first is better. The second case is when there are no idle processors. The ordering of the tasks is not critical in this case, since migrating the job first does not immediately contribute to reducing the job delay, as the job may not begin execution upon arrival at the destination. This result is true for any job migration and max/min identification, and performing the migration first will result in a total delay equal to or smaller than performing the max/min identification first. Consequently, job migration should have priority over max/min identification.

Next, the relative priority between result return and job migration is considered. When there are no idle processors, delay is added directly to the job waiting for result return. Since the job waiting for job migration will not be able to begin execution immediately upon arrival at the destination, result return should take precedence. When there are idle processors, reducing either the delay for job migration or the delay for result return will reduce the overall delay. In short, performing the result return first always improves the overall performance as much or more than performing job migration first. It is also easy to see that result return should always have precedence over max/min identification by a similar argument.

In summary, the priority ordering for tasks using the bus network is 1) regular message transfer, 2) result return, 3) job migration, and 4) max/min identification.

The scheduling strategy GAMMON consists of two steps that are executed repeatedly. The first is to determine which of the current tasks has the highest priority, and the second step is to execute that task. Due to the ordering of the priorities, only one job will ever be waiting for job migration, but potentially more than one may be waiting for result return.

A consideration with priorities are the overhead of priority resolution among tasks and the overhead of the tasks themselves. The discussion above assumes that each of the steps has similar overhead. If the overheads associated with tasks are considerably different, priority enforcement changes. A specific case is processors sharing a common secondary storage. Job migration and result return have lower overhead in such a system than in one with a shared disk. File transfer is not explicitly needed as all processors have access to the common secondary storage. Furthermore, if the overhead of resolving priorities is large, then it is more efficient not to schedule tasks according to priorities. Such a tradeoff is performed in the implementation of the load balancing algorithm, which is presented in Section V.

III. Window Protocol for Distributed Extremum Searches

Carrier-sense-multiaccess networks with *collision detection* (CSMA/CD) are a type of local-area network with packet switching and a bus topology [18]. CSMA/CD networks evolved from CSMA networks that have *listen-before-talk* protocols to avoid overlapping transmissions. The collision-detection ability of CSMA/CD networks allows processors to additionally *listen-while-talk,* so collisions resulting from simultaneous transmissions can be detected and stopped immediately.

There are three types of protocols for contention resolution CSMA/CD networks. Collision-free protocols strictly schedule bus accesses, so no collisions occur. Contention protocols function at the other extreme by allowing processors to transmit whenever they find the bus idle. When collisions occur because of simultaneous transmissions, processors stop transmitting, wait for some prescribed amount of time, and try again. The backoff algorithm of Ethernet [16] is an example in this class. The disadvantage of collision-free protocols lies in the overhead of waiting for transmission, while the disadvantage of contention protocols is the time wasted during collisions. A third type of contention-resolution protocol is the limited-contention protocol. This type of protocol chooses a processor for transmission from among those waiting to transmit based on *a priori* information, such as the channel load.

The *virtual-window protocol* (VWP) proposed by Wah and Juang [10], [21]–[23] is an example of a limited-contention protocol. It is based on a three-state collision-detection mechanism. After each attempted broadcast, there are three possible outcomes: *collision* (more than one broadcast), *idle* (no broadcast), and *success* (exactly one broadcast). The protocol can be adapted easily to perform distributed extremum searches in a load balancing strategy. In the remainder of this section, we briefly explain this protocol and discuss its limitations.

Stations wishing to transmit packets participate in a *contention period* that consists of a number of *contention slots.* Each station generates a random number called a *contention parameter* that is used for the entire contention period. The parameter is in an interval with upper and lower bounds U and L, respectively. Without loss of generality, assume that the station with the minimum contention parameter is sought. The results developed apply to the case in which the station with the maximum contention parameter is to be found. Successive choices of smaller intervals in each contention slot attempt to isolate the minimum contention parameter.

For regular message transfers, each station has equal chance of being chosen for transmission, so the contention parameters are random numbers generated from a uniform distribution in the interval (0, 1]. The stations maintain a common window (or interval) for contention. In a contention slot, stations having contention parameters within the window broadcast a short signal to contend for the channel. If a collision or no transmission occurs, the window boundaries are adjusted in parallel for the next contention slot. Stations having contention parameters outside the window stop contending and wait for the next contention period. The above steps are repeated until a single station is isolated in the window. This station is the winner and is allowed to transmit its packet. The distribution of the contention parameters and an estimate of the channel load are used to update the window efficiently, so the number of contention slots is kept to a minimum.

The global window required in the protocol can be maintained by updating an initially identical window with a common algorithm and using the identical information broad-

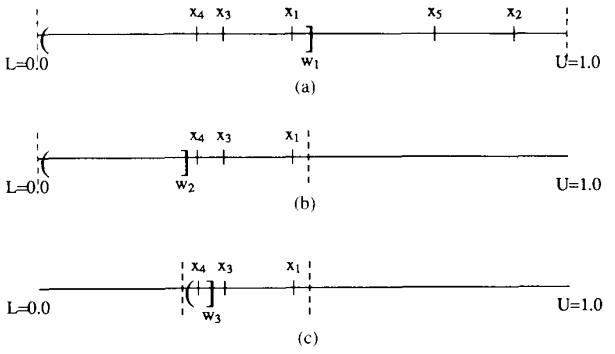


Fig. 2. Example of the virtual-window protocol. The dashed lines indicate the portion of the interval being searched during the current contention slot. The current window, enclosing stations eligible to contend, is delimited by [ ]. (a) First iteration. (b) Second iteration. (c) Third iteration.

cast on the bus. Assuming that the information broadcast is received correctly by all stations, the global window will be synchronized at all sites.

An example of the VWP is shown in Fig. 2. There are five processors contending, and station i has contention parameter x_i. In this example, $x_1 = 0.48$, $x_2 = 0.90$, $x_3 = 0.35$, $x_4 = 0.30$, and $x_5 = 0.75$. These contention parameters were chosen arbitrarily, but for different purposes they may reflect processor loads or priorities. The windows chosen in these examples are not the optimal windows but are chosen to illustrate the characteristics of the protocols. w_1, the upper bound for the first window chosen, is 0.51. All stations with contention parameters less that or equal to 0.51 are allowed to broadcast; in this case, stations 1, 3, and 4. The result of this contention slot is a collision; the interval to be searched is updated to (0, 0.51]; and stations 2 and 5 are eliminated from further contention. w_2, the upper bound for the next window, is 0.25. The result of the second contention slot is idle (no broadcast), so the interval is updated to (0.25, 0.51]. No stations were eliminated as a result of this contention slot. For the third contention slot, the upper bound of the window is chosen to be 0.32. The result is a successful transmission, so station 4 is isolated and "wins" the contention.

The window-selection process can be formulated as a dynamic programming algorithm, and details have been shown elsewhere [22]. Analyses and simulations have shown that contention can be resolved in an average of 2.4 contention slots, independent of the number of contending stations and the distribution function of the contention parameters, if the parameters are independent and identically distributed [21].

A major limitation of the VWP is that its implementation requires minor hardware modifications of existing Ethernet interfaces [22]. At the applications level, such modifications are not always possible. Many existing networks do not make three-state collision-detection information available to the applications software because a contention slot is a small amount of time (50–100 μs) relative to the time required to propagate information through all levels of software to the applications level (hundreds of microseconds). Consequently, a different protocol must be developed for distributed extremum search at the application level. Several alternatives are discussed in the next section.

IV. Window Protocols with Two-State Collision Detection

At the applications level, each station has an independent *search parameter,* and an iteration is a *broadcast slot* which is a contention resolution at the network interface followed by a broadcast of a message to all stations. A broadcast slot has two possible outcomes, *idle* (no stations attempt to broadcast), or *transmission* (one or more stations attempt to broadcast resulting in contention resolution, and one station broadcasts its search parameter). A broadcast slot may consist of a number of contention slots, and information about each contention slot is not sent to the applications level.

There are a number of differences between searching at the applications level and the network level.

1) The contention parameters are the search parameters for the VWP, which is not the case for the window protocol at the applications level.

2) An iteration of the VWP is a contention slot with three possible outcomes: idle, collision, and success. An iteration of the window protocol at the applications level is a broadcast slot with two possible outcomes: transmission and idle.

3) An iteration of the VWP takes less time than an iteration of the window protocol at the applications level. Normally, a contention slot takes tens of microseconds, while broadcasting a short message takes hundreds of microseconds.

Since the information available for window selection is different at the applications level, the decision process has to be modified. Three possible window-search strategies to identify the minimum are described below. The identification of the maximum is similar and is not described. Further details about these strategies can be found elsewhere [1], [2]. In contrast to the VWP, dynamic programming methods to optimize window choices cannot be used here because the Principle of Optimality is violated. The performance of these strategies is compared using the number of broadcast slots they require to isolate the minimum search parameter. These strategies assume that information about the distribution of search parameters is available. This distribution is characterized experimentally in Section IV-D.

A. One-Broadcast Strategy

The one-broadcast strategy allows a maximum of one broadcast slot per iteration. Starting with an interval $(L, U]$, each station has a search parameter x_i in the interval. The stations maintain a global window in the interval. Stations with parameters within the window attempt to broadcast their search parameters, and if there are one or more parameters in the window, there will be a contention resolution followed by a broadcast of one of the search parameters. The upper bound of the interval will be updated to the value broadcast. If there are no parameters within the window, the lower bound of the interval is updated to the upper bound of the window used, and the protocol continues. The minimum is identified when the lower bound of the interval is equal to the upper bound.

An example of the one-broadcast strategy is shown in Fig. 3. The stations and parameters are the same as those in Fig. 2. In the first iteration [Fig. 3(a)], the upper bound of the window chosen is 0.51. Stations 1, 3, and 4 attempt to broadcast their

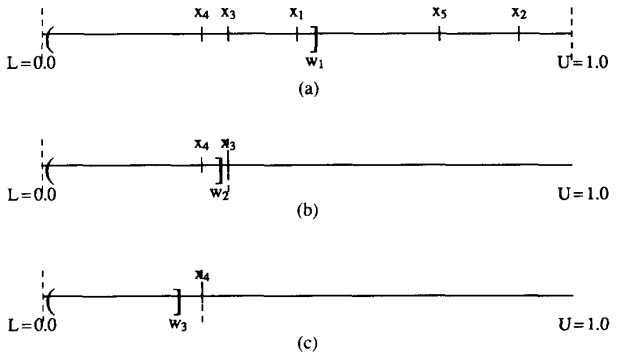


Fig. 3. Example of the window protocol using the one-broadcast strategy. The dashed lines indicate the portion of the interval being searched during the current broadcast slot. The current window is delimited by [ ]. (a) First iteration. (b) Second iteration. (c) Third iteration.

parameters. Suppose that station 3 is the winner and transmits. The next interval to be searched is $(0, x_3)$. Let the upper bound of the next window chosen be 0.33. Only station 4 trys to transmit its parameter, and x_4 is broadcast. The search has not concluded even though x_4 is the minimum because the fact that it was the only station broadcasting is not available to station 4 or to the other stations. The next window chosen is 0.25. There is, of course, no broadcast. This process will continue until the bounds of the window isolate x_4, and the minimum is globally known.

The choice of the window in each broadcast slot is based on the probabilities of the two states, transmission and idle, which are dependent on previous broadcasts. If a previous broadcast slot resulted in the transmission of a value, say x_{b1}, then any subsequent transmissions must be less than x_{b1}. This implies that any subsequent x_i's broadcast were eligible to broadcast during the iteration that x_{b1} was broadcast, but lost the contention. The probability of the subsequent transmissions must be conditioned on the fact that any x_i's in the current window did not broadcast when they were eligible during previous iterations. The choice of the window is, thus, dependent on previous broadcasts, and the Principle of Optimality is not satisfied. As a result, the choice cannot be optimized by dynamic programming methods.

Assume that station i has an independent search parameter x_i with distribution $F(x)$ and density $f(x)$. The following definitions are used to formulate the problem of choosing the upper bound of the next window as a recurrence after k broadcasts.

$N_{E1}(a, b, v^k, q^k)$: the minimum expected number of broadcast slots to isolate the minimum x_i in the interval $(a, b]$ using a one-broadcast strategy, given that there have been k previous broadcasts with values and corresponding upper bounds of windows stored in the k-element arrays v^k and q^k, respectively;

$\phi_E(a, b, w, v^k, q^k)$: the probability of a transmission in the interval $(a, w]$, given that there have been k previous broadcasts with values and corresponding upper bounds of windows stored in the k-element arrays v^k and q^k, respectively;

$\theta_E(a, b, w, v^k, q^k)$: the probability of idle in the interval $(a, w]$, given that there have been k previous broadcasts with values and corresponding upper bounds of windows stored in the k-element arrays v^k and q^k, respectively.

The notation v^k and q^k indicates a set of k values broadcast and the corresponding upper bounds of windows used. v_k and q_k are the kth value and the corresponding upper bound of window used. It follows directly from the above definitions that

$$\phi_E(a, b, w, v^k, q^k)+\theta_E(a, b, w, v^k, q^k)=1.0. \quad (1)$$

After k successful broadcasts, there are $2(k+1)$ subintervals in the interval $(a, U]$. They are $(a, w]$, $(w, v_k]$, $(v_k, q_k]$, $(q_k, v_{k-1}]$, $\cdots$, $(v_1, q_1]$, and $(q_1, U]$. For reference, they can be numbered from left to right and from 1 to $2(k+1)$. Let s be a set of elements $\{s_i\}$, where s_i is the number of x_i's in the ith subinterval. Let S be the set of s that are possible with the previous windows and values broadcast, and let I be a subset of S such that $s_1 = 0$. The set I is the subset of S that corresponds to a distribution of x_i's such that there will be no x_i's in $(a, w]$ and that the result of the broadcast slot is idle. Then

$$\theta_E=\frac{\sum_{s\in I}\left(\Pr[\text{arrangement } s]\sum_{i=1}^{k}\Pr[v_i \text{ broadcast with a window upper bound } q_i|s]\right)}{\sum_{s\in S}\left(\Pr[\text{arrangement } s]\sum_{i=1}^{k}\Pr[v_i \text{ broadcast with a window upper bound } q_i|s]\right)}. \quad (2)$$

The probability of a given arrangement is found using the distribution function $F(x)$. Let $b(a, b, i) = [F(b) - F(a)]^i$ then

$$\Pr[\text{arrangement } s]=\binom{n}{s_1} b(a, w, s_1)\binom{n-s_1}{s_2} \cdot b(w, v_k, s_2)\cdots\binom{n-\sum_{i=1}^{2k} S_i}{s_{2k+1}} \cdot b(v_1, q_1, s_{2k+1})b(q_1, U, s_{2k+2}). \quad (3)$$

$\Pr(v_k \text{ broadcast with a window upper bound } q_k|s)$ is easily determined because each station in the subinterval search has equal probability of winning and broadcasting in a broadcast slot, so

$$\Pr(v_i \text{ broadcast with a window upper bound } q_i | s) = \frac{1}{\sum_{j=1}^{2(k-i+1)} s_j}. \quad (4)$$

Using a conditional density function

$$f_c(a, w, x_b) = \frac{f(x_b)}{\Pr(a < x_b \leq w)} = \frac{f(x_b)}{F(w) - F(a)}, \quad (5)$$

the choice of the upper bound of the next window is formulated as a recurrence. Let

$$\gamma(a, w, v^k, q^k) = \int_a^w f_c(a, w, x_b) \cdot N_{E1}(a, x_b, v^{k+1}, q^{k+1})\, dx_b \quad (6)$$

then

$$N_{E1}(a, b, v^k, q^k) = \min_{a<w<b} \{1 + \phi_E(a, b, w, v^k, q^k)\gamma(a, w, v^k, q^k) + \theta_E(a, b, w, v^k, q^k) N_{E1}(w, b, v^k, q^k)\} \quad (7)$$

with

$$N_{E1}(a, b, v^k, q^k) = 1 \qquad \text{for all } b = a. \quad (8)$$

The first term on the right-hand side of (7) counts the current broadcast. The second term is the expected number of additional broadcast slots to isolate the minimum if the current broadcast slot results in a transmission. γ is the weighted average number of broadcast slots for the value broadcast, x_b, and the probability that this value was broadcast. The third term is the number of additional broadcast slots if the current broadcast slot is idle.

Boundary conditions must be set to terminate the evaluations after a reasonable number of broadcast slots. In practice, the x_i's may represent indistinguishable physical measures when their difference is less than δ. It is assumed that when the window size is smaller than δ, the probability that two stations have generated parameters in this interval is so small that contention can always be resolved in one step. The boundary condition becomes

$$N_{E1}(a, b, v^k, q^k) = 1 \qquad \text{for all } (b - a) < \delta. \quad (9)$$

The optimal window choices found in (7) and (9) can be organized by the decision tree shown in Fig. 4. The top of the structure contains the roots of decision trees with different numbers of processors. For a given n_i, there is an initial window $w_{1,1}$, and two pointers to substructures corresponding to the two outcomes: transmission and idle. Note that the substructure for a transmission contains windows for each of the possible values that can be transmitted in the subinterval. Each box in the structure corresponds to a decision point, and the contents of the box, $w_{a,b,c}$, is the window upper bound for the current broadcast slot. The subscripts of the window upper bound indicate the iteration number, whether the last iteration outcome was transmission or idle (transmission = 1, idle = 2), and the value broadcast if the last iteration was a transmission. There are two branches from each decision point corresponding to the two possible outcomes in each broadcast slot. Starting from the root, if the broadcast slot results in a transmission of b, then the search will terminate if $(b - L) < \delta$; otherwise, the search will continue with a new decision point corresponding to the interval $(L, b]$. If the broadcast slot is idle, then the search will terminate if $(U - w_{1,1}) < \delta$; otherwise, the search continues with the interval $(w_{1,1}, U]$.

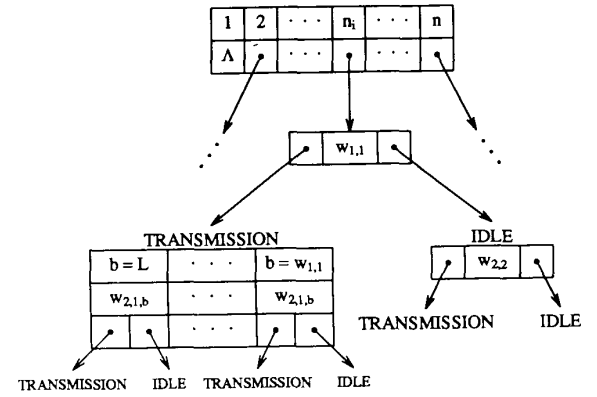


Fig. 4. Data structure for the exact solution of the one-broadcast strategy. The data structure shows a method for storing the windows and indicates the tree-structured progression of window choices.

The data structure in Fig. 4 shows the final window choices, but during computation of the best window choice for each decision point, all possible choices of windows have to be tried. This evaluation process is extremely complex. For each possible window, there can either be a transmission or no transmission. If there is a transmission, all possible values within the window must be considered as the possible value broadcast. Each level of the tree indicates the outcome of an iteration. For every decision in the exact solution, the entire tree above the current decision point, which is determined by v^k and q^k, must be taken into consideration in computing the next set of branching probabilities. With $\delta = 1/(10n)$, and $n = 5$, there are 69 007 690 decision points, and for $n = 6$, the number increases to 8 501 194 726. The number of decision points increases so rapidly that the problem becomes intractable. Fortunately, reasonable results can be obtained using a heuristic decision based on the current upper and lower bounds only.

In the approximate solution, the probabilities of transmission and idle are assumed to be independent of previous broadcasts and are computed without information from previous broadcasts. The following definitions are used.

$N_{A1}(a, b)$: the minimum expected number of broadcast slots to isolate the minimum x_i in the interval $(a, b]$ using an approximate solution of the one-broadcast strategy, given that all x_i's are in $(a, U]$, and that at least one x_i is in $(a, b]$;

$\phi_A(a, b, w)$: the probability of a transmission in the interval

$(a, w]$, given that all x_i's are in $(a, U]$, and that at least one x_i is in $(a, b]$;

$\theta_A(a, b, w)$: the probability of no transmission in the interval $(a, w]$, given that all x_i's are in $(a, U]$, and that at least one x_i is in $(a, b]$.

It is obvious that

$$\phi_A(a, b, w) + \theta_A(a, b, w) = 1.0. \tag{10}$$

There are two cases to consider when calculating $\theta_A(a, b, w)$, namely, $b = U$ and $b \neq U$. When $b = U$, it is uncertain whether there is an x_i at b, and all arrangements of the n x_i's must be considered, so

$$\theta_A(a, U, w)|_{b=U} = \frac{(F(U) - F(w))^n}{(F(U) - F(a))^n}. \tag{11a}$$

When $b \neq U$, there must be a station at b, since b is only updated to a value of x_i in the event of a transmission. In this case, we are only concerned with the placement of at most $(n - 1)$ of the x_i's,

$$\theta_A(a, b, w)|_{b=U} = \frac{(F(U) - F(w))^{n-1}}{(F(U) - F(a))^{n-1}}. \tag{11b}$$

The recurrence for choosing the window is

$$N_{A1}(a, b) = \min_{a<w<b} \left\{ 1 + \phi_A(a, b, w) \cdot \left[\int_a^w f_c(a, w, x_b) N_{A1}(a, x_b)\, dx_b \right] + \theta_A(a, b, w) N_{A1}(w, b) \right\}. \tag{12}$$

Again, the three terms on the right-hand side of the above equation count the current broadcast slot, additional broadcast slots in the event of a transmission, and additional broadcast slots if the current broadcast slot is idle.

The assumption that contention can be resolved in one step when the window size is smaller than δ holds, so the following boundary condition is used again.

$$N_{A1}(a, b) = 1 \qquad \text{for all } (b - a) < \delta. \tag{13}$$

The decision tree is the same as for the exact solution, but there is a savings since many of the nodes at different levels are duplicates. The data structure for storing the windows is simply a two-dimensional array. The number of decision points for the approximate solution is determined by the values of a and b. The total number of unique nodes with $\delta = 1/(10n)$ is $((10n)^2 + 30n)/2$, which is determined by counting the decision points indicated by the above recurrences. For $n = 5$ and $n = 6$, the numbers of decisions points are 1325 and 1890, respectively, and the complexity of the solution is considerably reduced from the exact solution.

The performance results of the one-broadcast strategy will be discussed in Section IV-C.

B. Other Strategies

Other strategies were considered to determine if it were possible to improve the performance of the one-broadcast strategy, particularly in reducing the number of iterations after the station with the minimum search parameter has broadcast. The one-broadcast strategy continues until the interval is so small that it is certain that there were no stations with parameters in that interval. An alternative is a *two-broadcast strategy,* which allows up to two broadcast slots per iteration and uses the second slot to determine whether there are any stations with search parameters smaller than the parameter broadcast in the first slot. Initially, as with previous strategies, the intervals is $(L, U]$, and each station has a search parameter x_i in the interval. A global window is determined, and stations with parameters within the window attempt to broadcast their search parameters. If there are no search parameters in the window, the strategy proceeds as in the one-broadcast strategy: the lower bound of the interval is updated to the window's upper bound, and the protocol continues. The difference between the one- and two-broadcast strategies occurs when there are parameters within the window. In this case, the upper bound is updated to x_{b1}, the value broadcast, and a second broadcast slot is allowed for all stations with $x_i < x_{b1}$. If the second slot is idle, x_{b1} is the minimum, and the algorithm terminates. If there is a broadcast, the next iteration begins with x_{b2}, the second value broadcast, as the upper bound of the interval. Note that x_{b2} is smaller than x_{b1}.

A problem in the two-broadcast strategy is that the window for the second broadcast slot is chosen suboptimally when there are stations with search parameters smaller than the current broadcast value. Therefore, a better solution is a *combined strategy,* which combines the one-broadcast and two-broadcast strategies and makes a decision in each iteration whether one broadcast or two broadcasts will be used. Again, the objective is to minimize the expected number of future broadcasts.

Analyses and complexities of the two-broadcast and the combined strategies are similar to those of the one-broadcast strategy. They are not presented here because of space limitation and of their inferior performance as compared to the one-broadcast strategy. Interested readers can find them in [1] and [2].

C. Simulation Results for the Approximate Distributed Searches

The simulation results for the distributed window search using the three strategies are shown in Fig. 5. The windows were generated using the equations derived in Sections IV-A and in [1] and [2]. The broadcast parameters were generated from a uniform distribution in (0, 1], and sufficient cases were simulated until a confidence interval of 0.95 was reached. The average number of broadcast slots is bounded by 2.7 for the two-broadcast strategy, and by 2.6 for the one-broadcast and combined strategies. The two-broadcast strategy is not as good as the one-broadcast strategy because, although it can reduce the number of broadcast slots after the minimum has been identified, it uses suboptimal window choices for earlier broadcast slots. The combined strategy always chooses the

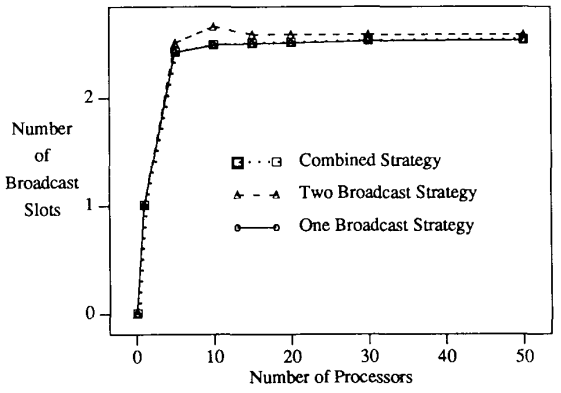


Fig. 5. Simulations results for different window-search strategies.

one-broadcast strategy, so their results are identical. The overhead for the combined strategy is higher than that of the one-broadcast strategy because the strategy for each decision point must be stored in addition to the window choices. For these reasons, the one-broadcast strategy is superior to the other two.

Note that the broadcast parameters are assumed to be independent and uniformly distributed in $(0, U]$ in the simulations. In case that the distribution function is nonuniform but independent and identical for all broadcast parameters, a uniformly distributed broadcast parameter can be obtained from the original broadcast parameter by the following formula:

$$z = F^{-1}(x) \tag{14}$$

where x is the original broadcast parameter with distribution F, and z is the new broadcast parameter with a uniform distribution.

The proposed scheme is practical as a result of the constant expected number of broadcast slots. The time required for a contention slot is approximately 50 μs, and the time required to broadcast a search parameter may be estimated at approximately 100 μs. It follows that each broadcast slot would require on the order of 220 μs if 2.4 contention slots [22] were required to resolve contention. If it takes 120 μs to resolve contention and 100 μs to transmit a one-kbyte packet, then the overhead of each load balancing decision to identify the maximally and the minimally loaded processors is equivalent to transmitting 5.2 one-kbyte packets.

D. Distribution of Load Averages

The knowledge on the distribution of workload (or load averages) is needed in the distribution search in order to choose the windows. In this section, we present statistics of load averages on a system experiencing a real workload. The study consisted of measuring the load on a system of ten Sun workstations (servers and clients). Every sixty seconds, the one-minute load average was measured and logged. The load data were analyzed using an adjusted Komolgorov–Smirnov test [14], [20]. This goodness-of-fit test can be used to detect differences between a normal distribution and the empirical distribution indicated by the measured data. The agreement was measured over time. The results indicate that the

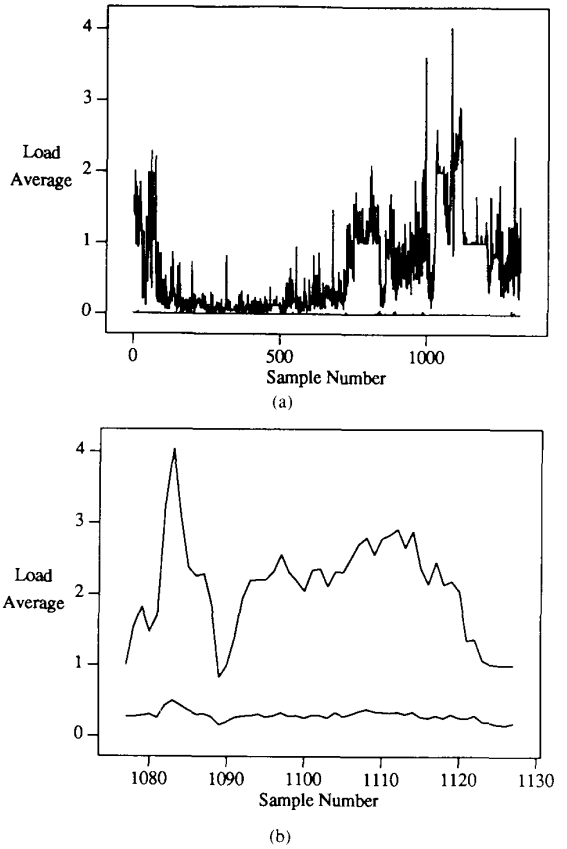


Fig. 6. Load maximums and minimums. (a) Load maxima and minima for one day. (b) Load maxima and minima for one hour.

distribution of load averages can be estimated using a normal distribution. During 80 percent of the time when the system is active, the distribution is within 0.215 of a normal distribution.

The results of the study are shown in Fig. 6. Fig. 6(a) shows the maximum and minimum load averages over time. The minimum is almost always zero and the maximum varies. The peak utilization is between sample number 700 and 1400 which reflects the load from 1 p.m. until midnight. From 1 a.m. until 11 a.m., the loads measured were uniformly low (≤ 1). Fig. 6(b) shows one hour of the minimum, average, and maximum load averages.

V. Implementation

The Sun system on which GAMMON has been implemented consists of servers and clients connected by Ethernets. The servers have secondary disk storage, and the clients do not. A client can access a server's disk via the network, and is allocated a portion of the server's disk for swap space. Swapping over the network is a part of the regular message transfer. A network file system (NFS) allows transparent access to remote file systems. This mechanism allows uniform access by the clients to the secondary storage.

The Sun system is multiprogrammed; when a process is initiated on a server or a client, a core image of that process containing run-time information exists in the swap space

TABLE I
RESULTS OF SIMULATION FOR THE TRUNCATED SEARCH (RANK OF THE MINIMUM IS 0)

Number of processors	% of searches unresolved	% resolved searches finding minimum	average rank of result	% difference of result from minimum
3	20.0	78.00	0.194	2.790
4	18.6	76.04	0.224	2.192
5	20.1	73.97	0.268	2.086
6	21.8	73.66	0.252	1.665
7	21.4	72.26	0.279	1.506
8	20.5	70.57	0.292	1.495
9	23.1	72.82	0.256	1.151
10	21.8	74.04	0.256	0.959

associated with that server or client. This core image may be quite large (on the order of several megabytes). Since it is not possible to copy data from one swap space to another, the only way to transfer the core image is over the network. Due to the size of the core image, this transfer would have high overhead (on the order of seconds); consequently, the current implementation migrates jobs only at their entry point. Future enhancements to GAMMON will study preemption strategies.

There are two program modules (daemons) that comprise GAMMON in the current implementation: the searcher and the job migrator. The search daemon periodically participates in a search for the minimum. Job migration is performed by sending the necessary information of a job at entry point to a remote processor; result return is performed when execution of the migrated job is completed. Priorities of the various phases in load balancing, as discussed in Section II, are not enforced because the overhead of explicit priority resolution is very high and is not compensated by the resulting reduction in response time.

A. Implementation of the Distributed Search

The search for the minimum load normally takes 2.6 broadcast slots on the average (see Section III). Table I shows the simulation results of the search truncated at one broadcast slot. This table shows that, under this condition, the absolute minimum is located 70–78 percent of the time and there are 18.6–23.1 percent of the unresolved cases in which all search parameters are in the interval $(w_l, U]$ and no workload information is broadcast. Unresolved searches are not critical here because they reflect a condition in which no processors are lightly loaded enough to accept additional jobs. For cases that are resolved, those that do not find the absolute minimum have a one to two percent difference from the absolute minimum. Since workload information is heuristic in nature, small errors in identifying processors for load balancing are not critical. An important tradeoff we have achieved is that reasonable results are obtained at substantially lower overhead.

The minimum search as implemented on the Sun network is shown in Fig. 7. Execution is initiated, as mentioned above, by an alarm signal, or by a packet arriving from another search module. When an alarm is received, the processor's current load is compared to the lower bound of the window. If the load is smaller than the lower bound, it is broadcast with a time stamp and processor address. If a packet is received, the load is accepted as the minimum and is stored with the current time. If more than one processor sends a load packet due to their both receiving an alarm simultaneously, the minimum load is accepted. If the loads are identical, the processor address is used as a tie breaker. The alarm is set when packets are received, so the process is loosely synchronized. The search daemon obtains load information (in the current implementation the load average) directly from the kernel.

The contents of received packets are written to a file to reflect the current status of the network. As the status (the minimum load, and the location of the processor with the minimum load) is stored, the current time is also stored. When the status information is read from the file, its time stamp is used to determine if the load value is out of date. If the status information is out of date, the processor considers itself ineligible for job migration until it receives another status packet. This time stamping allows an unreliable communication mechanism to be used to communicate load information (broadcast datagrams) and reduces network traffic in the form of acknowledgments. If load information is lost by a processor, that processor does not migrate packets. This has a minimal affect on the overall performance of the scheduling strategy. Moreover, since packet loss is a relatively rare occurrence, acknowledgments are unnecessary.

The search was tested on two Sun systems: one with a server (Aquinas) and two clients (Calvin and Hobbes), and one with a server (Dwarfs) and ten clients. On Aquinas, the search was resolved in 50–80 ms, and on Dwarfs, the searches were resolved in 150–180 ms. The performance on Dwarfs can be considered the worst case because the Ethernet cable the clients are connected with is at the allowable length limit. The results for both Aquinas and Dwarfs were consistent with the simulation results in which the absolute minimum was located 70 percent of the time for resolved searches.

The search for the maximum load is not performed explicitly. It was observed that if a processor's load was above the upper bound of the initial window in the minimum search, its load was the maximum the majority of the time. As a result, processors with loads above the upper bound of the initial

```
/* INTERVAL: constant indicating time interval at which search is performed
 * packet{   information contained in the packets exchanged
 *  load:    processor load
 *  source:  processor address
 * }
 * load_status { information contained in load status
 *  timestamp:          timestamp of the load status
 *  minimum_load:       load at the minimally loaded processor
 *  minimum_location:   location of the minimally loaded processor
 * }
 * current_time():   function returning the current time
 * current_load():   function returning the current load at a processor
 * set_alarm():      function to set an alarm signal ARGUMENT seconds from now
 * on_alarm():       function that sets a call to the argument when an alarm occurs
 * reset_alarm():    function to reset the alarm signal to ARGUMENT seconds from now
 * wait_arrival():   function that waits for the arrival of a packet
 * write():          writes to a file
 */

structure load_status load_status;    /* global structure containing load status */

procedure main;
    on_alarm(send_packet());
    set_alarm(INTERVAL);
    while (TRUE)
        receive_packet();
    endwhile
end

procedure send_packet;
    if (current_load() < window) then
        packet.load = current_load();
        packet.source = MY_ADDRESS;
        broadcast(packet);
    endif
    set_alarm(INTERVAL);
end

procedure receive_packet();
    wait_arrival(packet);
    reset_alarm(INTERVAL);
    if (current_time() > load_status.time) then
        load_status.minimum_load = packet_load;
        load_status.minimum_location = packet_source;
        load_status.timestamp = current_time();
        write(load_status);
    endif
end
```

Fig. 7. Procedures for the minimum search.

window can migrate the current job if the minimum load is current, and an explicit search of the maximally loaded processor is not needed.

A potential problem with not explicitly identifying the maximally loaded processor is that a lightly loaded processor may be swamped by jobs from more heavily loaded ones. There are two solutions to resolve this problem. First, a processor may only be allowed to migrate jobs if it has a load higher than the upper bound of the initial window and a new arrival. Second, a processor may be allowed to migrate at most one job between searches. Our performance data indicate that swamping is not a problem for a moderate number of participating processors. However, if preemption were implemented, it would be necessary to identify a unique maximum, as a large fraction of the processors may be preempting jobs at any time.

Another obvious technique for distributing status information is to broadcast it periodically, as is done with the ***rwhod*** daemon in Unix. To determine the savings of using the proposed method as opposed to using the technique of the ***rwhod*** daemon, the resource utilization of both was measured for the period of one hour. The overhead is summarized in Table II. For comparison, a daemon that only broadcasts the load average value is studied (as opposed to the ***rwho*** daemon which broadcasts other information as well). Recall that the daemons perform a search every minute. When a search using GAMMON is performed, it is possible for daemons at two computers to start a broadcast simultaneously; hence, the total number of broadcast is slightly above 60 (first value in the last column of Table II). When the broadcast or ***rwhod*** daemons are used, each computer initiates a broadcast every minute independent of other computers.

TABLE II
COMPARISON OF OVERHEAD FOR STATUS DISTRIBUTION OF THREE COMPUTERS IN ONE HOUR OF OPERATION

Deamon	Cumulative System Time in Seconds	Cumulative User Time in Seconds	Total Number of Broadcasts
search	0.1094	0.0137	65
broadcast	0.2593	0.0316	181
rwho	0.2633	0.0613	181

TABLE III
PERFORMANCE OF THE GLOBAL SCHEDULING STRATEGY

Name	Job Time w/out Global Scheduling	Job Time with Global Scheduling	% Difference
Aquinas (Sun 3/260 server)	18963	18540	2.23
Calvin (Sun 3/50 client)	12143	10430	14.10
Hobbes (Sun 3/50 client)	7319	5912	19.22
Totals	38425	34882	9.2

The results in Table II indicate that both the ***rwhod*** and the simple broadcast daemon introduce considerably more communication overhead than the search daemon proposed here.

B. Implementation of Job Migration and Result Return

In the general case, result return and job migration require that input file (and executable files if necessary) be sent over the network from the source processor to the destination process, and that output files be sent back. In the Sun environment, it is not necessary to send files from the source to the destination because all processors and clients have access to the same secondary storage. In this environment, job migration requires sending the command from the source to the destination, and result return requires sending any error information back.

In the procedure for job migration, the current load of the processor is first compared to the upper bound of the initial window for the maximum search. If the load is above this upper bound, then the global minimum location and its time stamp are read from a local file. If the timestamp indicates the minimum is current, the job is executed on the processor with the minimum load using a remote shell. This causes any error messages associated with the remote execution to be send back directly. The remote shell uses the reliable TCP/IP protocol, so execution of the job is guaranteed. Further details of this implementation can be found in [3].

C. Performance of GAMMON

We evaluate GAMMON by comparing the response time of a system with GAMMON to that of a system without global scheduling. The server, Aquinas, with two clients, Calvin and Hobbes, are used. First, workload was generated by processes that either initiated a CPU-bound job or slept for the amount of time the job consumed during its last execution with equal probability. As these processes were executed, the load and the amount of time consumed by executing processes was tabulated. Also, a history of the initiation of jobs was created. Next, the global scheduling strategy was enabled and the jobs were initiated according to this history. Again the load was monitored and the execution time tabulated.

The results summarized in Table III show a small improvement for the server and a much larger improvement for the clients. This is expected as the server is about 2.7 times faster than the clients and has direct access to the secondary storage through the VME bus rather than the Ethernet. As a result, the clients will benefit more by sending jobs to the server.

VI. CONCLUDING REMARKS

In this paper, we have presented an efficient technique for a distributed extremum search and a load balancing protocol using this technique. The search technique can be implemented at the applications level on *existing* distributed computing systems connected by multiaccess networks. This is important because it is typically not possible to make hardware modifications to existing networks. The maximum or the minimum of a set of numbers, which reflect the workloads, can be identified in a small bounded number of broadcast slots on the average. Since the search technique has a constant average behavior, the GAMMON strategy using this efficient search technique is feasible. GAMMON was implemented on a network of Sun workstations. Performance measurements indicate that the system with GAMMON allows improvement in overall performance as well as improved individual processor performance.

ACKNOWLEDGMENT

We gratefully acknowledge the help of R. M. Kling, who has participated in developing the implementation on the network of Sun computers.

REFERENCES

[1] K. M. Baumgartner and B. W. Wah, "Load balancing protocols on a local computer system with a multiaccess bus," in *Proc. Int. Conf. Parallel Processing.* University Park, PA: Pennsylvania State University Press, Aug. 1987, pp. 851–858.

[2] K. M. Baumgartner, "Resource allocation on distributed computer systems," Ph.D. dissertation, School of Electrical Engineering, Purdue University, West Lafayette, IN, May, 1988.

[3] K. M. Baumgartner, R. M. Kling, and B. W. Wah, "Design and implementation of an efficient load balancing strategy for a local computer system," in *Proc. Int. Conf. Parallel Processing.* University Park, PA: Pennsylvania State University Press, 1989.

[4] T. C. K. Chou and J. A. Abraham, "Load Balancing in distributed systems," *IEEE Trans. Software Eng.*, vol. SE-8, pp. 401–412, July 1982.

[5] Y. C. Chow and W. Kohler, "Models for dynamic load balancing in a heterogeneous multiple processor system," *IEEE Trans. Comput.*, vol. C-28, pp. 334–361, May 1979.

[6] W. W. Chu, L. J. Holloway, M. T. Lan, and K. Efe, "Task allocation in distributed data processing," *IEEE Computer,* pp. 57–68, Nov. 1980.

[7] D. L. Eager, E. D. Lazowska, and J. Zahorjan, "Adaptive load sharing in homogeneous distributed systems," *IEEE Trans. Software Eng.*, vol. SE-12, pp. 662–675, May 1986.

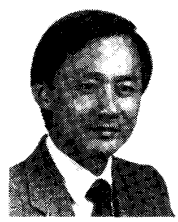
[8] D. Ferrari and S. Zhou, "A load index for dynamic load balancing," in *Proc. Fall Joint Comput. Conf.*, Nov. 1986, pp. 684–690.

[9] 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, Apr. 1982. Also in *Tutorial: Computer Architecture,* D. D. Gajski, V. M.

Milutinovic, H. J. Siegel, and B. P. Furht, Eds. New York: IEEE Computer Society, 1987, pp. 541-552.
[10] J. Y. Juang and B. W. Wah, "Unified window protocols for contention resolution in local multiaccess networks," in *Proc. INFOCOM,* Apr. 1984, pp. 97-104.
[11] — —, "Optimal scheduling algorithms for multistage resource sharing interconnection networks," in *Proc. Comput. Software Appl. Conf.,* Nov. 1984, pp. 217-225.
[12] — —, "Global state identification for load balancing in a computer system with multiple contention busses," in *Proc. Comput. Software Appl. Conf.,* Oct. 1986, pp. 36-42.
[13] A. Kratzer and D. Hammerstrom, "A study of load leveling," in *Proc. COMPCON,* Fall 1980, pp. 647-654.
[14] A. M. Law and D. W. Kelton, *Simulation Modeling and Analysis.* New York: McGraw-Hill, 1982.
[15] M. Livney and M. Melman, "Load balancing in homogeneous broadcast distributed systems," in *Proc. Modeling Perform. Eval. Comput. Syst.,* ACM SIGMETRICS, 1982, pp. 47-55.
[16] R. Metcalfe and D. Boggs, "Ethernet: Distributed packet switching for local computer networks," *Commun. ACM,* vol. 19, no. 7, pp. 395-404, 1976.
[17] L. M. Ni and K. Hwang, "Optimal load balancing strategies for a multiple processor system," in *Proc. 10th Int. Conf. Parallel Processing,* Aug. 1981, pp. 352-357.
[18] A. S. Tanenbaum, *Computer Networks.* Englewood Cliffs, NJ: Prentice-Hall, 1981.
[19] A. N. Tantawi and D. F. Towsley, "Optimal static load balancing in distributed computer systems," *J. ACM,* vol. 32, pp. 445-465, Apr. 1985.
[20] K. S. Trivedi, *Probability and Statistics with Reliability, Queuing, and Computer Science Applications.* Englewood Cliffs, NJ: Prentice-Hall, 1982.
[21] B. W. Wah and J. Y. Juang, An efficient protocol for load balancing on CSMA/CD networks," in *Proc. 8th Conf. Local Comput. Networks,* Oct. 1983, pp. 55-61.
[22] — —, "Resource scheduling for local computer systems with a multiaccess network," *IEEE Trans. Comput.,* vol. C-34, pp. 1144-1157, Dec. 1985.
[23] ——, "An efficient contention resolution protocol for local multiaccess networks," U.S. Patent 4630264, Filed Sept. 21, 1984, Granted Dec. 16, 1986.
[24] B. W. Wah and P. Mehra, "Learning parallel search in load balancing," in *Proc. Workshop Parallel Algorithms Machine Intel. Pattern Recognition,* AAAI, Minneapolis, MN, Aug. 21, 1988.
[25] Y. T. Wang and J. T. Morris, "Load sharing in distributed systems," *IEEE Trans. Computers,* vol. C-34, pp. 204-217, Mar. 1985.

Katherine M. Baumgartner (S'81-M'89) received the B.S.E.E., M.S.E.E., and Ph.D. in electrical engineering all from Purdue University, West Lafayette, IN, in 1981, 1984, and 1988 respectively.

From 1986 to 1988 she was a Research Assistant at the Coordinated Sciences Laboratory at the University of Illinois, Urbana. Her research interests include architectures for parallel and distributed computer systems and distributed operating systems. She is currently a Senior Engineer at the Digital Equipment Corporation in Maynard, MA.



Benjamin W. Wah (S'74-M'79-SM'85) received the Ph.D. degree in computer science from the University of California, Berkeley, CA, in 1979.

He was on the faculty of the School of Electrical Engineering at Purdue University, West Lafayette, IN, between 1979 and 1985. He is now a Professor in the Department of Electrical and Computer Engineering and the Coordinated Science Laboratory, University of Illinois at Urbana-Champaign, Urbana. Between 1988 and 1989, he served as a program director of the Microelectronic Systems Architecture Program, National Science Foundation. His areas of research include computer architecture, parallel processing, artificial intelligence, distributed databases, and computer networks.

Dr. Wah is an Associate Editor-in-Chief of the IEEE TRANSACTIONS ON KNOWLEDGE AND DATA ENGINEERING, an area editor of the *Journal of Parallel and Distributed Computing,* and an editor of *Information Sciences.* He serves as a member of the Governing Board of the IEEE Computer Society and a program evaluator for ABET (computer Engineering) and CSAC (computer science).