TABLE III
PERFORMANCE COMPARISON OF DIFFERENT STRIDES IN $M_2(256, 256)$

Performance measure	Completion time	Processor utilization	External fragment	NC†	NC††
$FS_{1,1}$	89830.3	47.7	33.7	16958.7	56937.3
$FS_{\text{half},\text{half}}$	9716.8	44.1	34.0	44.4	38.8
FS	9922.1	43.2	33.5	16.4	13.7

† Number of comparisons per successful attempt.
†† Number of comparisons per attempt.

and height, listed in row $FS_{\text{half},\text{half}}$ of Table III, where both side-length and residence time distributions are uniform. In the extreme case, when both horizontal and vertical strides equal 1 (the results are shown in row $FS_{1,1}$), time complexity becomes excessively high, but performance does not improve significantly. It is thus concluded that making strides equal to the side lengths of a requested submesh appears to be a good choice.

V. CONCLUSION

In this short note, we have introduced an efficient mesh processor allocation strategy based on frame sliding, termed the FS strategy, which can be applied to any mesh system and recognizes submeshes with arbitrary sizes at any location. The FS strategy allocates a submesh of the precise size to an incoming task, completely eliminating internal fragmentation. As a result, when compared with an earlier allocation strategy based on the buddy principle, this strategy leads to far better processor use and substantially reduces total time spent in finishing a batch of tasks. An effective implementation of the strategy that greatly simplifies the search of candidate frames for an available submesh is also presented. Simulation results confirm that our proposed strategy is consistently superior in allocating submeshes whose side lengths follow any of the four distributions, in terms of performance measures of interest. The mean search time per allocation attempt is fairly short, involving at most tens of simple comparisons. The FS strategy tends to have a larger search space than the buddy strategy, but its time complexity is kept low. It is observed from simulation that the CPU time spent in allocating 1000 tasks is less following the FS strategy than it is following the buddy strategy, when both strategies have the same search space. The FS strategy involves no higher complexity for a larger mesh system and is thus efficiently applicable to a system of any size. The performance of the FS strategy under different stride values is also pursued by simulation. Shortening strides gives rise to more candidate frames being examined and incurs higher time complexity. In the extreme case, where both horizontal and vertical strides are equal to 1, time complexity becomes excessively high, but performance improves insignificantly. It appears that making strides the same as side lengths of a requested submesh is a good choice.

We also have explored (but have not included in early sections, because of the space limitation) two approaches for reducing external fragmentation of our strategy: one by limiting the number of allowable submesh sizes and the other by using compaction [8], [9]. It is observed from simulation that our strategy is found to behave virtually the same as the buddy strategy if its allowable submesh sizes are limited to those permitted by the buddy strategy. As for compaction, not to mention its large migration overhead, it contributes no performance improvement, perhaps because of the essential nature of precise submesh allocation: reduced fragmentation, which is not further reducible. Compaction is thus unnecessary for our allocation strategy. We believe that the FS strategy is readily useful for any mesh connected system.

REFERENCES

[1] T. J. Fountain, K. N. Matthews, and M. J. B. Duff, "The CLIP7A image processor," *IEEE Trans. Pattern Anal. Machine Intell.*, vol. 10, pp. 310–319, May 1988.
[2] R. Alverson, *et al.*, "The Tera computer system," in *Proc. 1990 Int. Conf. on Supercomputing*, 1990, pp. 1–6.
[3] G. Zorpette, "Technology 1991: Minis and mainframes," *IEEE Spectrum*, pp. 40–43, Jan. 1991.
[4] R. W. Hockney and C. R. Jesshope, *Parallel Computers: Architecture, Programming and Algorithms.* Bristol, UK: Adam Hilger, 1981.
[5] K. Li and K.-H. Cheng, "A two-dimensional buddy system for dynamic resource allocation in a partitionable mesh connected system," *Proc. ACM Comput. Sci. Conf.*, Feb. 1990, pp. 22–28; also *J. Parallel Distrib. Computing*, vol. 12, pp. 79–83, May 1991.
[6] J. L. Peterson and T. A. Norman, "Buddy systems," *Commun. ACM*, vol. 20, no. 6, pp. 421–431, June 1977.
[7] M. Livingston and Q. F. Stout, "Parallel allocation algorithms for hypercubes and meshes," in *Proc. 4th Conf. Hypercube Concurrent Comput. Applications*, 1989, pp. 59–66.
[8] M.-S. Chen and K. G. Shin, "Task migration in hypercube multiprocessors," in *Proc. 16th Annu. Int. Symp. Comput. Architecture*, 1989, pp. 105–111.
[9] C.-H. Huang and J.-Y. Juang, "A partial compaction scheme for processor allocation in hypercube multiprocessors," in *Proc. 1990 Int. Conf. Parallel Processing*, vol. I, 1990, pp. 211–217.
[10] W. J. Dally and C. L. Seitz, "The Torus routing chip," *Distrib. Computing*, vol. 1, no. 3, pp. 187–196, 1986.

Performance Evaluation of an Efficient Multiple Copy Update Algorithm

T. V. Lakshman and Dipak Ghosal

***Abstract*— A well-known algorithm for updating multiple copies is the Thomas majority consensus algorithm. This algorithm, before performing an update, needs to obtain permission from a majority of the nodes in the system. In this short note, we study the response-time behavior of a symmetric (each node seeks permission from the same number of other nodes and each node receives requests for update permission from the same number of other nodes) distributed update-synchronization algorithm where nodes need to obtain permission from only $O(\sqrt{N})(N$ being the number of database copies) other nodes before performing an update. The algorithm we use is an adaptation of Maekawa's $O(\sqrt{N})$ distributed mutual exclusion algorithm to multiple-copy update-synchronization. This increase in the efficiency of the update-synchronization algorithm enhances performance in two ways. First, the reduction in transaction service time reduces the response time. Second, for a given arrival rate of transactions, the decrease in response time reduces the number of waiting transactions in the system. This reduces the probability of conflict between transactions. To capture the interaction between the probability of conflict and the transaction response time, we define a new measure called the *conflict response-time* product. Based on the solution of a queueing model we show that optimizing this measure yields a different and more appropriate choice of system parameters than simply minimizing the mean transaction response time.**

***Index Terms*—Transaction processing, concurrency control, replicated data, multiple copy update, transaction response time, conflict probability, transaction throughput, diameter-two sparse-graphs, finite projective planes.**

Manuscript received September 4, 1991; revised March 17, 1992.
The authors are with Bell Communications Research, 331 Newman Springs Road, Red Bank, NJ 07701.
IEEE Log Number 9214472.

I. Introduction

To improve performance and availability, it is often necessary to replicate data in a distributed database system. When data is replicated, mutual consistency of this replicated data must be maintained. Typically, to ensure correctness, the replicated-data concurrency-control algorithm must ensure that the concurrent execution of transactions on replicated data is equivalent to some serial execution of the same transactions on non-replicated data. Correctness requires that the data items read by a transaction are not made obsolete by updates from other conflicting transactions. When the probability of conflicts is low, one useful method of ensuring correctness is to initially assume that transactions will be conflict-free and hence, not lock any data items which are read. Then, when updates are to be performed, an update synchronization algorithm is used to prevent two conflicting transactions from making non-serializable updates. Increases in the efficiency of the update-synchronization algorithm enhance performance in two ways. First, a more efficient algorithm reduces the transaction service time and hence, the response time. Second, for a given arrival rate of transactions, the decrease in response time reduces the number of waiting transactions in the system. This reduces the probability of conflict between transactions. This reduction in the probability of conflict reduces the number of transaction aborts (and retries) and hence, reduces the user-perceived response time. Most previous studies of efficient update-synchronization algorithms have focussed on complexity analysis and have not considered the impact of increased efficiency on important user-perceived system parameters such as the response time. A simulation study of response time behavior of an efficient multiple-copy update algorithm is in [2]. In this short note, using a queueing model, we evaluate the system-performance effect of an efficient update-synchronization algorithm. As a paradigmatic example, we use an adaptation of Maekawa's $O(\sqrt{N})$ distributed mutual exclusion algorithm [14] to multiple-copy update-synchronization.

A well-known algorithm for updating multiple copies is the Thomas majority consensus algorithm [20]. In Thomas' scheme, a transaction reads a set of data items (called the transaction's base variables) from its local database copy without regard to any other concurrent access of the same base variables. However, before performing any updates the transaction has to get permission for the updates from a majority of the system nodes in which the database is stored. Once this permission is obtained the update is performed on all copies of the database. Nodes use an elaborate timestamp based method in deciding whether to grant, reject, or abstain from voting on update requests.

In the Thomas' majority consensus algorithm the only property of the majority sets used to prove correctness of the method is that any two majority sets have a nonnull intersection. An alternative to majority consensus is a scheme that divides the system *a priori* into a family of intersecting sets such that no two sets are disjoint. Before performing updates, a transaction must obtain permission for the updates from all nodes in any one set. A conflicting transaction must also similarly obtain permission from all nodes in any one set. Since no two sets are disjoint, there is at least one node which gets update requests from both the transactions and this node decides which of the two conflicting transactions can perform updates.[1]

Note that the choice of sets determines the degree of distributedness of the update synchronization algorithm. One criteria determining the choice of sets is the impact of failures on the reliability of the system. The impact of failures on vote assignments (for weighted voting schemes as in [6]) and choice of sets is extensively studied in [5].

A second criteria in choosing the family of sets is minimizing the transaction response time. In this short note, we only consider homogeneous systems with equal communication delays between all pairs of nodes. In this case, the cardinality of the sets (as opposed to the actual members of different sets) determines the impact of the update synchronization algorithm on the response time and to reduce response times we would like to choose sets with the least possible cardinalities. Also, we would like all sets to be of the same cardinality. This is because response times can be reduced by effective load-sharing and designing load sharing policies is simpler for homogeneous systems. If we start with a homogeneous distributed system, then this homogeneity can be maintained by choosing sets of the same cardinality. Choosing sets of different cardinalities would make the system heterogeneous since the transaction service times and update request arrival rates at different nodes would now be different.

A third criteria in choosing sets is that all nodes in the system should have a symmetric role in the sense that if requests are uniformly distributed over the system then each node is used equally often as a decision making node. This requires that all nodes belong to the same number of sets.

The problem of constructing a family of sets with good performance characteristics is equivalent to constructing a sparse regular graph of diameter 2. Each system node corresponds to a vertex in this graph and each set corresponds to a vertex and all its neighbors. Since the degree of the graph determines the cardinality of the sets (and hence, the number of nodes from which permission is sought), it is necessary (for a given N) to construct graphs with minimum degree. The Moore bound [8] gives an upper bound on the number of vertices N (or equivalently a lower bound on the degree when N is fixed) in a regular graph of degree r and diameter d. The bound is : $N \leq 1+r+\ldots+r(r-1)^{d-1}$. This implies that r is $\Omega(N^{1/D})$ and for diameter 2 graphs the lower bound on the degree is $\Omega(\sqrt{N})$. Since the degree of the graph determines the size of the sets, $\Omega(\sqrt{N})$ is a lower bound on the number of nodes from which update permission must be obtained. For efficient algorithms, we need graphs with r as close to the Moore bound as possible. Unfortunately, the Moore bound itself is known to be unachievable except in a few cases.

The update-synchronization algorithm we use in this short note, uses finite projective planes as the family of sets. This family of sets consists of N (the number of nodes in the system) sets. Every pair of sets has exactly one intersection and every pair of nodes is contained in exactly one set. All sets are of the same cardinality $O(\sqrt{N})$, and all nodes belong to the same number of sets. Sets with these properties are useful in many other distributed algorithms such as commit protocols [10], computing functions of distributed information [11], [12], [3], and distributed matchmaking [15]. Section II describes finite projective planes and points out properties of projective planes relevant to this short note. Section III describes an update-synchronization algorithm based on finite projective planes. Note that the majority sets in Thomas' algorithm are of size $O(N)$. Since our set-based scheme uses sets of size $O(\sqrt{N})$, a transaction before performing an update needs to obtain permission from only $O(\sqrt{N})$ other nodes. This reduction in both communication and processing should drastically improve performance. To determine the impact on performance, we develop a queueing model for distributed database systems using the proposed algorithm. The model is developed in a hierarchical manner with two levels. At the lower level, we model each database node as a single server queueing system serving two classes of customers. Class 1 jobs correspond to the permission requests arriving from the various database nodes

[1]The rule used to make the choice between conflicting updates is discussed in Section III. Note that any single-copy concurrency control algorithm can be used to make this decision.

and class 2 jobs correspond to the external arrival of update requests. Class 1 jobs have preemptive priority over class 2 jobs. The lower level model is solved to obtain the response time of each class of jobs. At the higher level, we derive expressions for the transaction response times taking into account the conflict probabilities which are based on the lower level model. Since some of the parameters of the lower level model depend on the higher level measures, an iterative technique is used for the solution of the entire model. The model, the assumptions, and the analysis are described in Section IV. There are a number of performance issues related to the algorithm. One interesting issue relates to the interaction between the degree of conflict and the mean transaction response time. Increase in the transaction response time increases the degree of conflict which in turn reduces the transaction response time since transactions are quickly aborted. To investigate this interaction we define a new measure called the *conflict response-time* product. Using the queueing theoretic model we show that minimizing the above measure results in a different and more appropriate choice of system configuration than that predicted by just minimizing the mean transaction response time. These and other results are discussed in Section V. Concluding remarks are in Section VI.

II. Finite Projective planes

A finite projective plane consists of a finite collection of points and lines (lines are sets of points) which satisfy the following postulates:

1) Two distinct points lie on one and only one common line.
2) Two distinct lines pass through one and only one common point.
3) There are four distinct points, no three of which lie on the same line.

Postulate P3 is needed to eliminate certain degenerate finite projective planes such as a set of points and a single line. By P3, the smallest possible projective plane would have at least four points. (Construction of finite projective planes is discussed in [16].)

The following theorems on finite projective planes [1] are necessary to establish properties of the algorithm to be discussed.

Theorem 2.1: In a finite projective plane, every point lies on the same number of lines, and every line passes through the same number of points.

Theorem 2.2: In a finite projective plane, the number of lines through each point is the same as the number of points on each line.

Theorem 2.3: A projective plane with $m + 1$ points on each line and $m + 1$ lines through each point has $m^2 + m + 1$ points and $m^2 + m + 1$ lines.

The number m in Theorem 2.3 is called the *order* of the finite projective plane.

Theorem 2.4: If $m = p^k$ for p prime and k a positive integer then there is a projective plane of order m.

III. Multiple-Copy Update Synchronization Algorithm

There are four phases in the execution of a transaction that are relevant to this short note: 1) the read phase when the transaction reads the base variables used in the transaction's computation; 2) the computation phase when the transaction's updates are computed; 3) the update-synchronization phase when the update-synchronization algorithm described below is used and 4) the update phase when the actual update is performed.

As in [20], the update-synchronization algorithm can be described using five rules: a) the communication rule, b) the timestamp generation rule, c) the voting rule, d) the request resolution rule, and e) the update application rule.

The Request Resolution and Communication Rules: These rules specify the manner in which the nodes in the distributed database system interact to obtain update permissions. In our algorithm, each node in the system is considered to be a point[2] in a finite projective plane. When a transaction wants to perform an update, the node generating the update request for the transaction arbitrarily chooses a line in the projective plane passing through this node and tries to obtain permission for the update from all nodes in this line. This may be done either in a daisy-chained manner (wait for one node's reply before seeking permission from another node in the line) or by a multicast to all the other m points in the chosen line. The line can be chosen based on implementation considerations. For example, if the communication costs are not uniform then the line can be chosen such that the communication cost is minimum. This choice can be static or can be dynamic based on load balancing schemes.

Each node receiving an update request applies the voting rule and either accepts the transaction by voting OK or rejects the transaction by voting REJ. A node can perform updates only if it receives OK votes from all nodes in a line. To commit updates, the node receiving permission for the updates sends commit signals to the participating nodes. If the node seeking permission for updates has to abort the update (due to receipt of a REJ vote), it must still inform the nodes from which it sought permission by sending abort signals.

To see why this method preserves mutual consistency, consider two conflicting transactions T_i and T_j originating at nodes i and j. Let the update requests of T_i and T_j choose lines L_i and L_j as the lines from which to seek permission for their updates. Since in a projective plane every two lines intersect in exactly one point, the point at which L_i and L_j intersect gets both T_i's and T_j's update requests. This node applies the voting rule to accept one of the transactions and reject the other.

It is easy to see from the properties of finite projective planes that each node communicates with only $O(\sqrt{N})$ other nodes. This is true even when a projective plane does not exist for the given N. This can be shown as follows: Let $m_0 = \left\lfloor \frac{-1+\sqrt{4N-3}}{2} \right\rfloor$ and let m_1 be the smallest integer greater than m_0 which is the power of a prime. It is known [7] that for any $n \geq 1$, there is at least one prime p such that $n < p \leq 2n$. Therefore, $m_0 < m_1 \leq 2m_0$, implying that m_1 is $O(\sqrt{N})$. By Theorem 2.4, a finite projective plane can be constructed with $N^* = {m_1}^2 + m_1 + 1$ points. Since $N^* > N$, $N^* - N$ virtual nodes have to be added to the system for the algorithm to be used. Since m_1 is $O(\sqrt{N})$ the addition of virtual nodes does not increase the complexity. However, these virtual nodes would have to act like the real system nodes. One method of emulating these virtual nodes is to route all requests addressed to virtual nodes to real nodes.

The Timestamp Rule: The timestamp rule is the same as that used in Thomas' scheme. Timestamps are used as sequence numbers to assign priorities to events. A larger timestamp implies a lower priority. Every transaction entering the system is assigned a timestamp and also stored with every data item is the timestamp of the transaction which last updated the data item. These timestamps are used both in the voting rule (to check whether the base variables on which an update is based on are obsolete) and in the update application rule. Since timestamps are used to assign priorities they have to be unique and monotonically increasing (for details refer [20, 13]).

The Voting Rule: The voting rule is used by each node to respond to update requests. Each update request contains the set of data items to be updated, the update timestamp, the base variables from which the updates were computed, and the timestamps of the base variables when the base variables were read. On receiving an update request, a node gives permission for the update by responding OK, or rejects the

[2] In the rest of the short note, points and nodes are used interchangeably.

update by responding REJ, or temporarily defers making a decision by storing the request in a deferred list. The decision making steps are the following.

1) Compare the timestamps of the request base variables with the timestamps of the corresponding local copies of the same variables.
2) Vote OK if the base variables are all current and this request does not conflict with any pending request.[3]
3) Vote REJ if the local copy of any base variable has a timestamp larger than the timestamp of the same variable in the update request.
4) Vote REJ if the base variables are current but the request conflicts with a pending request of higher priority[4]
5) If none of the above cases hold, defer voting. This happens when there is a conflict with a pending transaction of lower priority (since priorities are used, this defer cannot cause deadlocks).

The Update Rule: The update rule is Thomas' write rule. Before applying an update to a variable compare the update's timestamp with that of the variable's. If the variable has a higher timestamp then omit the update.

Since nodes seek update permission from all nodes in one line, only $m + 1$ permissions are needed for an update. The number of permissions sought is only $O(\sqrt{N})$ since in a finite projective plane $N = m^2 + m + 1$. This is significantly less than in Thomas' algorithm where nodes need $N/2 + 1$ permissions. Clearly, the larger the number of permissions needed the larger the number of messages and processing overheads. Hence, the finite projective plane based algorithm should perform better than the Thomas algorithm.

In the next section, we develop a queueing theoretic model to study the dynamic behavior of the algorithm and derive an expression for the mean request response time. The model allows us to study the behavior of the algorithm with respect to different input parameters and also allows us to investigate the relationship between the transaction response time and the transaction contention due to concurrent execution of transactions.

IV. Performance Model and Analysis

As in Section II, let $N = m^2 + m + 1$ be the number of nodes storing copies of the database in a distributed system. Transaction (request) arrivals at these nodes are assumed to be a Poisson process with rate λ_2. Jobs associated with these externally arriving transactions will be referred to as class 2 jobs. Each transaction reads $\bar{r}$ data items, performs some computation, and updates $\bar{w}$ data items.[5] The model does not require any specific topology for the network connecting the databases but it is assumed that the communication time, denoted by R_c, is fixed and load independent.

Apart from serving transactions arriving at the local nodes, each node participates in resolving requests initiated at other nodes. These update-permission requests that arrive at each node will be referred to as class 1 jobs. Besides voting requests, this class of jobs also includes commit and reject signals[6] arriving after a request has been resolved. In the following analysis, we will assume that the arrival of class 1 jobs is characterized by a Poisson process and that the arrival processes of class 1 and class 2 jobs are independent. This is an approximation supported by the following observation. Consider any particular node. Class 1 jobs arriving at this node correspond to transactions initiated at nodes belonging to the m different lines passing through the node being considered. Since messages initiated by nodes on different lines follow different paths and since the system is homogeneous, the arrival of class 1 jobs is well approximated by a sum of independent and identically distributed random variables. Furthermore, it is a good approximation that these random variables are exponentially distributed [17], and hence, the arrival rate of class 1 jobs can be approximated by a Poisson process with rate λ_1. Note that quality of the above approximation improves with the number of nodes in the system—as has been observed through a simulation study [4]. The analysis carried out in this short note can be easily extended to the case where the two arrival processes are correlated [21]. This would, however, require the knowledge of the exact nature of the dependency which in general is difficult to determine. Finally, we assume that class 1 jobs have preemptive priority over class 2 jobs. Note that λ_1 is not known *a priori* and must be obtained from the solution of the model. We assume that the service time of class 1 (class 2) jobs is generally distributed with first moment $E[S_1](E[S_2])$ and second moment $E[S_1{}^2](E[S_2{}^2])$. Fig. 1 shows the queueing model of each database node. The deferred list contains the transactions which are deferred and the pending list contains contains data items corresponding to transactions on which the node has voted but which have not yet been resolved.

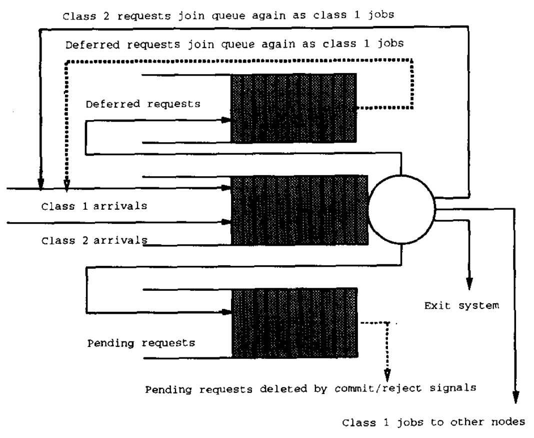
Class 2 requests join queue again as class 1 jobs
Deferred requests join queue again as class 1 jobs
Deferred requests
Class 1 arrivals
Class 2 arrivals
Exit system
Pending requests
Pending requests deleted by commit/reject signals
Class 1 jobs to other nodes

Fig. 1. The queueing model of each node.

A. Response Time Analysis

For a voting request serviced by a node, let p_{defer} be the probability with which a node defers voting. Also, let p_{rej} and p_{ok} denote the probabilities of voting REJ or OK, respectively. Let X be the random variable denoting the number of nodes a request goes through before being resolved. We can write the density function of X as follows

$$\begin{aligned} Pr(X = n) &= p'_{\text{rej}} p'^{\,n-1}_{\text{ok}} & n \le m; \\ &= p'_{\text{rej}} p'^{\,m}_{\text{ok}} + p'^{\,m+1}_{\text{ok}} & n = m+1; \end{aligned} \tag{1}$$

[3] Two requests are said to conflict if the intersection of the base variables of one request and the update variables of the other request is non-empty

[4] This is a pessimistic decision since the higher priority conflicting transaction might be eventually rejected. However, deferring a decision till the higher priority transaction's requests are resolved can cause deadlocks. Thomas' method uses a PASS vote. However, in a set-based scheme a PASS vote is equivalent to a REJ vote since for a node to perform updates all nodes in a set must vote OK.

[5] Note that $\bar{r}$ and $\bar{w}$ are means of distribution functions which characterize the sizes of the read and write sets in a transaction.

[6] Reject signals are sent to all the nodes even though it is sufficient to inform only those nodes that voted on this request.

where,

$$p'_i = p_i/(1 - p_{\text{defer}}), \qquad i = \text{rej, ok}.$$

The reason for normalizing p_i with $(1 - p_{\text{defer}})$ is as follows. Deferring a voting request results in delaying the resolution of the request. In the pure daisy chaining scheme considered in this short note, a deferred request must eventually be voted on before the request is resolved. Since in (1) we are interested in finding the number of nodes that a request goes through before being resolved, we consider only the probabilities which contribute to the resolution of the request, i.e., p_{rej} and p_{ok}. These are normalized with $1 - p_{\text{defer}}$ so that they sum to 1. We account for the defer votes later when computing the transaction response time. Equation (1) can then be explained as follows. For $n \leq m$, a request is resolved by the n^{th} node (the node initiating the request is denoted as 1) if the n^{th} node rejects the request and the previous $n-1$ nodes have accepted it. For $n = m+1$, the request is resolved either by a REJ vote or because the request has received $m+1$ OK votes. Let $\bar{X} = E[X]$ denote the expected value of X. $\bar{X}$ is given by

$$\bar{X} = \sum_{n=1}^{m+1} n \Pr(X = n) = \frac{1 - p'^{m+1}_{\text{ok}}}{1 - p'_{\text{ok}}}. \qquad (2)$$

Let $\bar{R}_1$ and $\bar{R}_2$ be the mean response times of class 1 and class 2 jobs, respectively. We are interested in deriving an expression for the mean transaction response time. This is dependent not only on the synchronization algorithm, but also the time spent elsewhere in the read and write phases of the transaction. When a transaction wants to read or write a data item in a replicated database the logical read or write request is translated into a set of physical reads and writes of the replicated data.[7] For correctness it is not necessary that all data items be read or written into. It is only necessary that the physical read and write sets intersect. For example, a transaction's logical read can be translated to a physical read of only the local copy of the database. In this case, logical writes would have to be translated into physical writes of the whole database. This read-one write-all method is analyzed in the next section and is a good method when the number of reads exceeds the number of writes by a large factor. When reads and writes are of equal cost, one good method (especially in conjunction with the finite-projective-plane based synchronization method) is one which translates a logical read into physical reads of copies at all nodes in any one line of the projective plane. (When a logical read is translated into multiple physical reads the logical base-variable value returned by the read is the most recently updated value—the one with the highest timestamp). Similarly, logical writes are translated into physical writes of copies at all nodes in any one line of the projective plane. Since any two lines of the projective planes intersect, the physical read and write sets always intersect. Also, the read and write sets are of cardinality $O(\sqrt{N})$. This symmetric $O(\sqrt{N})$-read $O(\sqrt{N})$-write algorithm is also analyzed below.

Read-one Write-all Algorithm First we need to find the mean time before a node votes on a request. Note that a request's response can be either REJ or OK or the response can be deferred. Let $\bar{R}_{\text{defer}}$ be the mean time a request spends in the deferred list each time it is deferred. Let X_d be the random variable which denotes the number of times a request is deferred.[8] Clearly, the probability that a request is deferred i times is given by

$$Pr[X_d = i] = p^i{}_{\text{defer}}(1 - p_{\text{defer}}) \qquad i = 1, 2, 3, \cdots, \infty.$$

Note that $i = 0$ corresponds to the case when the first time the request is considered for a vote it is assigned an OK or REJ vote. The mean time required to assign a vote to a request at a given node given that it is deferred i times is $(i+1)\bar{R}_1 + i\bar{R}_{\text{defer}}$. This is because each time the request is taken out of the deferred list it faces the mean response time as a class 1 transaction before it is assigned an OK or REJ vote or is deferred again. From this, the mean voting time is given by

$$\bar{R}_{\text{vote}} = \sum_{i=0}^{\infty} p^i{}_{\text{defer}}(1 - p_{\text{defer}})[(i+1)\bar{R}_1 + i\bar{R}_{\text{defer}}] = \bar{R}_1 + \frac{p_{\text{defer}}}{1 - p_{\text{defer}}}(\bar{R}_1 + \bar{R}_{\text{defer}}). \qquad (3)$$

Since the commit/reject signals are never deferred, these messages face the mean response time of class 1 transactions, $\bar{R}_1$, at each database node. Hence, the mean request response time is

$$\bar{R}_{\text{req}} = \bar{R}_2 + [\bar{X}\bar{R}_{\text{vote}} + (\bar{X} - 1)R_c] + [R_c + \bar{R}_1], \qquad (4)$$

where the first term is the mean time (measured from the instant the transaction arrived at the node) before the initiating node completes the computation and inserts it into its class 1 queue, the second term is the time required to resolve the request and the third term is the time required to commit/reject the request at each database node.[9]

$O(\sqrt{N})$-ReadO$(\sqrt{N})$-Write Algorithm The multiple reads of all nodes in line are assumed to performed in a daisy-chained manner with the read request arriving at each node in the chosen line as a class 1 transaction. The mean time in the read phase is

$$\bar{R}_{\text{read}} = m(R_c + \bar{R}_1). \qquad (5)$$

Equation (5) is added to (4) to obtain the total request response time for this algorithm.

Computing the Conflict-Probabilities: The probability of conflict depends on the sizes of transaction's read and write sets ($\bar{r}$ and $\bar{w}$) and the number of transactions whose update-requests are concurrently being synchronized (remember that a serial execution has no conflicts). In computing the conflict probabilities, $\bar{r}$ and $\bar{w}$ are input parameters but the concurrency in the system has to be determined from the system's throughput and response times. The probability of conflict is used to compute the probabilities p_{ok}, p_{defer}, and p_{rej}. We assume that update-requests reference data items uniformly, i.e., a request references any data item with equal probability $1/D$ where D is the number of data items in the database. We can also incorporate methods of computing conflict probabilities for non-uniform access—such as the method in [18], [19]—into our model.

We recapitulate that associated with each node are two lists: the pending list and the deferred list (see Fig. 1). The time a request is in the pending list at a given node depends on the distance (in terms of number of hops in the daisy chain) of the node from the node that initiated the request. We estimate the mean time a request is pending as

$$\bar{R}_{\text{pend}} = \frac{\bar{R}_{\text{req}} - \bar{R}_2}{2}. \qquad (6)$$

A request is inserted into the pending list with probability p_{ok} and hence, the rate at which data items are inserted into the pending list is $\min(\lambda_2, \mu_2)\bar{w}p_{\text{ok}}\bar{X}$, where $\min(\lambda_2, \mu_2)p_{\text{ok}}$ is the rate at which voting requests are sent out from each node, $\bar{X}$ is the mean number of nodes from which a node receives voting requests, and each

[7] This read-set and write-set is not to be confused with transaction read-sets and write-sets which are the logical data items read and written by a transaction.

[8] A request can be deferred more than once due to multiple conflicts.

[9] The commit/abort messages are not sent in a sequential daisy-chained manner and instead are sent to all nodes in parallel. This is reflected in the expression for the mean request response time.

transaction updates $\bar{w}$ data items. Using Little's Law [9], $\bar{N}_{\text{pend}}$, the mean number in the pending list is given by

$$\bar{N}_{\text{pend}} = \min(\lambda_2, \mu_2)\bar{X}\bar{w}p_{\text{ok}}\bar{R}_{\text{pend}}. \tag{7}$$

Since the access pattern of the D data items by requests is assumed to be uniformly distributed, the probability that the data item referenced in the request does not conflict with any update to the same data item by requests in the pending list is equal to $(1 - \bar{N}_{\text{pend}}/D)^{\bar{r}}$ where $\bar{r}$ is the mean number of data items read by the request. Hence, the probability that it does conflict is equal to $1 - (1 - \bar{N}_{\text{pend}}/D)^{\bar{r}}$.

Let p_{sd} be the probability that the timestamp of the transaction is greater than the timestamp of the data in the local copy. Thus, the probability that all the transaction's base variables are current, i.e., the timestamps are smaller than the corresponding timestamps in the local copy, is given by $p_{\text{sd}}^{\bar{r}}$. Thus the probability that at least one data item in the local copy has a higher timestamp is given by $1 - p_{\text{sd}}^{\bar{r}}$. Similarly, let p_{sp} be the probability that the timestamp of the transaction is greater than at least one data item in the pending list which is referenced by the transaction. From the above, we can write down the following probabilities:

$$\begin{aligned} p_{\text{ok}} &= p_{\text{sd}}{}^{\bar{r}}(1 - \bar{N}_{\text{pend}}/D)^{\bar{r}} \\ p_{\text{rej}} &= (1 - p^{\bar{r}}{}_{\text{sd}})(1 - \bar{N}_{\text{pend}}/D)^{\bar{r}} + p_{\text{sp}}[1 - (1 - \bar{N}_{\text{pend}}/D)^{\bar{r}}] \\ p_{\text{defer}} &= (1 - p_{\text{sp}})[1 - (1 - \bar{N}_{\text{pend}}/D)\bar{r}]. \end{aligned} \tag{8}$$

The probabilities p_{sd} and p_{sp} are evaluated from the input parameters of the system and assuming that all the clocks in the different nodes are completely synchronized. Finally, the mean time $\bar{R}_{\text{defer}}$ is estimated by $\bar{R}_{\text{pend}}/2$.

C. Response Time of Each Class of Customer

The response time for each class of customer can be found by solving the standard two class priority queue. Then the mean response time can be evaluated using Little's Law. However, as pointed out before, we do not know *a priori* the values of λ_1. Now, λ_1 depends on the rate at which transactions are initiated in the system and also on the read/write method used. If the *read-one write-all* method is used then class 1 jobs consist of voting requests to the participating nodes and commit/reject signals which are sent to all the nodes. Thus each node receives on the average $(1 + \frac{\bar{X}}{N})$ number of class 1 jobs for each transaction that is initiated in the system. Since there are N nodes and at each node the throughput is λ_2, the total arrival rate of class 1 transactions is estimated by

$$\lambda_1 = \min(\lambda_2, \mu_2)N(1 + \frac{\bar{X}}{N}). \tag{9a}$$

On the other hand if the $O(\sqrt{N})$-Read $O(\sqrt{N})$-Write algorithm is used then class 1 transactions consist of read requests, request for votes, and commit/reject signals. Read requests, voting requests, and commit/reject signals are sent to all the nodes in the set which is of size $m + 1$. For this protocol the arrival rate of class 1 jobs is approximated by

$$\lambda_1 = \min(\lambda_2, \mu_2)m\left(2 + \frac{\bar{X}}{m}\right). \tag{9b}$$

V. Results and Discussion

This section discusses the results obtained by solving the model discussed in the previous section. Since the analysis does not yield closed form solutions, it is solved iteratively. For all cases the iterative method converged within a few steps.

Fig. 2 plots the request response time as a function of the order for different loads. The load denotes the total arrival rate of transactions to the entire distributed database system. For a given load increasing the order, i.e., increasing the number of nodes, corresponds to decreasing the arrival rate at each node. It is assumed here that the load is equally divided among all the nodes. The other parameters are shown in the figure. From the figure, it can be observed that at low load there is no advantage in replicating as the communication cost dominates and increases the mean request response time. This, however, is not true at high load because the waiting time of transactions as class 2 and class 1 customers increases with increase in load. Increasing the replication reduces the arrival rate at each node and hence, the waiting time in each queue. However, increasing the number of nodes beyond a certain number results in only a marginal reduction in the response time and is not enough to compensate for the increase in communication overhead. Note that the degree of replication that achieves minimum response time increases with the increase in load.

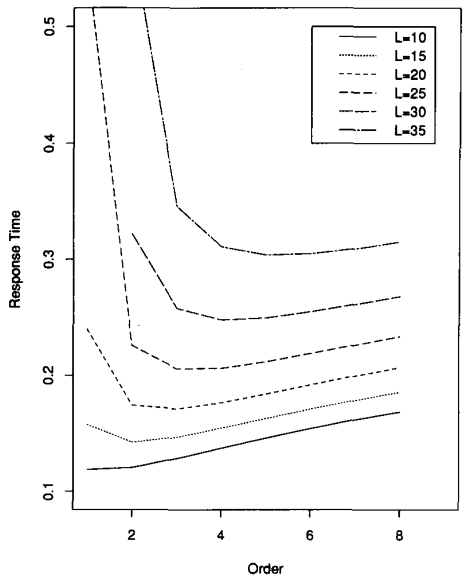

Fig. 2. Performance of the Read-One Write-All Algorithm (L = Load, $\mu_1 = 75.0$, $\mu_2 = 25.0$, $rc = 0.001$, $D = 5.0e + 05$, $\bar{r} = \bar{w} = 10$).

Performance results for the $O(\sqrt{N})$-Read $O(\sqrt{N})$-Write algorithm are shown in Fig. 3. The behavior is very similar to the read-one write-all protocol. One trade-off that exists between these two protocols relates to the communication overhead and the traffic that is generated by these protocols. In the read-one write-all protocol, the read operation is performed locally but the update is performed in all the N copies. On the other hand in the read-set write-set protocol the data is first read from m nodes and then updated in the same or some other m nodes.

At low load the read-one write-all protocol performs better since the read is performed locally. In the $O(\sqrt{N})$-Read $O(\sqrt{N})$-Write algorithm, requests experience the mean response time as class 1 transactions in m nodes from which the data is read. As the load is increased, the $O(\sqrt{N})$-Read $O(\sqrt{N})$-Write algorithm performs better since the total number of messages generated are fewer than the read-one write-all protocol. These results are shown in Fig. 4 which plots the mean request response time for the two protocols as a function of the load for a fixed order equal to 4.

The effect of the relative sizes of the read and write sets on the performance was also studied. For a fixed communication cost, the overall behavior of the two algorithms were similar. At high load, the

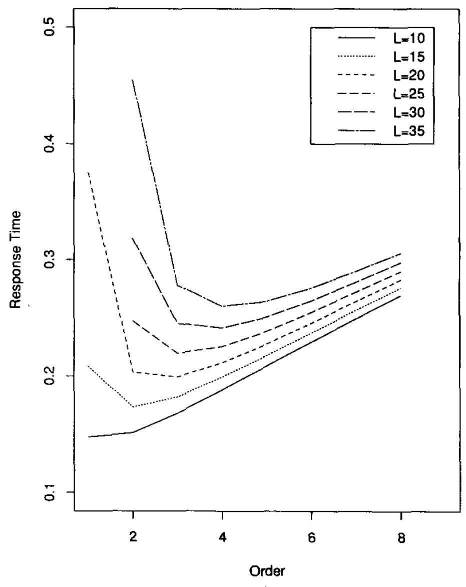

Fig. 3. Performance of the $O(\sqrt{N})$-Read $O(\sqrt{N})$-Write Algorithm (L = Load, $\mu_1 = 75.0$, $\mu_2 = 25.0$, rc = 0.001, $D = 5.0e+05$, $\bar{r} = \bar{w} = 10$).

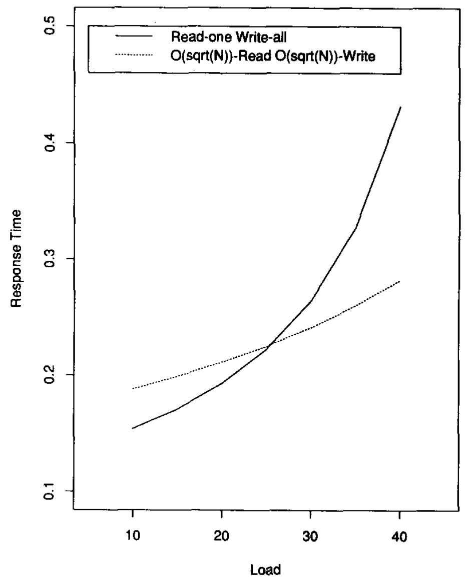

Fig. 4. Comparison of the two algorithms (oa : Read-One Write-All, ss : $O(\sqrt{N})$-Read $O(\sqrt{N})$-Write, $\mu_1 = 75.0$, $\mu_2 = 25.0$, $rc = 0.001$, $D = 5.0e+05$, $\bar{r} = \bar{w} = 10$, Order = 4).

$O(\sqrt{N})$-Read $O(\sqrt{N})$-Write algorithm performs better (unless the transaction is read-only in which case the read-one algorithm always performs better). The communication cost rc usually depends on the topology and typically increases with N and the system load. In this short note, rc was assumed to be constant and load independent. The sensitivity of the system performance with respect to rc can be addressed once this is appropriately characterized with respect to the relevant system parameters. This is a subject of future research.

Another interesting relationship exists between the contention and the mean transaction response time. When the mean request response time is high, a request remains in the pending list for a longer time which in turn increases the probability of conflict. The probability of conflict in turn affects the mean request response time since lower probability of OK would result in quicker rejects and thus smaller time to resolve the request. Table I shows the fraction of transactions that are rejected for different system sizes and two different loads. The important factor that affects the contention is the time that a request spends as a class 1 transaction. Increasing the number of nodes increases this time and hence, the probability of conflict. Thus as can be observed from Table I, the probability of a transaction being rejected increases with the increase in the system size.

TABLE I
FRACTION OF THE TRANSACTIONS THAT ARE ABORTED AS A FUNCTION OF ORDER ($\mu_1 = 75.0$, $\mu_2 = 25.0$, rc = 0.001, $D = 5.0e+05$, $\bar{r} = \bar{w} = 10$).

Order	Load=30 Fraction Aborted	Load=35 Fraction Aborted
1	0.0495	0.071
2	0.0936	0.0944
3	0.1355	0.136
4	0.1748	0.1752
5	0.2116	0.2119
6	0.2461	0.2464
7	0.2786	0.2789
8	0.309	0.309

From the above, it appears that mean transaction response time may not be a very good measure to compare different system configurations. Indeed for large system sizes the mean request response time can decrease due to a large number of aborts. In order to capture the trade-off that exists between the contention and the mean request response time, we propose using the metric *conflict response-time* product defined as the product of the mean request response time and the probability of a transaction being aborted. We believe that choosing system parameters that minimize this measure is a more appropriate choice than choosing parameters that minimize only the response time.

Fig. 5 plots the conflict-response time product as a function of the different number of nodes for the read-one write-all algorithm. Note that at higher loads, the value of m that minimizes the conflict-response time product is lower than that which minimizes the mean request response time. Similar behavior was also observed for the $O(\sqrt{N})$-Read $O(\sqrt{N})$-Write algorithm.

VI. Concluding Remarks

We evaluated the system-performance effect of an efficient $O(\sqrt{N})$ symmetric algorithm for multiple-copy update-synchronization in distributed databases. This algorithm (an adaptation of Maekawa's $O(\sqrt{N})$ algorithm for distributed mutual exclusion [14]) was developed using finite projective planes to obtain a family of intersecting sets with the needed symmetry properties. However, any sparse regular graph of diameter 2 can be used to generate the sets needed by the algorithm presented. Since $\Omega(\sqrt{N})$ is a lower bound on the degree of regular diameter-2 graphs, using other graphs can at best improve the constants in comparison to the finite projective plane based algorithm. For a homogeneous system, an $O(\sqrt{N})$ algorithm is the best possible symmetric algorithm (the Thomas algorithm is $O(N)$). Previous work on efficient update-synchronization algorithms has been limited to complexity analysis of the algorithms. However, the most important benefit of an efficient algorithm is the reduction in transaction response-time. To determine this reduction, the dynamic behavior of the algorithm needs to be analyzed. This dynamic behavior (performance) analysis was done for the $O(\sqrt{N})$ algorithm using a hierarchical multiclass priority queueing model. We believe that

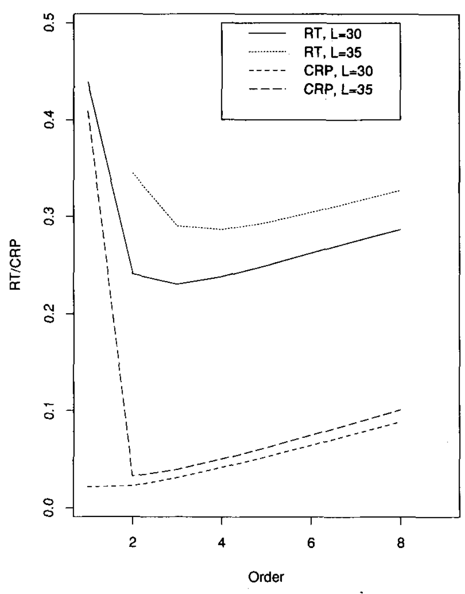

Fig. 5. Comparison of the measures Response Time (RT) and Conflict-Response Time Product (CRP) ($\mu_1 = 75.0$, $\mu_2 = 25.0$, $rc = 0.001$, $D = 5.0e + 05$, $\bar{r} = \bar{w} = 10$).

transaction response time should not be the only metric in evaluating these systems because it does not completely characterize the effect of concurrency. With increasing concurrency, the probability of conflict (and hence, aborted transactions) increases. This is not reflected in the mean transaction response time even though increased conflicts increase the transaction response time perceived by the user (this is because retries are considered separate transactions). We used the conflict-probability response-time product as our performance metric and found that the system size for optimal performance is lower (lesser concurrency) using this metric than when using response time alone as a metric.

An important criteria in the choice of sets is the effect of the choice on fault-tolerance. A problem for future study is constructing a family of sets which not only yields an efficient algorithm but is also non-dominated (as defined in [5]). An immediate extension of this work is to show that finite projective planes yield a non-dominated family of sets. The Thomas' majority consensus scheme is tolerant to $N/2$ node failures. The finite projective plane based scheme can function as long as all nodes in one line are functioning. Hence, as many as $N - (m + 1)$ nodes can fail and the system can still function. However, the reliability of the projective plane method is sensitive to the combinations of nodes which fail and not just on the number of failing nodes (as in the majority consensus). If all the $m + 1$ nodes in a line fail, the whole system becomes unavailable. In the case of partitions, partitioning the system into two parts always leaves a functioning partition when using the majority consensus. When using projective planes, a partition can function only if it contains all nodes in any one line (in which case no other partition will have all nodes in any one line). However, it is possible that no partition contains a line. When the system partitions into more than two parts, for the majority consensus scheme to function at least one partition must have $N/2 + 1$ nodes. With the projective plane scheme, a partition can function with as few as $m + 1$ nodes in one partition provided all these nodes are on one line. Another extension to this work is the availability analysis of $O(\sqrt{N})$ algorithms.

REFERENCES

[1] A. A. Albert and R. Sandler, *An Introduction to Finite Projective Planes*. New York: Holt, Rinehart and Winston, 1968.
[2] S. Y. Cheung, M. Ammar, and M. Ahamad "The grid protocol: A high performance scheme for maintaining replicated data," *Proc. Sixth Int. Conf. of Data Eng.*, 1990, pp. 438–445.
[3] C. Colbourn and P. C. van Oorschot, "Combinatorial designs in computer science," *ACM Computing Surveys*, vol. 21, June 1989.
[4] D. L. Eager, E. D. Lazowska, and J. Zahorjan, "Dynamic load sharing in homogenous distributed systems," *IEEE Trans. Software Eng.*, vol. SE-12, pp. 662–675, May 1986.
[5] H. Garcia-Molina and D. Barbará, "How to assign votes in a distributed system," *JACM*, vol. 32, no. 4, pp. 841–860, Oct. 1985.
[6] D. K. Gifford, "Weighted voting for replicated data," *Proc. Seventh Symp. Operating Syst. Principles*, Dec. 1979, pp. 150–162.
[7] G. H. Hardy and E. M. Wright, "Theorem 418," in *An Introduction to the Theory of Numbers*. Oxford: Oxford Univ. Press, 1954, p. 343.
[8] A. J. Hoffman and R. R. Singleton, "On Moore graphs with diameter 2 and 3," *IBM J. Res. Develop.*, vol. 4, pp. 497–504, 1960.
[9] L. Kleinrock, *Queueing Systems Volume 1: Theory*. New York: John Wiley, 1975, ch. 2.
[10] T. V. Lakshman and A. K. Agrawala, "$O(N$ SQRT$(N))$ Decentralized commit protocols," in *Proc. Fifth Symp. Reliability in Distributed Software and Database Syst.*, Jan. 1986, pp. 104–110.
[11] ——, "Efficient decentralized consensus protocols," *IEEE Trans. Software Eng.*, vol. 12, pp. 600–607, May 1986.
[12] T. V. Lakshman and V. K. Wei, "On computing functions of distributed information," Presented at the *Proc. 26th Annu. Allerton Conf. Computing, Commun., and Contr.*, Oct. 1988.
[13] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Commun. ACM*, vol. 21, pp. 558–565, July 1978.
[14] M. Maekawa, "A SQRT(N) algorithm for mutual exclusion in decentralized systems," *ACM Trans. Comput. Syst.*, vol. 3, pp. 145–159, May 1985.
[15] S. J. Mullender and P. M. B. Vitanyi, "Distributed match-making for processes in computer networks," Rep. CS-R8424, Centrum voor Wiskunde en Informatica, Amsterdam, Dec. 1984.
[16] F. S. Roberts, *Applied Combinatorics*. Englewood Cliffs, NJ: Prentice Hall, Inc., 1984, Sect. 9.3.3, 9.3.4, and 9.3.5.
[17] S. Salza and S. S. Lavenberg, "Approximating response time distributions in closed queueing networkmodels of computer performance," *Performance*. Amsterdam: North Holland, 1981, pp. 133–145.
[18] M. Singhal and Y. Yesha, "A polynomial algorithm for computation of the probability of conflicts in a database under arbitrary data access distribution," *Inform. Processing Lett.*, vol. 27, no. 2, pp. 69–74, Feb. 1988.
[19] ——, "Analysis of transaction blocking in arbitrary data access distribution in database systems," presented at the *Proc. 2nd Int. Workshop on Appl. Math. and Perform./Reliability Models of Comput./Commun. Syst.*, Univ. of Rome II, Rome, Italy, May 25–29, 1987.
[20] R. H. Thomas, "A majority consensus approach to concurrency control for multiple copy databases," *ACM Trans. Database Syst.*, vol. 4, no. 2, pp. 180–209, July 1979.
[21] D. Towsley and S. K. Tripathi, "A single server priority queue with server failure and queue flushing," Dept. of Comput. Sci., Rep. TR-2207, Univ. of Maryland, College Park, Feb. 1989.