Replication Algorithms in a Remote Caching Architecture

Avraham Leff, *Member, IEEE*, Joel L. Wolf, *Senior Member, IEEE*, and Philip S. Yu, *Fellow, IEEE*

Abstract—**We study the cache performance in a *remote caching architecture*. The high performance networks in many distributed systems enable a site to access the main memory of other sites in less time than required by local disk access. *Remote memory* is thus introduced as an additional layer in the memory hierarchy between local memory and disks. *Efficient* use of remote memory implies that the system caches the "right" objects at the "right" sites. Unfortunately, this task can be difficult to achieve for two reasons. First, as the size of the system increases, the coordinated decision making needed for optimal decisions becomes more difficult. Second, because the participating sites in a remote caching architecture can be *autonomous*, centralized or socially optimal solutions may not be feasible. In this paper we develop a set of distributed object replication policies that are designed to implement different optimization goals. Each site is responsible for local cache decisions, and modifies cache contents in response to decisions made by other sites. We use the optimal and greedy policies as upper and lower bounds, respectively, for performance in this environment. Critical system parameters are identified, and their effect on system performance studied. Performance of the distributed algorithms is found to be close to optimal, while that of the greedy algorithms is far from optimal.**

Index Terms—**Autonomy, distributed systems, object replication, performance comparison, remote caching.**

I. Introduction

In modern distributed systems, large numbers of computing sites are connected together by fast networks. The availability of high speed interconnection has created the potential for a new type of resource sharing. In this environment, it is possible to develop efficient mechanisms that support request/response exchanges for objects that reside on a remote site. This ability to access objects cached at remote sites introduces a new level in the classic memory hierarchy—main memory accessed through the network—whose access time may be significantly faster than that of local disks. We call this *remote memory*. Unlike shared main-memory architectures, sites using remote memory do not require the capability of direct read/write into remote memory locations.

Remote memory is important because disk access performance has been limited by seek time, stuck for decades in the range of a few tens of milliseconds. In contrast, current remote procedure calls (RPC) implementations over Ethernet take only a few milliseconds for the round trip [27]. Moreover, the bottleneck in communication protocols is CPU power and software overhead. With RISC technology doubling CPU speed every few years, we can expect even smaller ratios of remote memory *versus* local disk access time in the near future. Furthermore, faster gateways, higher network bandwidth, and specialized hardware [1] will steadily bring down communication overhead over local and metropolitan area networks. Implementations of systems using remote memory are already being built [2], [6], [10], [13], [19], [20], [26].

A *remote caching architecture (RCA)* makes use of remote memory by allowing all sites in the system to take advantage of each other's local memory. The symmetric architecture blurs the distinction between clients and servers because all sites in the system can "serve" requests if their buffers (main-memory) contain the requested data item. An RCA resembles distributed shared virtual memory (DSVM) [20] in that both types of system take advantage of the aggregate memory that is available (through a network) in a distributed system. There are two main differences, however, between the systems. First, remote caching architectures emphasize the differences between the memory levels (i.e., local, remote, and disk), in contrast to the emphasis that DSVM places on a single large, homogeneous, memory space. As a result, RCA research focuses on such *policy* issues as what objects should be cached at what sites instead of simply caching objects on a demand basis as in DSVM. The systems also differ in *implementation*. In an RCA, remote memory need not be mapped into a single, coherent, virtual memory space. Sites do not need to have the same page sizes or memory architectures; all they require is that they share a common naming scheme for objects in the system (the distributed object model). Because an RCA does not require the full functionality of a DSVM, system overhead can be reduced. Of course, an RCA can also be implemented in a DSVM: the key feature is that sites can request (and receive) objects from remote sites with an order of magnitude faster response time than even local disk access.

Simulation studies have shown that performance in an RCA is better, over a wide range of cache sizes, than a distributed client/server architecture [23]. The performance gains are due to the large amount of remote memory made available by the (symmetric) remote caching architecture. However, the following tradeoff in object replication must be resolved in order to use memory resources efficiently. On the one hand, each site should replicate (i.e., cache) important objects in main-memory, and store less important objects on disk. On the other hand, such naive cache management in an RCA is *inefficient* in the sense that memory resources are not utilized

Manuscript received January 2, 1992; revised July 30, 1992.

A. Leff was with IBM Enterprise Systems, Poughkeepsie, NY, 12602. He is now with the IBM Research Division, T. J. Watson Center, Yorktown Heights, NY 10598.

J. L. Wolf and P. S. Yu are with the IBM Research Division, T. J. Watson Center, Yorktown Heights, NY 10598.

IEEE Log Number 9213475.

as well as they would be in a centrally coordinated system [17]. Some sites should instead cache *less* important objects, and rely on remote memory to access important objects. This counter-intuitive approach can improve both local and overall system performance, because fewer objects must be accessed on disk. The problem, of course, is to make this idea precise: how many replicas of each object should be maintained?

We first devise optimal object replication algorithms for an RCA. We consider both the cases of optimizing the average performance and of optimizing the performance of the worst site. The former case is solved by reduction of the problem to a capacitated transportation problem [15]. The optimal number of replicas is a function of the hot-set curve, available cache storage, and differences among site access patterns. However, sites in distributed environments might not wish to be constrained by the decisions of other sites. (This is known as *autonomy* [14].) Thus efficient use of RCA system resources is especially difficult to accomplish. Even if autonomy is not an issue, because optimal solutions require that decisions be coordinated among the sites, they may not scale up as the number of sites in the system increases. We therefore investigate two distributed algorithms that partition the cache management problem among the sites in the remote caching architecture. Each site maintains a snapshot of the system configuration, and as conditions change sites may change their own (local) cache management decisions. The same mechanisms that make remote memory possible (e.g., broadcast) are used to propagate dynamic state information as well. Site autonomy is factored in implicitly, because sites cannot directly affect the decisions of any other site. The two distributed algorithms differ in their objectives on whether to pursue local or global performance optimization. Also considered are two simple greedy algorithms. The optimal and greedy policies provide upper and lower bounds on the performance for this environment, respectively.

In the next section, we describe our model of the RCA system, and formalize the problem of cache management in an RCA. Section III presents various cache management strategies, and Section IV describes their implementation in this environment. The performance of these algorithms (as a function of various system parameters) is analyzed in Section V. In Section VI we summarize our results and discuss some future work suggested by this paper.

II. Remote Caching Architecture

A. The Model

The memory hierarchy of the RCA consists of local main-memory, remote main-memory (accessed over the communications network), and disk. In terms of access time, there are single order of magnitude differences between the local memory (tenths of a millisecond), remote memory (milliseconds) and disks (tens of milliseconds). The RCA cache management system must implement the following three components: 1) an object location algorithm, 2) a replacement algorithm, and 3) a consistent update algorithm. A set of these algorithms, together with a detailed discussion of execution paths for read and write file/object access, can be found in [23]. (Algorithms that maintain transaction serializability in such an environment are discussed in [4], [8], [30].) Basically, if a site fails to find a copy of an object in local cache, then the site broadcasts a request, and at the same time sets a timeout. All sites with a copy of the requested object queue for the network and reply: the requesting site simply discards all replies after the first.[1] Expiration of the timeout period indicates that the object is not cached at any remote site. The object must then be fetched from disk. In other words, when an object is needed at a given site, the site traverses the memory hierarchy looking at local cache, remote cache, and disk in turn.

Let c_{ij} denote the time needed to access the ith object at the jth site ($i = 1, \cdots, M, j = 1, \cdots, N$). The cost function that we want to minimize involves c_{ij}.

Let p_{ij}^{LH}, p_{ij}^{RH}, and p_i^{NH} denote, respectively, the Local Hit probability, Remote Hit probability, and No Hit probability, when accessing the ith object at the jth site. These probabilities sum to 1.0 because they represent the traversal of the memory hierarchy that is done in order to access an object. Then,

$$c_{ij} = (t_l \times p_{ij}^{LH}) + (t_r \times p_{ij}^{RH}) + (t_d \times p_i^{NH}) \qquad (1)$$

where t_l is the time required to access local main memory, t_r is the time to access remote main memory (including network delays), and t_d is the time needed to access the disk storing the ith object.[2] (Of course, these access times are cumulative. For example, the time needed to test for a local cache hit adds to the total t_r because remote memory is only accessed after attempting to access local memory.)

The probability of an object being cached by a given site depends on 1) whether the object is considered important enough to cache in the first place, and 2) how many *other* objects are eligible for caching by that site. The second factor determines the hit ratio for *eligible* objects. Let $X_{ij} = 1$ when the ith object is eligible for caching (i.e., it may be stored in main memory) at the jth site, and $X_{ij} = 0$ otherwise. Let $[H_{ij}(X)]$ be the matrix whose i,jth term is the hit ratio of object i at site j. This matrix specifies the configuration of the system at a given moment. Then

$$p_{ij}^{RH} = (1 - \prod_{k=1 k \neq j}^{N} (1 - H_{ik}(X))) \times (1 - H_{ij}(X)), \qquad (2)$$

and

$$p_i^{NH} = \prod_{j=1}^{N} (1 - H_{ij}(X)). \qquad (3)$$

These equations apply trivially in the situation where $H_{ij}(X)$ is binary-valued (i.e., a site never allows more objects to be cache "candidates" than it has storage for).

[1] Alternatives to the broadcast mechanism can be devised. For example, in a database environment with a centralized lock manager, lock retention schemes can track the location of objects in the RCA system [9].

[2] Note that our model ignores the issue of queueing and contention at sites in the system. These factors imply that the cost of accessing an object may depend on the site at which the object is cached. We believe, however, that these factors impose "second-order" effects which do not much change the development of the caching algorithms for the RCA.

They also apply in the situation where, because the site wants more objects in cache than it can physically store, $H_{ij}(X)$ varies between 0 and 1. Note that the matrix X (in theory) completely determines the hit-ratio because, regardless of the caching policy, once the system specifies the eligibility of objects at a given site, the hit ratio for that object is implicitly set as well. For example, if a naive replacement policy such as "uniform replacement" is used, then $H_{ij}(X)$ can be computed as M/S (where M is the number of objects and S is the per-site cache size) for all objects i. Less trivially, cache replacement algorithms such as LRU can also be approximated as a function of the X matrix [7], [28].

In this paper, we shall assume, for the most part, that H_{ij} is *indeed* binary valued, with sites caching only those objects for which they have space. Because the $H_{ij}(X)$ term is included in the equations, our approach is easily extended to the case of more complicated hit-ratio functions.

B. The Problem

The problem addressed in this paper is how, given the RCA system parameters, can we best specify X so as to minimize the "cost" of the system. The task of specifying the X_{ij} is a version of the *File Allocation Problem (FAP)* [11], [29]. Certain versions of the FAP problem are NP-complete [11], [25]. Heuristics which perform well have been proposed [5], [21], [25], [31]. In our case, there are M objects and N sites in the system, and each object can be cached at any site. There are thus 2^{NM} possible values for X.

Let C be the average time needed to access objects, given an RCA configuration X. This cost function will, in general, be a function of the c_{ij} of (1). The optimization approach will require complete and centralized information, and use a centralized algorithm to compute the file assignment. These limitations suggest that distributed algorithms may be worthwhile—even if they do not yield optimal performance. The issue of *autonomy*, however, is the major difference between the classic FAP problem and the problem which arises in a remote caching architecture context. Prior work in the area of FAP assumes that the sites in the system are committed to optimizing overall system performance. In an autonomous [14] RCA environment, however, a given workstation is not specifically interested in improving performance at another workstation, but is rather concerned with decisions that affect its own performance. Sites are willing to cooperate in servicing requests for copies, but they are not willing to constrain their caching decisions based on these requests. A given site can only make caching decisions regarding its resources. No site can make such decisions about another site's resources. In an autonomous environment, even if sites are willing to cooperate in decision making, they will not agree to cache items if the result is substantially worse performance than other sites in the system. Autonomous sites may not even agree to suffer performance penalties—even if a given allocation policy results in better *average* performance for the system as a whole. Because sites cannot unilaterally determine the cache contents of other sites, classic FAP approaches will not necessarily extend to an autonomous environment.

C. Dimensions of the Problem

There are quite a few dimensions to the problem of determining the optimal X. We discuss some of them here, and then show where this work fits into the large state space.

- *Cost Function*

We obviously need to specify the nature of the cost function C, which must be minimized. At one extreme, sites can be interested solely in *local* performance. At another extreme, sites can be interested in improving *global* performance. Alternatively, sites can implement a *fair* policy, in which no site does "much" worse than any other site. The issue of *efficiency* is closely related to that of the cost function. Under a *global* optimization policy, sites cooperate with one another, so that the number of object replicas in the overall system results in optimal (overall) performance. Thus, a site may cache an object that *it* rarely accesses, simply because other sites access it often. Under a *local* optimization policy, however, sites face the following tradeoff. On the one hand, a site can be "greedy" and simply cache as many of the most valuable objects as it can. On the other hand, if many sites replicate the same objects, then a given site is "wasting" cache storage because it could have used the space to cache an unreplicated object, and rely on other sites when it needs to access the replicated object. The key point is that even if a site caches relatively less important objects, and pays more (because it must go over the network) for more valuable objects, not only may *overall* system performance improve (because storage is used more efficiently), but *local* performance may improve as well.

- *Eligibility for Replication*

When a site caches an object, this has a number of consequences. If the object is "read-only," then the site obviously gains from reducing object access time. The only issue that must be resolved is how to trade-off the presence of one object in cache against the crowding out of *another* object from the cache. However, when objects are "read/write" the situation becomes far more complicated. Although the site still benefits in read situations, writes require the application of a consistency maintenance algorithm. The system must somehow ensure that replicas of an object maintain the same value: this process requires inter-site communication, interrupting the sites so as to receive the messages, and finally the update propagation. This cost of consistency maintenance is such that a site may actually incur a performance *penalty* by caching an object replica.

In this paper, our approach is based on the following observation. A major reason for sites to cache read-write objects—despite the associated overhead for consistency maintenance—is that the object is so important that it must always be readily available for read access. In other words, the "delta" between disk and local memory access is so large that it makes sense to maintain replicas of "write" objects—despite the additional overhead. In an RCA, this motivation is not as strong—as long as a *single* copy of the object can be accessed from some site in the system. In such a case the delta that motivates object replication is only between local and remote memory, and the overhead

of consistency maintenance usually outweighs the smaller delta.

This intrinsic characteristic of an RCA architecture therefore suggests the following heuristic. Read-write objects are constrained to have at most *one* copy available in a site's main-memory, eliminating the need for replication decisions. Sites do, of course, benefit from cached read-write objects; replication decisions, however, are only made about read-only objects. This paper investigates the more clear-cut tradeoffs involved with an object's read value.

- *Local Object Value*

The "value" (i.e., performance benefit) that a cached (read-only) object gives to the caching site, is proportional to the frequency that the object is accessed by transactions executing at the site. Sites can thus rank objects based on their access frequency.[3] Access frequencies can be estimated with varying degrees of accuracy. At one end of the spectrum, sites have perfect information. More realistically, sites can dynamically estimate these values (with an exponential weighting term, for example). At the other extreme, sites may not bother to calculate values explicitly, and simply manage cache with an LRU policy. (This dimension relates to the issue of whether object access rates vary *over time*. For example, in a "static" environment, perfect information is a much more reasonable assumption than in a dynamic environment.)

- *Degree of Coordination*

If a site has no knowledge about the contents of *other* sites' caches, then the site should simply cache as many of its most valuable objects as it can (see above). By contrast, if a site knows what objects are cached at other sites, it can rely on remote sites for those objects and instead cache objects that would otherwise be only accessible on disk. At one extreme, sites may make cache decisions in a completely *coordinated* fashion (this leads to optimal global performance). More realistically, sites may make decisions in a *sequential (or synchronous)* fashion. Only one site at a time makes a set of decisions. Information about decision outcomes are then passed to the next site. Alternatively, sites may make decisions in *asynchronous* fashion. Although sites make the results of their decisions available to other sites, these other sites may be making their own decisions simultaneously. (This dimension is related to the question of *what* information is communicated between sites. Useful analogies have been drawn to research in load sharing—for example "sender-initiated" versus "receiver-initiated" cache management [24].)

D. The Problem, Revisited

Given the discussion of the many issues implicit in determining the optimal X, it is important to state which issues are addressed in this paper. We assume, as indicated above, that sites only make cache replication decisions about "read-only" objects. For the most part (except for the dynamic greedy algorithm), we also assume that sites have perfect information about object access rates. Because of these assumptions, the dimensions of *replication eligibility* and *local object value* do not pose especially difficult problems. However, because autonomy can be very important in an RCA environment, we focus on its implications for the dimensions of *cost function* and *degree of coordination*. In other words, we examine the problem of how performance is affected by 1) different cost functions that sites can use and 2) the different ways that sites can coordinate their decisions with one another.

In the next section, we describe a set of cache allocation policies that use a variety of cost functions and have different degrees of coordination. Implementations of these policies, and their performance, are studied in Sections IV and V, respectively.

[3]Note that, for simplicity of exposition, we assume that all objects have the same size so that the value does not need to be scaled by the amount of cache storage the object occupies. This is not a serious restriction since the algorithms can be modified to take varying size objects into account.

III. RCA Cache Management Strategies

Cache management requires that a system first determine which objects are eligible for caching, and then, when cache storage is full, determine which object should be swapped out to make room for an incoming object. As discussed in Section II-A, this paper focuses mainly on the eligibility issue and assumes that eligible objects are always available in cache. This is achieved by simply limiting the number of eligible objects to the amount of cache storage available. In other words, the hit ratio is always 1, so that the term $H_{ij}(X)$ in (2)–(4) is either 0 or 1. (The performance of a dynamic greedy algorithm, which uses the classic LRU replacement algorithm, is also examined.)

We investigate three classes of policies: *centralized*, *distributed*, and *isolationist* policies. These classes are distinguished by the amount of remote caching information that is used when making cache decisions. At one extreme, sites operating with isolationist policies make decisions in complete ignorance of the decisions made by other sites. At the other extreme, sites that centralize the decision-making for the entire system operate with optimal policies, because they make decisions in complete coordination with other sites. In distributed policies, sites make decisions independently of other sites, but also utilize information about previous decisions made by other sites.

Within a single class—e.g., optimal policies (that make decisions in coordinated fashion)—policies can differ based on the *performance goal*. Performance goals affect an object's eligibility for caching. The goal of the first optimal strategy is fairness: although sites want to achieve good overall performance, they insist that no site should suffer "unduly" in achieving such performance. The goal of the second strategy is simply to maximize overall (average) performance, without regard to how individual sites will do under a given object allocation. These strategies are both examples of global optimization polices. In contrast, we examine both a local and global performance policy in the class of distributed policies.

Each of the optimal strategies shares the following assumptions. First, all sites have complete knowledge about the access patterns at every other site. Second, eligibility decisions are completely coordinated, so that at any moment

the best decision is always made. Clearly, these assumptions are not realistic in real-world RCA environments. We are interested in these strategies because they give an upper-bound on RCA performance. The aggregate resources in an RCA are very large: it is important to determine the best performance that can be achieved through efficient use of system resources. The performance of the isolationist policy shows how important some degree of inter-site cooperation is for RCA efficiency. This policy does not completely ignore the benefits of the RCA architecture because sites *will* respond to requests from other sites—if the object is resident in cache. The point is that sites do not have (or ignore) information about remote cache contents that could guide local decisions about object eligibility. By investigating the differences between the optimal and greedy strategies, we gain insight into the problem of devising caching policies that can operate in *distributed* fashion. Our goal is to develop distributed strategies with performance between the optimal and greedy strategies. (In addition, the distributed algorithms are more adaptable to changes in system state than the optimal. Because the decisions are localized rather than centralized, sites make fewer changes to the cache configuration after detecting that, for example, access frequencies have changed.) Although these strategies require cooperation among sites, we believe that this requirement is not necessarily a violation of site autonomy.

A. Optimal Strategies

Equations (1)–(3) show that the cost of accessing a *single* object at a given site depends on the probability of 1) the object being in local cache and 2) the probability of the object being in at least *one* other site's cache. Let P_{ij} denote the probability of read-access for object i at site j. Then $\sum_{i=1}^{M} c_{ij}P_{ij}$ represents the overall time needed to access objects at site j. Given the access probability distribution for all objects i at all sites j, two cost functions can be specified to evaluate the performance of a given configuration X. (These definitions of the cost functions assume that sites have equal amounts of activity. Section V-G discusses the effect of different degrees of site activity.)

$$Cost(X) = max_j \sum_{i=1}^{M} c_{ij}P_{ij} \tag{4}$$

$$Cost(X) = \frac{\sum_{j=1}^{N}\sum_{i=1}^{M} c_{ij}P_{ij}}{N}. \tag{5}$$

A policy that determines a configuration which minimizes (4) implies that the "worth" of the overall system is no better than the performance of the worst-performing site in the system. The goal of this policy is to use the combined resources of the sites in the RCA efficiently, and at the same time ensure "fairness." If sites insist on overall "fairness" criteria, then other strategies cannot surpass the performance of the optimal fair policy. A policy that determines a configuration X which minimizes (5) corresponds to solving the basic file allocation problem (FAP) because it minimizes average (overall system) response time, without allowing individual sites to impose any specific constraints on object allocation. If site autonomy is not an issue, then other strategies cannot surpass the RCA performance achieved by the optimal "average" policy.

B. Distributed Strategies

The key feature of the distributed strategies is that sites do *not* make decisions in a completely coordinated fashion, but *do* use information about other sites when making local decisions. Recall from Section II-A that c_{ij} is the cost of accessing object i at site j, and is composed of the costs of accessing each of the memories in the storage hierarchy, weighted by the probabilities of needing to access a given memory. Assume that site $k \in$ sites$\{1, \cdots, N\}$ is making caching decisions. If site k turns object i "on," this has two effects. First, there is a local effect because site k now has object i in the fastest media. Second, there is a global effect because object i is now available to all the other sites in the second fastest media. Note that while the first effect always improves performance, the second effect will tend not to effect performance if some other site j has already cached object i.[4] If site k turns the object "off" then the magnitude of the first effect depends on whether site k can already access the object through remote memory at some other site. Even if site k has remote access, performance will always get worse because the object is no longer available in local main-memory. Other sites will only suffer if the copy maintained at site k was the only replica.

If a strategy is concerned with local optimization, then only the first effect is relevant. If a strategy does global optimization, then both effects are important. Because we are dealing with read-only objects, the *marginal value*, m_{ik}, of caching object i can be calculated independently of cache decisions regarding other objects.[5] The value, m_{ik}, is always positive and is based on the difference between the (local or overall) access time for object i when it is cached at site k and the (local or overall) access time for the object when it is not cached. Under either type of strategy, the magnitude of the marginal benefit depends on 1) the "local" importance of the object and 2) the presence of the object in some other site's memory. The value of m_{ik} must, of course, be weighted by the (local or overall) probability of accessing the object. Notice that these strategies differ from the greedy strategy described below in that an object which, from a purely local context, is valuable, will have less value when the site realizes that the object is cached in remote memory. Under the local distributed strategy, sites also operate in a greedy fashion, but the marginal value calculations factor in information about the state of other sites' caches.

The strategies discussed here make cache decisions in synchronous fashion, and then broadcast the results of the

[4] If site k is closer (or faster) than site j to other sites in the system, then there *will* be a global effect when site k also caches the object. This effect, however, is minimal compared to the local effect. More importantly, in this paper we examine RCA performance in the context of a LAN environment. In consequence, remote sites are "symmetric" in the sense that all sites are equally distant from one another.

[5] We use the term "marginal value" to emphasize the fact that we are not examining the *net* effect of caching one object while swapping out another object. We examine only the performance benefit that results from caching the object. The constraint of having only a given amount of cache available is factored in later (see Section IV-B).

decisions to other sites. As a result, when site k makes local cache decisions, the information that determines marginal benefit is up-to-date. The key point here is that a single, hard problem is partitioned into N smaller problems. Instead of the overall system trying to solve the problem in centralized fashion, each site tries to minimize the given cost function on its own. Of course, the composition of N individually optimal pieces may be suboptimal. The hope is that the distributed solution will not differ greatly from the optimal solution.

C. Isolationist Strategies

1) Static Greedy Strategy: The static greedy strategy represents an extreme of autonomous ("isolationist") behavior in an RCA: essentially, each site ignores all other sites' caching decisions when making its own caching decisions. Each site simply caches the objects that maximize the percentage of access probability distribution available in its own cache. The greedy algorithm can result in much system replication, as heavily accessed objects are replicated at each site. On the other hand, each site is guaranteed to get its most heavily accessed objects with minimal cost. In contrast, under the optimal strategies *and* under the distributed strategies, sites are aware of other site's decisions. Under the distributed local policy a site may well cache a relatively unimportant object because it relies on other sites for access to more valuable objects.

2) Dynamic Greedy Strategy: The final strategy examined in this paper uses no knowledge of P_{ij} at all, but otherwise resembles the static greedy strategy in that sites simply cache (what they perceive to be) their most valuable objects. It is a "greedy" strategy in that sites do not attempt to avoid caching highly replicated objects by snooping on other sites. Essentially, sites assign LRU-based values to objects, and use these values to determine which objects should be swapped out to make room for incoming objects.

D. Performance Issues Between the Strategies

Before discussing the implementation of the strategies, we summarize the issues that differentiate them. Three degrees of coordinated decisions (through exchange of state information) are examined: *centralized*, *distributed*, and *isolationist*. Three performance goals are examined: maximizing *average*, *fair*, and *local* performance. Two ways of assigning local object values are examined: *exact access frequency* and *LRU-based*.[6] Table I lists, for each strategy discussed in this paper, where the strategy lies on the spectrum.

TABLE I
DIFFERENCES AMONG THE STRATEGIES

	RCA Performance Dimensions		
Strategies	Coordination	Goal	Object Value
Optimal Average	centralized	average	access frequency
Optimal Fair	centralized	fair	access frequency
Distributed Local	distributed	local	access frequency
Distributed Global	distributed	average	access frequency
Static Greedy	isolationist	local	access frequency
Dynamic Greedy	isolationist	local	LRU stack position

IV. STRATEGY IMPLEMENTATIONS

In this section, we develop implementations of the RCA cache strategies. Solutions for the optimal object configurations can be determined analytically; the solution approach is detailed in Section IV-A. The configurations for the distributed strategies are obtained through event driven simulations of the algorithms described in Section IV-B. The dynamic greedy strategy is also evaluated through simulations. For the static greedy strategy, determining the configuration is trivial: the most frequently accessed objects at each site are cached in its memory.

A. Implementing the Optimal Strategies

The problem of determining the optimal RCA configuration given by (6) involves a nonlinear binary programming problem. We want to solve for minimal

$$\sum_j \sum_i c_{ij} X_{ij} \tag{6}$$

where c_{ij} is a nonlinear function of X_{ij} [note the product term in (2)–(3)]. The X_{ij}, of course, are binary valued. We now show that this problem is, in fact, reducible to the capacitated transportation problem [15]. As a result, the optimal solution can be determined fairly easily [3]. First we define an objective function that differs from the RCA function by a constant, so that the optimal solutions are identical. Then we show that the optimal solution to the new function will necessarily meet the constraints of the RCA problem.

Recall that there are M objects and N sites. Construct an augmented M by $N + 1$ matrix, Z, in the following way. The entries in the first N columns are P_{ij} $(t_r - t_l)$, and the entries in the last column are $\{\sum_j \sum_i P_{ij}\}(t_r - t_d)$. (Here $i = 1, \cdots, M$, $j = 1, \cdots, N$.) The P_{ij} terms are the (normalized) object access rates discussed above. The t_l, t_r, and t_d terms denote the access time for local memory, remote memory, and disk, respectively. Consider the optimization problem of determining the maximum value of $\sum_k \sum_i Z_{ik} X_{ik}$ $(k = 1, \cdots, N + 1)$ subject to the constraints that

1) $\sum_k X_{ik} \geq 1$.
2) $\sum_i X_{ik} \leq B$ (B = site cache size) for $k \leq N$.
3) $\sum_i X_{ik} < \infty$ (for k = $N + 1$).
4) X_{ik} is binary.

Constraints 1–4 correspond exactly to the model of the capacitated transportation problem [15]. The first is a row constraint; the second and third are column constraints. We first show that this objective function differs from the RCA

[6] Although most of the algorithms analyzed here make use of *a priori* knowledge of object access rates, this does not imply that the algorithms are not "implementable." In practice, sites would periodically do dynamic estimation of the access frequencies based on previous history (e.g., through exponential weighting). After some period of time sites would do a fresh determination of the optimal configuration. Because we do not consider that dynamic frequency estimation imposes any difficulty (as opposed to the use of exact knowledge), we chose to reduce simulation time by using static knowledge.

function by a constant. The first N columns represent an initial state in which all objects can be accessed, by all sites, from remote memory. The idea is that we then solve for maximum incremental benefit from the initial state. The last column allows us to model the situation of having to access an object from disk (because of memory constraints). An $X_{ik} = 1$ entry in the last column means that no copy of the ith object is in main-memory. As a result, sites have to pay the incremental (with respect to remote memory access) cost of accessing the object on disk.

We now show that the optimal solution to the transportation problem is also the optimal solution to the RCA problem. Note that the optimal transportation solution will never have both an $X_{ik} = 1$ for some $k \leq N$ *and* also $X_{i[N+1]} = 1$, since entries in the last column represent "negative" benefit. Also, because *all* elements in the first N columns are positive, the sum of each column $\leq N$ (in the optimal solution) will equal B. The optimal solution will therefore correspond to the optimal solution for the RCA. First, sites do not access an object on disk if it is present in main-memory. Second, each site uses all of its available main-memory for cache storage in the RCA.

B. Implementing the Distributed Strategies

The distributed strategies were discussed in Section III-C. Under the FAP constraint of $H_{ij}(X) = 1$ for all "on" objects, the equations needed for calculating marginal value have a very simple form. Let $R_{ij} = 1$ iff there is a remote (with respect to site j) replica of object i (0 otherwise). Say that site k is making a caching decision with respect to object i. Under the local optimization strategy, the local marginal value is

$$m_{ik} = P_{ik}((t_r R_{ik}) + (t_d(1 - R_{ik})) - t_l). \tag{7}$$

This follows because site k is only concerned with the *local* effect of caching object i. Under a global optimization strategy, site k is also concerned with the overall system improvement that results from the caching decision.

The marginal value of site k caching object i for *another site* $j \neq k$ is

$$\begin{cases} 0 & \text{if } R_{ij} = 1, \text{ and} \\ P_{ij}(t_d - t_r) & \text{otherwise.} \end{cases} \tag{8}$$

Then, under the global optimization policy, the overall value of site k caching object i is

$$m_{ik} = P_{ik}((t_r R_{ik}) + (t_d(1 - R_{ik})) - t_l) + \sum_{j=1, j\neq k}^{N} (1 - R_{ij}) P_{ij}(t_d - t_r). \tag{9}$$

Each of the distributed algorithms contains the following steps. When a site k makes a set of caching decisions, it has a snapshot X which gives the state of the *other* sites in the system at a given time. X is the same eligibility matrix discussed in Section II-A—except that the cache contents of site k is uninitialized. Site k then uses a greedy algorithm which proceeds as follows.

1) For all objects i, site k calculates m_{ik}.
2) Site k orders the objects by decreasing m_{ik}.
3) For cache size B, site k then simply sets $X_{ik} = 1$ for the first B objects, and sets $X_{ik} = 0$ for all other objects.

The distributed algorithms are implemented in a detailed discrete event simulation. In the simulation, each site maintains a table of its cache contents. One simulation module implements the cache management algorithms, while the other one generates the object requests and determines whether a local hit or remote hit occurs. In the simulation, the distributed algorithms execute the cache eligibility algorithm periodically: we found that the rate of convergence was very rapid (see Section V-F). (The simulation does not "charge" when sites swap objects in and out of memory. Again, this is to facilitate comparison with the optimal algorithms, which do not change the configuration after it is initially determined. Because the distributed algorithms converge rapidly to an "optimal" configuration, the cost of adjusting to dynamic changes in the object access frequencies would be small.)

C. Implementing the Isolationist Strategies

Under the isolationist strategies, a site simply caches the objects that maximize the percentage of access probability distribution available in its own cache. The static strategy does this trivially because information about the P_{ij} is available. The dynamic greedy strategy is also implemented in the detailed discrete event simulation. In the simulation module implementing the cache management algorithms, the dynamic greedy strategy simulates an LRU replacement algorithm for each site independently.

V. Performance Analysis

In this section, we compare the different RCA cache strategies based on their performance in various system configurations. Important system dimensions are varied, while basic system characteristics are held constant in this analysis. In Table II we show the constant system parameters of a remote caching architecture implemented in a workstation environment.[7] We are not that concerned with the exact values in Table II because the benefits of using an RCA apply over a wide range of system parameters and access frequencies [23]. The key performance characteristic of the system is that order-of-magnitude differences in access speed exist between layers of the memory hierarchy. The number of objects is kept small to facilitate generation and analysis of the results. In Section V-F we show that the distributed algorithms scale to a system that is at least two order of magnitude larger.

[7] The values of the access time parameters assume the following. The RCA sites are connected by an Ethernet network, and the generic "object" is a packet on the order of 500 bytes. Raw bus time is then approximately 0.5 ms. Disk access consists of one third of the end-to-end seek time plus one half of the rotational latency. Sites make one disk access to retrieve an object's index, and make another to access the object itself. With a slow file system adding file system overhead, we use the round figure of 50 ms per object "access." (Objects are assumed to be partitioned at the disk level; the cost for disk access reflects average object time.) Local main memory access time*very* conservatively takes 1 ms (including hash-table access followed by a 500 byte copy). The RPC time is taken from [27]. Given these "raw" access times, the memory hierarchy access times are calculated as described in Section II.

TABLE II
CONSTANT PARAMETERS IN THE RCA ANALYSIS

System Parameters	Parameter Values
Number of Sites	10
Number of Objects	1000
Local Main-Memory Access Time	1 ms
Remote Main-Memory Access Time	6 ms
Disk Access Time	63 ms

TABLE III
NUMBER OF OBJECTS COMPRISING THE HOT-SET

Hot-Set Curves	25th percentile	50th percentile
$\theta = 0.1$	3 objects	7 objects
$\theta = 0.05$	6 objects	14 objects
$\theta = 0.001$	25 objects	49 objects

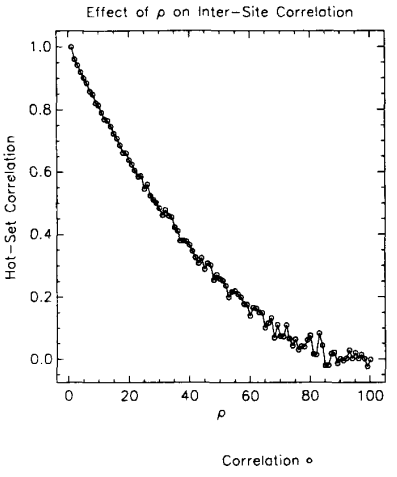

Fig. 1. Effect of ρ (100 objects).

A. Performance Parameters

1) Hot-Set Parameter: In order to study the effect of different access probability distributions on performance, we introduce a hot-set parameter θ that models data skew and variability. The hot-set parameter determines the access probability distribution for the data objects. If objects are ordered by decreasing access frequency, access frequency can be graphed on the "y-axis" against object identifier on the "x-axis." We refer to the resulting (monotonic decreasing) curve as a *hot-set curve*, because it shows the objects that are accessed most often at a site. When the distribution curve is "steep," then fewer objects comprise the hot-set. When the curve is flat, then many objects comprise the hot-set. Intuitively, caching becomes less and less effective as the hot-set curve becomes flatter, because more objects must be cached in order to maintain a given cache-hit ratio. Let P_{ij} denote the probability of site j accessing object i. Then, when sites have identical hot-sets, we model a site's hot-set curve by setting

$$P_{ij} = \frac{e^{-\theta i}}{T} \tag{10}$$

where i is the object number, θ is the hot-set parameter, M is the total number of objects, and $T = \sum_{i=1}^{M} e^{-\theta i}$ is a normalization constant. In other words, the hot-set curve is a normalized negative exponential distribution for the specified θ. Thus, smaller θ values represent flatter hot sets. Hot-set curves generated with a given θ maintain their *shape* for all values of M: the precise P_{ij} will of course depend on the number of objects in the system. In Table III we show the minimum number of objects that are needed to cache 25% and 50% of the access distribution for various hot-sets (when the total number of objects is 100).

2) Correlation of Site Hot-Set Curves: Equation (7) assumes that all sites have the same hot-set curve—i.e., all sites access a given object with the same frequency. In modeling the situation where sites have different hot-set curves, we do not change the value of θ: what varies is that the objects comprising the hot-sets are different. Intuitively, we want to capture the degree of overlap or *correlation* between hot-sets with a single parameter. To do so, we follow [32]. Note that the M objects in the system can be ordered, in descending order, by access frequency. Hot-set correlation is described by a single parameter ρ that takes on integer values between 1 and M. Consider object 1, the most frequently accessed object at site 1. Object 1 occupies position 1 (i.e., most valuable) relative to all other objects. At all other sites, object 1 occupies a randomly chosen position between 1 and ρ. More generally, object i of the original curve is placed in a randomly chosen position from 1 to MIN($\rho + i - 1$, M) except that the position occupied by a previous step is not allowed. The larger the ρ, the smaller the correlation between sites. Thus $\rho = 1$ corresponds to perfect correlation among the hot-set curves; $\rho = M$ corresponds to a random relationship between the hot-set curves. To get an idea of how ρ affects the inter-correlation of sites' hot-set curves, examine Fig. 1. The average Spearman correlation (used because of its robustness compared to Pearson's r) [12] of the hot-set *contents* for sites $2, \cdots, 10$ with that of site 1, is shown for ρ values $1, \cdots, 100$.

3) Relative Site Activity: Although θ determines the *shape* of a site's hot-set curve, because P_{ij} is normalized to sum to 1.0, it cannot model differences in *relative* site activity. Picture a situation where site 1 has five times the activity of the other sites in the system. A global optimization policy will weight the needs of site 1 more heavily than the needs of the other sites. In contrast, under a local optimization policy, sites will not take the fact of different degrees of site activity into account when making cache decisions. As a result, overall system performance will be relatively worse as compared to a situation with equal degrees of site activity.

Let η denote the parameter that determines relative site activity. The activity of each site is given by $a_j = 1/(A * j^{1.0-\eta})$ where $A = \sum_{j=1}^{N} 1/j^{1.0-\eta}$. In other words, the relative site-activity distribution has a "Zipf-like" shape [16], controlled by the value of η. When $\eta = 1.0$, then sites have the same amount of activity. When η is 0.0 (and $N = 10$) then the relative site activity is $\{1.00, 0.50, 0.33, 0.25, 0.20, 0.17, 0.14, 0.12, 0.11, 0.10\}$. Individual site mean response time is $\sum_{i=1}^{M} c_{ij} P_{ij}$, but overall

(system) mean response time is $\sum_{j=1}^{N} a_j \sum_{i=1}^{M} c_{ij} P_{ij}$. We look at the performance implications of three values of η: 1.0, 0.5, and 0.0.

B. Performance Statistics

We report algorithm performance in two ways. First, the performance of the optimal average algorithm is graphically shown as a function of values of θ, ρ, and cache size. This algorithm serves as a baseline for the "best" possible results for a given RCA configuration. The performance of the other algorithms is shown *relative* to that of the optimal average algorithm. These figures allow us to get a feel for the RCA "state space." Second, we take a closer look at algorithm behavior by presenting tables of statistics for the performance of a small slice of the state space. Per-site cache size (B) is held constant at 5% of the total number of objects. In order to "normalize" values of θ across any number of objects, we use $STORE_{MAX}$, defined to be $\sum_{i=1}^{B} P_{ij}$ when the P_{ij} are sorted in decreasing order. That is, $STORE_{MAX}$ is the maximum percent of a site's hot-set that can be stored locally by that site. Values of θ are adjusted so that, if a site simply caches objects in static greedy fashion, it can cache $STORE_{MAX}$ percent of its hot-set. We report two sets of statistics: the first gives a sense of "absolute" performance, the second gives a sense of "how" the algorithm achieves its performance.

- *RTime* (Response Time) is the basic performance metric used to judge the effectiveness of a given algorithm. We report mean object response (i.e., access) time.
- *STDev* (Standard Deviation) is the standard deviation of *RTime* over the sites in the system. Certain algorithms may offer good overall performance, but at the cost of large site-to-site variations.
- C_{HIT} (Cache Hit) is the percent of object requests that were met by either local or remote cache. Effective policies will get the most frequently accessed objects into the faster local/remote main-memory media, and therefore have high C_{HIT} values. We report the mean value over all sites.
- $REPL$ (Degree of Replication) is the number of object replicas stored under a given configuration. Assuming that *at least* one copy of an object is resident in system cache, we report the mean number of object replicas. In the case of the static algorithms, only one system configuration needs to be evaluated. In the case of the distributed algorithms, this statistic (as well as that of $BNFT$) is the "mean of the means" under all system configurations generated by the algorithm.
- $BNFT$ (Benefit) is the average benefit that each cache slot gives the system. Let $s_1 = t_d - t_l$ and $s_2 = t_d - t_r$. Then $s_1 * P_{ij}$ is the (weighted) benefit that a site gets from caching an object locally, and $s_2 * P_{ij}$ is the (weighted) benefit that remote sites get from the presence of the replica in remote memory. Let there be n_1 object replicas. If n_1 is at least 1, then $n_2 = N - n_1$ sites can access the object through remote memory. (The presence of multiple copies does not increase the remote benefit.) Then the benefit that the overall system gets from the cache slots devoted to that object is $n_1 * s_1 + n_2 * s_2$. (P_{ij} is used to weight the benefit that a given site actually gets from the cached object.) We report the benefit averaged over all cached objects. Intuitively, an efficient strategy will have high $BNFT$ values because its eligibility criteria will tend to reflect the need to 1) use cache storage most effectively by caching the most important objects, and 2) limit the amount of replication of a given object.

In the performance tables (Tables IV–XII) four values are examined for θ (those generating the four $STORE_{MAX}$ values), and four for ρ (1, 10, 50, and 100). These values represent two extreme points for the parameter in addition to two intermediate points. The total number of objects in the system is 100, so that when ρ = 100, the sites have only "random" correlation among their hot-sets. Per-site cache size is held constant at 5% of the total number of objects.

In the performance figures, per-site cache sizes vary from 1% to 10% of the total number of objects. Values of θ vary from 0.001 to 0.082 in increments of 0.009, and values of ρ vary from 1 to 96 in increments of 5.

The performance of the optimal and the static greedy algorithms is derived analytically, while that of the distributed and the dynamic greedy algorithms is from simulations. Because the distributed algorithms use exact knowledge of object access rates, their simulation did not require a "warm up" period. The dynamic greedy algorithm had a warm up period of 1000 object accesses. Each simulation ran for 10 000 object accesses. Preliminary results showed that the performance statistics reported from this single large run were indistinguishable from "batched means" simulations that used a stopping criterion of a relative half width of 0.1 and a 95% confidence interval. (The reason for such behavior is the relative lack of "noise" in the simulations.) We therefore used single-batch simulations to generate the data points for the figures.

C. Optimal Average Performance

When sites have no cache storage at all, then mean response time will be 63 ms (see Table II). If all objects are cached locally, then mean response time will be 1 ms; if all objects are available in either local or remote memory, then mean response time will be between 1 and 6 ms. To the extent that sites must access objects at disk, response time will, of course, exceed main-memory access times.

In Fig. 2, the performance of the optimal average algorithm is shown as a function of θ and cache size. The larger the value of θ (i.e., the smaller or sharper the hot set), the fewer objects need to be cached in order to attain a given cache-hit ratio. Consequently, the larger the θ, the better the performance. The shape of the performance curve mirrors the access distribution. Thus, when the hot-set curve is flat (θ = 0.001), performance is a linear function of cache size. When the hot-set curve is sharp (θ = 0.1), performance is a nonlinear function of cache size, because 50% of read accesses are to only 7% of the objects.

Fig. 3 shows mean access time (ms) as a function of both θ and ρ (per-site cache size is maintained at 2% of the number of objects). As before, when other factors are held constant, performance improves with larger θ. As ρ increases from 1

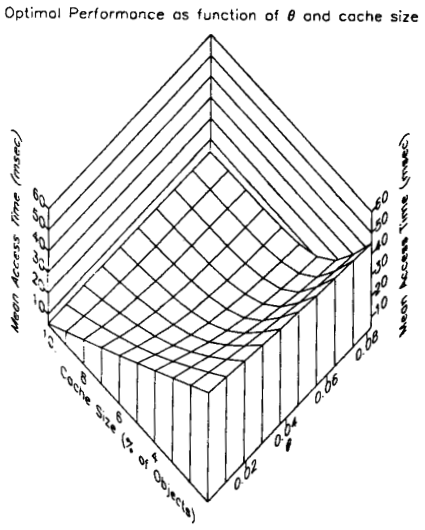

Fig. 2. Optimal performance (ρ = 26).

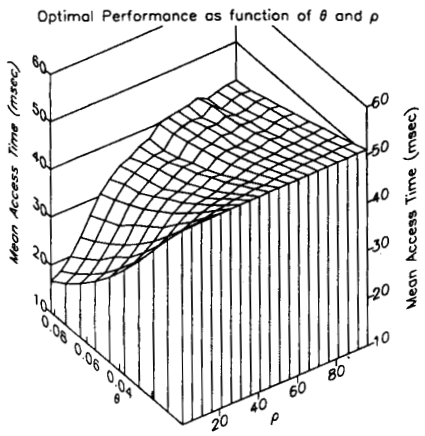

Fig. 3. Optimal performance (C_{SIZE} = 2%).

to M, the system is in one of three configurations: sites have a) identical hot-sets, b) partial hot-set overlap, or c) random hot-set relationship. In configuration a) performance is best, because the smaller system-wide hot set implies that caching is most effective. In configuration c), performance is worst, because the system hot set is large. In moving from a) to c), performance tends to degrade—but there are dips and valleys. Even if there is slightly less overlap among the sites' hot-sets, sites can get a higher *local* hit-ratio, and therefore improve performance slightly.

D. Optimal Fair Performance

Only the optimal average algorithm (henceforth "average") explicitly optimizes for the performance metric; the optimal fair (henceforth "fair") algorithm can therefore never exceed the former's performance. Figs. 4 and 5 show the relative performance of the algorithm compared to average. We show *relative* mean access time: i.e., the ratio of access time under

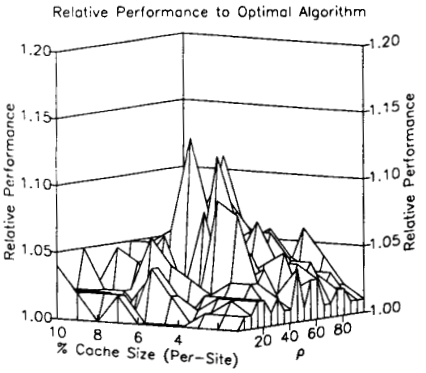

Fig. 4. Optimal fair algorithm (θ = 0.082).

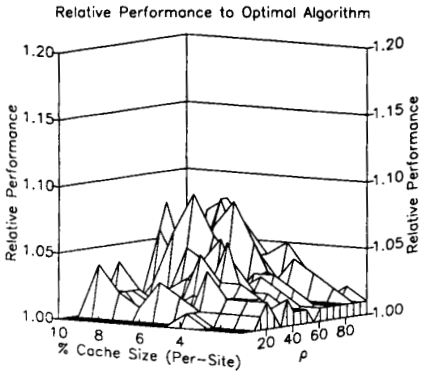

Fig. 5. Optimal fair algorithm (θ = 0.046).

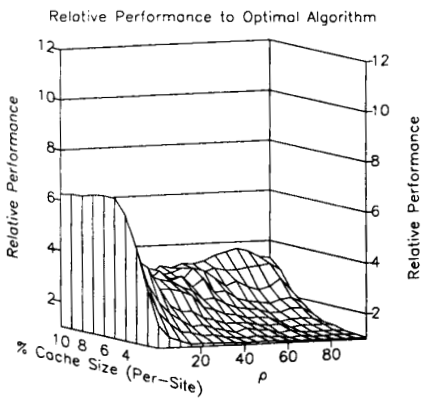

Fig. 6. Static greedy algorithm (θ = 0.082).

the fair strategy to the access time under the average strategy. (Similar ratios are reported in Figs. 6–15.) These figures are quite "jagged" because the performance range among sites depends on the exact composition of their hot-sets and on the amount of cache storage in the RCA system. Because of the randomness associated with ρ, the relative performance of the fair (a "min/max") algorithm can vary a great deal despite small changes in system configuration. Nevertheless, certain trends are quite clear.

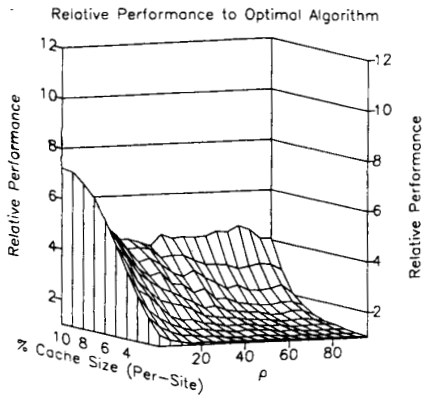

Fig. 7. Static greedy algorithm (θ = 0.046).

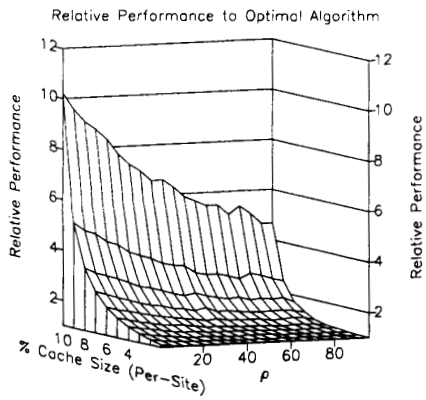

Fig. 8. Static greedy algorithm (θ = 0.001).

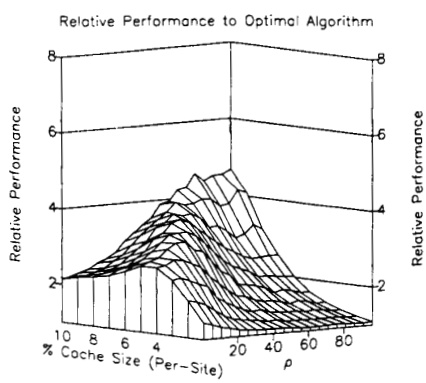

Fig. 9. Dynamic greedy algorithm (θ = 0.082).

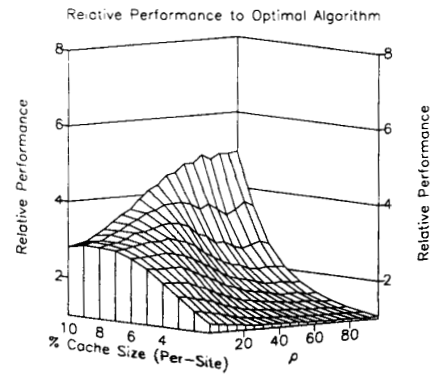

Fig. 10. Dynamic greedy algorithm (θ = 0.046).

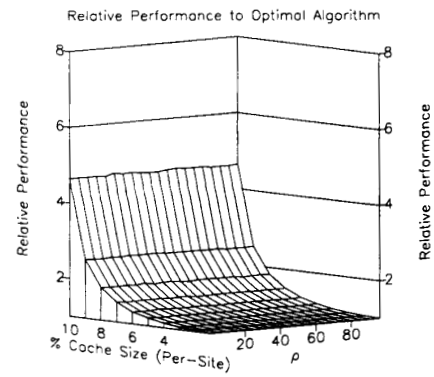

Fig. 11. Dynamic greedy algorithm (θ = 0.001).

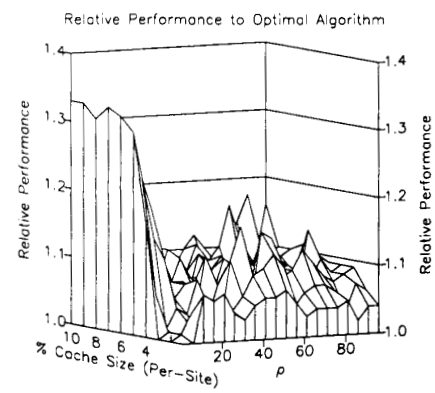

Fig. 12. Distributed local algorithm (θ = 0.082).

The performance gap increases for larger values of θ, and the reason for this involves the fairness criterion. When θ is large, then the few objects in the hot-set are valuable and other objects are not. On the one hand, if sites replicate locally valuable objects, then cache storage is wasted because *other* objects could be stored locally, with access to the valuable objects being through fast remote memory. (Performance of the isolationist algorithm suffers precisely because too many replicas are made of valuable objects.) On the other hand, if there are too few replicas of valuable objects, performance suffers because local memory access is still faster than remote memory access. The optimal algorithm caches the optimal number of valuable objects at some sites and caches less valuable objects at other sites. Sites with valuable objects do better than other sites. Overall RCA

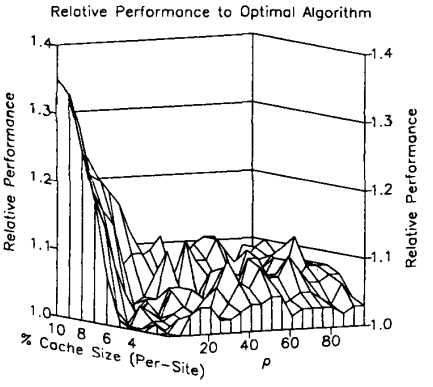

Fig. 13. Distributed local algorithm (θ = 0.046).

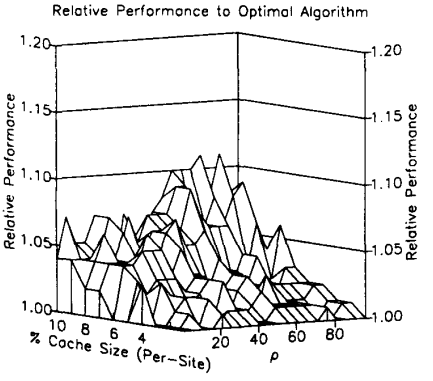

Fig. 14. Distributed global algorithm (θ = 0.082).

performance is optimized because *all* sites benefit from remote memory. Such a configuration is not always "good" for the fair algorithm because sites responsible for caching less valuable objects do *individually* worse than those caching more valuable objects. Because the fair policy optimizes the performance of the *worst* performing site, the improved overall solution is ignored in favor of a solution in which all sites have good (but not optimal) performance. As hot-sets become flatter (Fig. 5), cache decisions about a given object have less effect on performance because objects are individually less valuable. This leads to smaller "min *versus* max" performance gaps, and fair performance approaches that of the average. At the limit of flat hot-sets, the performance of the two algorithms is identical. Table IV shows that for a given amount of cache, the difference between the optimal and fair algorithms can decrease—even though the hot-set curve becomes flatter (compare the "20%" and "30%" entries for ρ = 50 and 100). The reason for this is that, for a given amount of storage, the system can cache the global hot-set while still maintaining a fair policy. Overall, however, the largest performance differences between the algorithms occur with a sharper hot-set curve.

The fairness criterion also tends to cause the fair algorithm to do relatively worse (in general) as hot-sets overlap less and less. When hot-sets are closely related there are relatively few "good" configurations because objects, in general, give approximately the same benefit to all sites. As correlation decreases, there are many more configurations that the average algorithm can exploit, because caching an object at one site has different implications than caching it at another site. Only a subset of these configurations "make sense" for the fair algorithm, because while some sites will benefit from the cached object, other sites get little benefit.

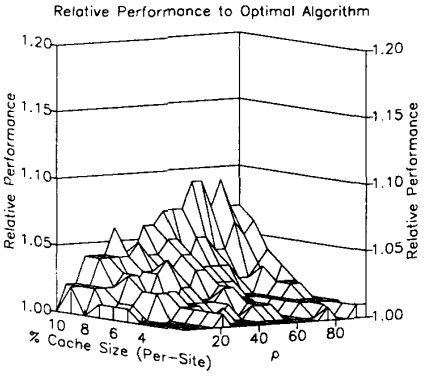

Fig. 15. Distributed global algorithm (θ = 0.046).

The *SDev* values shown in Table IV show that the fair algorithm succeeds in ensuring that no site does badly. The large difference between the *SDev* of the algorithms for large θ and ρ shows that the average algorithm gets its performance by requiring some sites to cache objects primarily for the benefit of other sites. Moreover the C_{HIT} values (representing local *and* remote cache hits) show that the fair algorithm is forced to leave certain objects on disk—because caching them at *any* site would cause the performance spread among sites to become too wide. It is interesting to note that in Table V (for per site cache storage of 5% of total number of objects) the average and fair policies maintain only a *single* object replica for all values of θ and ρ considered. The *benefit* per cached object, however, is slightly greater for the average than for the fair when the correlation is less. In other words, even though the coordinated decisions of the fair algorithm result in no "replication," it does not cache the same amount of "value" because this would not satisfy the fairness criterion. Note that the two algorithms get the same amount of benefit when sites have flat hot-sets. Even when sites have relatively sharp hot-sets ($STORE_{MAX}$ = 30%), the algorithms still have the same $BNFT$ when sites have strong correlation. Only when correlation is low *and* sites have sharp hot-sets does the performance of the two algorithms diverge.

E. Performance of the Isolationist Strategies

Figs. 6–11 show the performance (relative to the optimal) of the two isolationist (or "greedy") algorithms. The static and dynamic greedy algorithms have features in common, but also differ significantly in their behavior.

First we examine the behavior of the static greedy algorithm (Figs. 6–11). When hot-sets are identical or closely overlap, relative performance is quite bad. As hot-sets diverge more,

TABLE IV
PERFORMANCE OF THE OPTIMAL AND STATIC GREEDY ALGORITHMS (PER-SITE CACHE STORAGE HOLDS 5% OF OBJECTS)

		Optimal Average			Optimal Fair			Static Greedy		
$STORE_{MAX}$	ρ	RTime	SDev	C_{HIT}	RTime	SDev	C_{HIT}	RTime	SDev	C_{HIT}
30%	1	7.1	0.4	0.972	7.1	0.3	0.972	44.1	0.0	0.300
30%	10	7.1	0.3	0.967	7.1	0.2	0.966	29.4	2.6	0.560
30%	50	14.2	2.1	0.833	14.3	1.6	0.833	24.9	3.15	0.639
30%	100	20.8	2.1	0.716	21.4	0.8	0.707	27.9	4.0	0.587
20%	1	11.2	0.2	0.900	11.2	0.3	0.900	50.3	0.0	0.200
20%	10	11.5	0.3	0.891	11.5	0.1	0.891	37.7	1.6	0.422
20%	50	19.6	2.1	0.743	20.2	0.8	0.734	30.6	2.9	0.547
20%	100	24.9	1.8	0.652	25.7	1.0	0.639	32.2	3.1	0.520
10%	1	22.5	0.0	0.702	22.5	0.1	0.702	56.4	0.0	0.100
10%	10	22.9	0.2	0.695	22.9	0.1	0.695	48.1	0.5	0.247
10%	50	28.0	1.5	0.603	28.1	1.3	0.601	37.7	1.7	0.431
10%	100	30.3	1.5	0.563	30.6	0.1	0.557	36.7	1.4	0.449
5%	1	34.0	0.0	0.500	34.0	0.0	0.500	59.5	0.0	0.050
5%	10	34.0	0.0	0.500	34.0	0.0	0.500	54.4	0.0	0.140
5%	50	34.0	0.0	0.500	34.0	0.0	0.500	42.0	0.0	0.360
5%	100	34.0	0.0	0.500	34.0	0.0	0.500	39.1	0.0	0.410

TABLE V
REPLICATION DONE BY THE OPTIMAL AND STATIC GREEDY ALGORITHMS (PER-SITE CACHE STORAGE HOLDS 5% OF OBJECTS)

		Optimal Average		Optimal Fair		Static Greedy	
$STORE_{MAX}$	ρ	*REPL*	*BNFT*	*REPL*	*BNFT*	*REPL*	*BNFT*
30%	1	1.0	11.1	1.0	11.1	10.0	3.7
30%	10	1.0	11.1	1.0	11.1	3.6	6.6
30%	50	1.0	9.7	1.0	9.6	1.4	7.5
30%	100	1.0	8.4	1.0	8.2	1.2	6.9
20%	1	1.0	10.3	1.0	10.3	10.0	2.5
20%	10	1.0	10.2	1.0	10.2	3.6	4.8
20%	50	1.0	8.6	1.0	8.5	1.4	6.4
20%	100	1.0	7.6	1.0	7.4	1.2	6.1
10%	1	1.0	8.0	1.0	8.0	10.0	1.2
10%	10	1.0	7.9	1.0	7.9	3.6	2.9
10%	50	1.0	6.9	1.0	6.9	1.4	5.0
10%	100	1.0	6.5	1.0	6.4	1.2	5.2
5%	1	1.0	5.7	1.0	5.7	10.0	0.6
5%	10	1.0	5.7	1.0	5.7	3.6	1.6
5%	50	1.0	5.7	1.0	5.7	1.4	4.1
5%	100	1.0	5.7	1.0	5.7	1.2	4.7

relative performance levels out (as a function of ρ). Under the static greedy algorithm, sites that have much overlap among their hot-sets cause N-degree replication for valuable objects and no caching for less important objects. As correlation decreases, the degree of (wasteful) replication is automatically reduced—even under the greedy policy—because sites seek to cache *different* objects. When hot-sets are small, the static greedy policy at least succeeds in achieving high *local* cache hit-ratios. Relative performance (compared to the optimal) suffers because the tradeoffs between local and remote cache are not evaluated. This leads to low remote cache hit-ratios. This performance aspect is accentuated for small ρ (at the limit, remote cache hit-ratios are 0). Note that this "correlation effect" is only observed for large cache sizes in Fig. 11. When hot-sets are large, the static greedy policy results in small *local* hit-ratios: The key performance issue is the effective use of remote cache even when sites have little hot-set overlap. As a result, relative performance does not improve much with decreasing correlation. Only when sites have large amounts of cache does decreasing correlation lead to relative performance improvement.

Note that when sites have small amounts of cache storage, relative performance is fine; relative performance is worse for a sharper hot-set than for a flat hot-set. However, when sites have large amounts of cache storage, then relative performance *degrades* as hot-sets become flatter. The key characteristic of the greedy algorithm is that sites do local optimization without knowledge of conditions at other sites. The local benefit of caching object i at site j when *no other* site has cached that object is $P_{ij}(t_d - t_l)$. If one replica is already cached at some other site, then the benefit is reduced to $P_{ij}(t_r - t_l)$. Consider the task of selecting among a set of cache candidates. Under the optimal algorithm, sites often cache objects with smaller P_{ij} because they can already access the more valuable objects through remote memory. Cache benefit is increased by reducing the access time for less valuable objects—i.e., objects are moved up the memory hierarchy from disk access to main-memory access. Under the greedy policy, the "memory hierarchy" factor is ignored in favor of the "object importance" factor. With sharp hot-sets and small cache sizes, the P_{ij} factor is large, so the "penalty" for local caching of an already replicated object is high. In other words, the alternative candidates for caching are sufficiently important that ignoring them greatly degrades relative performance. When sites have sharp hot-sets *and* lots

TABLE VI
ILLUSTRATION OF THE BENEFITS OF REMOTE CACHING (PER-SITE CACHE STORAGE HOLDS 5% OF OBJECTS)

		Mean Object Access Time		
$STORE_{MAX}$	ρ	Optimal	LRU, Remote Cache	LRU, Without Remote Cache
30%	1	7.1	19.5	44.4
30%	10	7.1	20.5	44.4
30%	50	14.2	30.8	44.4
30%	100	20.8	34.6	44.4
20%	1	11.2	27.1	47.8
20%	10	11.5	27.8	47.8
20%	50	19.6	35.0	47.8
20%	100	24.9	37.3	47.8
10%	1	22.5	37.3	50.4
10%	10	22.9	37.4	50.4
10%	50	28.0	39.1	50.4
10%	100	30.3	39.6	50.4
5%	1	34.0	40.5	51.0
5%	10	34.0	40.5	51.0
5%	50	34.0	40.5	51.0
5%	100	34.0	40.5	51.0

of cache, the P_{ij} term is *very* small (sites operate at the tail end of the access distribution), so that the overall penalty is relatively small. Conversely, when sites have flat hot-sets, the P_{ij} factor is small, so that the penalty is relatively small. Consequently, if sites have small amounts of cache, because the P_{ij} term is small and the penalty is taken over relatively few decisions, relative performance is not too bad. However, when sites have large amounts of cache, the penalty is taken for many cache decisions, so that relative performance is bad.

The dynamic greedy algorithm differs from the static greedy algorithm in the way that its performance is affected by the inter-site correlation parameter ρ. The static greedy algorithm tends to do better with decreasing correlation—dramatically better, in fact, except in the case of a flat hot-set. The effect of ρ on the dynamic greedy algorithm is not as clear-cut, and is related to hot-set shape and cache size. When hot-sets are sharp or moderate (Figs. 9 and 10), the performance of the dynamic greedy algorithm is less than three times worse than optimal for $\rho < 50$. As sites have less correlation, the relative performance degrades greatly (for larger cache sizes). Recall that the dynamic greedy algorithm manages local cache with an LRU policy. When hot-sets are relatively small sites can cache much of their hot-sets locally. The key performance issue is whether remote cache can be used given a local cache-miss. Even though sites do not know which objects are "statically" important, the small hot-set size means that "hot" objects will tend to be valuable objects. Because of the randomness introduced by the LRU policy, one site's cache contents does not duplicate another site's cache. These two factors result in relatively better performance than static greedy. However, as inter-site correlation decreases, the dynamic greedy (i.e., local cache management) policy degrades because the percent of remote cache-hits decreases especially for large cache sizes.

When hot-sets are flat (Fig. 11) the performance of the dynamic greedy algorithm (for a given amount of cache) is almost completely unaffected by ρ. Instead, the key determinant of performance is cache size. When per-site cache size is less than 8% of the total number of objects, then relative performance is less than two times worse than the optimal. As cache size increases, relative performance degrades to more than 4.6 times worse than the optimal. Because hot-sets are so large, one object is about as valuable to a given site as any other object. The LRU policy leads to an almost random relationship between sites' cache contents. As a result, for large hot-set the percent of remote cache hits increases despite the fact that local cache hits decrease. The large performance gap between the optimal and dynamic greedy algorithms occurs when there is enough cache storage in the system that the penalty for suboptimal decisions begins to really add up.

The above analysis also explains the performance differences *between* the greedy algorithms. For sharp or moderate hot-sets, the static greedy algorithm does better than the dynamic except in the case of strong inter-site correlation. Conversely, when hot-sets are flat, the dynamic greedy algorithm does better than the static except in the case of very weak correlation. Both algorithms do too much replication (see Tables V and VIII) because they are not aware of the cache decisions at other sites. The dynamic algorithm avoids complete duplication of cache contents because cache contents are dynamically determined by the actual (i.e., dynamic) pattern of object accesses. The advantage of the static algorithm is that it has complete knowledge about object value. Sharp hot-sets imply that local cache-hit ratios are quite high, regardless of remote cache. Except in the case of strong site correlation (where the static greedy algorithm does, in the limit, N-site replication), higher local cache hits are more important than higher remote hits. Flat hot-sets imply that local cache is ineffective, and that remote cache must be used efficiently for good performance. The dynamic greedy therefore does better than the static (except when the weak correlation automatically leads to LRU type of randomness). Tables V and VIII confirm this analysis. Note how the mean number of replicas declines "automatically" under the static greedy algorithm as inter-site correlation declines. Although the number of replicas also declines for the dynamic algorithm, the range is much smaller. For low values of ρ, the mean benefit per replica is much higher under the dynamic algorithm. As ρ increases, the situation reverses, and the static algorithm has higher $BNFT$ values. When sites have flat hot-sets ($STORE_{MAX}$ = 5%), the dynamic algorithm has $BNFT$ = 4.6 for all values of ρ. The static algorithm has lower values—except at the extreme of random correlation, where $BNFT$ = 4.7.

1) Benefits of Remote Caching: Although the focus of this subsection has been on the weaknesses of the isolationist algorithms as compared to the optimal, it is also important to note that the benefits of remote caching apply for all algorithms. In Table VI we compare the performance of two systems that use the LRU policy to manage local cache. The systems differ in that in one (the second column) sites are able to access remote cache; in the other system (the third column) remote caching is not supported. We see that with only ten sites

in the system, remote caching offers a very large performance improvement. Performance in the non-RCA system is not affected by the degree of inter-site correlation because sites cannot access remote cache in any case. It is also interesting to note that even with random correlation among the sites' data access, the RCA system using the LRU algorithm still has much superior performance compared to the non-RCA system. The benefits of remote memory architecture compared to more traditional client/server architectures are described more fully in [17].

F. Performance of the Distributed Strategies

The distributed algorithms fill the performance gap between the optimal and greedy algorithms. Instead of order of magnitude performance differences, the distributed local (henceforth "local") algorithm never does worse than 1.4 times optimal. The distributed global (henceforth "global") does even better. When per-site cache sizes are 5% or less of the total number of objects, then performance is less than 1.03 worse than optimal. Even with larger cache sizes, performance is never worse than 1.15 of optimal.

The isolationist algorithms do relatively worse with increasing cache size, and (for small cache sizes) do better with sharp hot-sets. This behavior is due to the ineffective use of remote memory. In contrast, the relative performance of distributed algorithms is much less affected by cache size, and gets better with flat hot-sets. (At the extreme of $\theta = 0.0$, performance is indistinguishable from the optimal). Tables V and VIII show that the greedy algorithms maintain a high number of replicas per cached object. These replicas crowd out other objects which must then be accessed on disk. In contrast, the distributed algorithms have $REPL$ values which are very close to optimal.[8] This indicates that sites are aware of the cached objects at other sites and take advantage of these replicas to bring other objects into main-memory. Because these algorithms factor remote site decisions into local cache decisions, the performance gap with respect to the optimal has to do with the issue of coordinated decision making (and the cost function in the case of the local algorithm).

The relative performance of both distributed algorithms is clearly dependent on the shape of the hot-set: the sharper the hot-set, the worse the relative performance. Compare, for example, Fig. 12 to Fig. 13 and Fig. 14 to Fig. 15. At the limit of completely flat hot-sets, the performance of the distributed algorithms is indistinguishable from the optimal. The reason for this behavior is that, to the extent that these algorithms make suboptimal decisions, a greater performance penalty is incurred when objects are individually more valuable.

Cache size also has an important effect on relative performance. In Fig. 14, if per-site cache size is less than 6% of the total number of objects then the performance gap is less than 1.05. The performance gap only exceeds 1.10 when per-site cache size is 10%. Although the performance gap of the local algorithm is larger than the global, the gap in Fig. 12 becomes large only when per-site cache size exceeds 4%. Similar behavior is seen in Figs. 13 and 15. This behavior resembles the effect of hot-set shape: the penalty for suboptimal decisions increases as the ratio of per-site cache storage to the total number of objects increases.

Inter-site correlation plays a role in system behavior. In examining Fig. 14 we see that (when per site cache size is larger than 6%) relative performance degrades as correlation is reduced beyond the point of $\rho = 35$. In Fig. 15, when per site cache size is larger than 7%, performance degrades as correlation decreases for ρ greater than 50.

The consequences of decreasing inter-site correlation on the effectiveness of distributed processing impact both the global and local algorithms. The local algorithm, however, also degrades with *increasing* correlation. As a result, relative local performance degrades at both extremes of the ρ parameter, and does best for intermediate values. In the case of sharp hot-sets (Fig. 12), local performance tends to degrade with increasing correlation. Although local algorithm *does* use information about remote memory, this information *per se* is not enough (for optimal average performance) when sites have strong correlation among their hot-sets. Sites replicate the same set of valuable objects—even though the objects are already cached at other sites. The "object importance" factor is sufficiently large that, coupled with the difference between local and remote access time, caching valuable objects benefits sites more than bringing less valuable objects off disk. Observe in Table VIII how the local algorithm has the largest $REPL$ values for sharp hot-sets and strong correlation.

The global algorithm therefore has the greatest performance improvement, compared to to the local, when sites have large amounts of cache and their hot-sets are closely correlated. Under the global strategy, certain sites are constrained to cache less important objects so that *no* site has to access the disk for these objects. Note that these constraints causes *all* sites to improve their performance. The standard deviation of site performance (Table VII), when hot-sets are either moderate-flat *or* when correlation is strong, is about the same for both the local and global algorithms. This shows that some sites do not do appreciably better than other sites—despite the fact that they are doing local optimization. Instead, *all* sites suffer about the same magnitude of performance loss. Under sharp hot-sets and little correlation, the standard deviation is much greater under the local than under the global algorithms. Local optimization causes sites that happen to benefit from remote cache (due to the randomness caused by ρ) to do much better than other sites. The cooperation caused by the global optimization goal "smoothes out" the intrinsic randomness of the system.

When sites have moderate hot-sets (Fig. 13), the local algorithm (relative to the optimal) exhibits different behaviors depending on the amount of cache storage. When sites have little cache, then performance can actually *degrade* with decreasing correlation. Only for large amounts of storage does performance improve with decreasing correlation. The hot-sets of Fig. 13 are larger than in Fig. 12. Consequently, when

[8] In a symmetric topology caching a *second* replica cannot improve the performance of remote sites because all sites are equally distant from one another. The LAN environment investigated here has a symmetric topology, which is why the optimal $REPL$ values are about 1 in all cases. In contrast, the greedy algorithms have much higher $REPL$ values than the optimal, while the distributed algorithms have small $REPL$ values.

TABLE VII
PERFORMANCE OF THE DISTRIBUTED AND DYNAMIC GREEDY ALGORITHMS (PER-SITE CACHE STORAGE HOLDS 5% OF OBJECTS)

		Distributed Local			Distributed Global			Dynamic Greedy		
$STORE_{MAX}$	ρ	RTime	SDev	C_{HIT}	RTime	SDev	C_{HIT}	RTime	SDev	C_{HIT}
30%	1	8.4	0.5	0.942	7.1	0.5	0.972	19.5	0.3	0.747
30%	10	8.0	0.6	0.949	7.3	0.4	0.966	20.5	0.6	0.729
30%	50	14.7	2.1	0.823	14.4	2.2	0.716	30.8	1.0	0.549
30%	100	23.3	4.9	0.671	21.0	2.5	0.716	34.6	0.9	0.804
20%	1	11.2	0.3	0.900	11.2	0.3	0.900	27.1	0.3	0.619
20%	10	11.9	0.6	0.886	11.6	0.4	0.891	27.8	0.3	0.608
20%	50	20.2	2.3	0.735	19.8	2.2	0.745	35.0	1.0	0.482
20%	100	27.3	3.7	0.605	25.0	2.1	0.652	37.3	0.5	0.441
10%	1	22.5	0.3	0.702	22.5	0.2	0.702	37.3	0.3	0.445
10%	10	23.1	0.4	0.692	22.9	0.4	0.695	37.4	0.3	0.445
10%	50	28.4	1.5	0.599	28.1	1.7	0.605	39.1	0.5	0.415
10%	100	31.7	1.7	0.534	30.4	1.5	0.561	39.6	0.3	0.407
5%	1	34.0	0.3	0.499	34.0	0.3	0.499	40.5	0.3	0.391
5%	10	34.0	0.3	0.499	34.0	0.3	0.499	40.5	0.2	0.392
5%	50	34.0	0.3	0.498	34.0	0.3	0.498	40.5	0.3	0.391
5%	100	34.0	0.4	0.501	34.0	0.4	0.501	40.3	0.2	0.395

sites are evaluating the cache candidate set, the "object importance" factor is smaller and the "memory hierarchy" factor implies that sites should concentrate on bringing nonreplicated objects from disk into main-memory. Closer overlap among sites then leads to larger remote cache-hit ratios. As sites have more available cache storage decisions are made about objects at the tail end of the access distribution. Local optimization then encourages replicating even marginally (locally) important objects as opposed to caching even *less* important objects that would tend to benefit the overall system. In this situation, the local strategy does relatively better when low inter-site correlation implies that sites can benefit from the differing cache contents of other sites.

1) Convergence Properties of the Distributed Algorithms: By its nature, the solution achieved by either of the distributed algorithms improves monotonically with each new iteration of the algorithm. Two questions therefore arise. First, how quickly do the distributed algorithms converge to a stable solution? (A *stable* solution can be defined as one which does not change through any further iterations of the algorithm. Because of the monotonicity property and the finite cardinality of the state space, a stable solution *must* be reached eventually.) Second, how close is this stable solution to the global optimal solution? (Since each iteration of the algorithm can change only the eligibility decisions at one site, global optimality may not be achieved.) In this subsection we answer these questions for the distributed global algorithm.

In Table IX statistics for the distributed global algorithm are presented. (The cases examined are the same as in Table VII.) Entries in the third and fourth columns indicate the number of algorithm iterations needed to reach a given percent of the optimal algorithm's performance. For example, when $STORE_{MAX}$ = 30% and ρ = 1, then a solution which is at least 95% as good as the optimal is reached by the ninth site on the first round of cache decisions (denoted by "1,9"). Because individual cache decisions at the first nine sites can yield so much benefit, good overall performance is achieved—even though the tenth site has not yet made its first set of decisions. An additional site (the tenth) must make its decisions in order to reach a solution which is at least 98% as good as optimal ("2,0"). Clearly, very good solutions are reached quite rapidly. In fact, though not indicated in the table, all examples achieved stability by the third round of iterations. The *EQUIV* column indicates whether or not

TABLE VIII
REPLICATION DONE BY DISTRIBUTED AND DYNAMIC GREEDY ALGORITHMS (PER-SITE CACHE STORAGE HOLDS 5% OF OBJECTS)

		Distributed Local		Distributed Global		Dynamic Greedy	
$STORE_{MAX}$	ρ	*REPL*	*BNFT*	*REPL*	*BNFT*	*REPL*	*BNFT*
30%	1	1.4	10.7	1.1	11.0	1.8	8.7
30%	10	1.2	10.8	1.0	11.0	1.8	8.4
30%	50	1.0	9.5	1.0	9.6	1.4	6.4
30%	100	1.0	7.8	1.0	8.3	1.3	5.6
20%	1	1.1	10.1	1.1	10.2	1.5	7.2
20%	10	1.1	10.0	1.0	10.1	1.5	7.0
20%	50	1.0	8.4	1.0	8.5	1.3	5.6
20%	100	1.0	7.0	1.0	7.5	1.3	5.1
10%	1	1.1	7.8	1.1	7.9	1.3	5.2
10%	10	1.1	7.7	1.0	7.8	1.3	5.2
10%	50	1.0	6.8	1.0	6.9	1.3	4.7
10%	100	1.0	6.1	1.0	6.4	1.2	4.7
5%	1	1.1	5.6	1.1	5.6	1.3	4.6
5%	10	1.0	5.6	1.0	5.6	1.2	4.6
5%	50	1.0	5.7	1.0	5.7	1.2	4.6
5%	100	1.0	5.7	1.0	5.7	1.2	4.6

TABLE IX
NUMBER OF ITERATIONS REQUIRED BY DISTRIBUTED GLOBAL ALGORITHM TO ACHIEVE GIVEN PERCENT OF OPTIMAL PERFORMANCE

$STORE_{MAX}$	ρ	95% of Optimal	98% of Optimal	*EQUIV*
30%	1	(1,9)	(2,0)	YES
30%	10	(2,0)	(2,0)	NO
30%	50	(1,9)	(1,9)	NO
30%	100	(1,7)	(1,7)	NO
20%	1	(1,9)	(1,9)	YES
20%	10	(1,9)	(1,9)	NO
20%	50	(1,9)	(1,9)	NO
20%	100	(1,7)	(1,7)	NO
10%	1	(1,9)	(1,9)	YES
10%	10	(1,9)	(1,9)	YES
10%	50	(1,4)	(1,9)	NO
10%	100	(1,4)	(1,7)	NO
5%	1	(1,9)	(1,9)	YES
5%	10	(1,9)	(1,9)	YES
5%	50	(1,4)	(1,9)	YES
5%	100	(1,3)	(1,7)	YES

the stable solution reached by the distributed global optimal algorithm is within 0.1% of the global optimal solution. Obviously, not all examples achieve this.

Table IX shows that the rate of improvement is not related to whether this equivalence of solutions actually occurs. For example, when ρ = 100 the rate of improvement is the most rapid of all the $STORE_{MAX}$ = 30% cases. However, equivalence does not ultimately occur—in contrast to the case of ρ = 1, which has a slower improvement rate.

Recall that the optimal algorithm determines the optimal configuration X in "one" step. The distributed algorithms are heuristics in which each site makes B eligibility decisions (where B is the cache size) before the next site makes *its* decisions. The set of these decisions is supposed to transform the site's cache contents into the state it has in the optimal configuration. In practice, one site's decisions constrain the decisions of all other sites. As we have seen, because the sites under the distributed algorithm do not make coordinated decisions, the union of locally optimal decisions do not always equal the globally optimal configuration. Globally suboptimal decisions have, of course, a greater effect when individual cache slots are more valuable (large $STORE_{MAX}$). Equally importantly, when sites have less correlation among their hot-sets, the implications of one site's decisions on other sites are more subtle than when they are closely correlated. As a result, the issue of coordinated decisions (i.e., the optimal algorithm) plays a larger role.

The rapid improvement of the distributed algorithms points to an important advantage over the optimal algorithm. If the system need only determine a configuration once, then it would make sense to use the optimal rather than the distributed algorithm. In practice, of course, sites do not have access to exact, *a priori* knowledge of access frequencies. A dynamic estimation, based on previous system history, would then be done periodically; the new estimate of the P_{ij} would then be input to a new invocation of the optimization algorithm. Because the optimal algorithm requires a set of coordinated decisions, changes in access frequencies can potentially require many changes in sites' caches. In contrast, the localized decisions made by under the distributed algorithms means that fewer cache changes need be made if only some sites have different access frequencies. Since the distributed algorithms improve rapidly, an RCA implementation would prefer them over the optimal because they can afford to do the optimization more frequently than the optimal algorithm.

2) Scalability of the Distributed Algorithms: The number of objects, M, in the experiments described in the tables and figures is set to 100 in order to facilitate generation and analysis of the results. By varying the ratios of per-site cache size to M we can predict the behavior of a system with much larger M and proportionally more cache storage. An obvious issue is the *scalability* of the distributed algorithms. The complexity of the optimal algorithm is a linear function of the number of sites and the number of objects (see Section IV-A). As explained in Section IV-B, the distributed algorithms are more efficient than the optimal algorithms. However, if the *performance* of the distributed algorithm is also a function of the amount of per-site cache storage B, then the algorithm will not scale well in a realistic system.

To analyze the scalability of the distributed global algorithm we perform the following experiment (see Table X). The number of objects in the system is increased by two orders of magnitude (from 100 to 10 000). The θ values are adjusted to maintain the same $STORE_{MAX}$ values of Table VII and Table VIII. Four "correlation" values are listed per case: "total" corresponds to ρ = 1; "random" corresponds to ρ = M. The values of 0.9 and 0.5 correspond, respectively, to setting ρ to be one tenth and one half of M. Per-site cache storage is maintained, in both cases, at 5% of M. We find that the algorithm scales very well. Performance for most cases is identical. Differences between the performance are due to the effect of ρ being an input to the process of a random generation of site hot-sets. In some cases (the second case of $STORE_{MAX}$ = 30%), performance is better when M is 10 000. In other cases (the fourth case of $STORE_{MAX}$ = 30%), performance is better when M is 100.

G. The Effect of η on Performance

The previous analysis of performance involves the situation where all sites have uniform *relative* activity (i.e., η = 1.0). Each site's contribution to average performance is thus the same as any other site. We now examine situations in which some sites have greater activity than others. Table XI shows how the optimal average algorithm performs for three values of η. Recall that smaller η implies a few, very active, sites; other sites in the system are much less active.

One clear trend is that mean access time decreases as relative site activity is more skewed (i.e., as η decreases). Also, the improvement is more marked when sites have sharp hot-sets (large $STORE_{MAX}$ values). Finally, the relative improvement (over η values) is more pronounced when sites have less hot-set correlation (large ρ).

TABLE X
SCALABILITY PERFORMANCE OF THE DISTRIBUTED GLOBAL ALGORITHM FOR TWO VALUES OF M (PER-SITE CACHE STORAGE HOLDS 5% OF OBJECTS)

		Distributed Global	
$STORE_{MAX}$	*Correlation*	M = 100	M = 10 000
30%	TOTAL	7.1	7.1
30%	0.9	7.3	7.2
30%	0.5	14.4	13.9
30%	RANDOM	21.0	22.0
20%	TOTAL	11.2	11.2
20%	0.9	11.6	11.5
20%	0.5	19.8	19.2
20%	RANDOM	25.0	25.7
10%	TOTAL	22.5	22.5
10%	0.9	22.9	22.9
10%	0.5	28.1	27.6
10%	RANDOM	30.4	30.5
5%	TOTAL	34.0	34.0
5%	0.9	34.0	34.0
5%	0.5	34.0	34.0
5%	RANDOM	34.0	34.0

TABLE XI
PERFORMANCE OF THE OPTIMAL AVERAGE ALGORITHM FOR THREE VALUES OF η (PER-SITE CACHE STORAGE HOLDS 5% OF OBJECTS)

		Optimal Average		
$STORE_{MAX}$	ρ	η = 1.0	η = 0.5	η = 0.0
30%	1	7.1	7.0	6.9
30%	10	7.1	7.0	6.9
30%	50	14.2	13.6	12.3
30%	100	20.8	20.5	19.2
20%	1	11.2	11.2	11.1
20%	10	11.5	11.5	11.3
20%	50	19.6	19.0	17.7
20%	100	24.9	24.9	23.6
10%	1	22.5	22.5	22.5
10%	10	22.9	22.8	22.8
10%	50	28.0	27.6	26.8
10%	100	30.3	30.3	29.6
5%	1	34.0	34.0	34.0
5%	10	34.0	34.0	34.0
5%	50	34.0	34.0	34.0
5%	100	34.0	34.0	34.0

As η decreases, a few sites become increasingly more important relative to the other sites in the system. When other system characteristics are held constant (e.g., per-site cache size, hot-set distribution, and inter-site hot-set correlation), then the optimal decision is for unimportant sites to cache objects for the important sites. These sites behave, in a sense, as "object servers" rather than as independent workstations. Important sites therefore have higher remote cache-hit ratios. Although less active sites may suffer a performance loss (because locally important objects are not stored on-site), overall system performance improves because the active sites serve a greater number of object requests. Caching a hot object yields higher cache-hit ratios when hot-sets are sharper. As a result, benefits from the "object-serving" of less active sites increase for sharper hot-sets.

Changes in overall system performance (due to η) are relatively small when sites exhibit much correlation between their hot-sets. In such a case, sites *already* cache much of one another's objects: emphasizing the importance of certain sites has little effect on replication decisions. In other words, when sites already get maximum benefit from the RCA system (due to small ρ), changes in η do not change the contents of the system hot-set much. As sites have less correlation, however, the system's hot-set is much larger. Decreasing η implies that the system hot-set must be weighted by an object's importance (i.e., the site-weighted frequency of access). The large hot-set that exists for $\eta = 1.0$ becomes much smaller when $\eta = 0.0$. Under the optimal algorithm, sites cooperate to cache globally important objects, and performance improves considerably.

Note that this analysis of the behavior of the optimal algorithm's performance under different values of η does not involve the issue of coordinated *versus* distributed decision making. The factor of relative site activity only affects the value that a site assigns to the presence of an object from cache. This only involves the issue of the optimization goal. As one would therefore expect, we found that the relative performance of the distributed global algorithm (compared to the optimal) is unaffected by changes in η. In almost all cases, the distributed global algorithm does not suffer noticeable degradation of its relative performance as site activity becomes more skewed.

Under the static greedy and the distributed local algorithms, individual sites do local optimization. Table XII shows the relative performance of these algorithms (compared to the optimal) for three different instances of relative site activity. The factors that determine relative performance when sites have the same amount of activity were previously discussed. The local optimization goal means that individual sites maintain the same behavior for all values of η. As relative activity diverges, overall performance depends on *which* site has *what* degree of importance. For example, if site 1 had the best performance of all sites—a purely random occurrence with no bearing on system modeling—then mean performance will improve as η decreases. Analysis of these algorithms (i.e., holding other variables constant, and varying η) is therefore complicated because of the effect of these "random" inputs. Nevertheless, a few trends are clear.

When individual cache slots are not valuable (because of flat hot-sets), then relative performance is almost completely unaffected by decreasing η (e.g., $STORE_{MAX}$ = 10% and 5%). As shown in Table XI, in such cases the optimal algorithm can hardly exploit the varying site activity because the presence of any given object has little effect on performance. The actual performance of the local optimization algorithms is unaffected by η for similar reasons.

When sites have sharp hot-sets, then relative performance is mainly affected by the degree of inter-site correlation. Table XII shows that the distributed local algorithm suffers

TABLE XII
RELATIVE PERFORMANCE OF THE LOCAL OPTIMIZATION ALGORITHMS FOR THREE VALUES OF η (PER-SITE CACHE STORAGE HOLDS 5% OF OBJECTS)

		Static Greedy			Distributed Local		
$STORE_{MAX}$	ρ	η = 1.0	η = 0.5	η =0.0	η = 1.0	η = 0.5	η = 0.0
30%	1	6.24	6.30	6.41	1.19	1.19	1.20
30%	10	4.14	4.09	4.08	1.13	1.12	1.13
30%	50	1.76	1.81	1.92	1.03	1.06	1.13
30%	100	1.34	1.38	1.50	1.12	1.17	1.30
20%	1	4.49	4.50	4.53	1.00	1.00	1.33
20%	10	3.27	3.26	3.25	1.03	1.02	1.02
20%	50	1.56	1.59	1.65	1.03	1.04	1.08
20%	100	1.29	1.31	1.40	1.10	1.13	1.22
10%	1	2.50	2.50	2.51	1.00	1.00	1.00
10%	10	2.10	2.10	2.10	1.01	1.01	1.01
10%	50	1.35	1.35	1.37	1.01	1.02	1.02
10%	100	1.21	1.22	1.26	1.05	1.06	1.10
5%	1	1.75	1.75	1.75	1.00	1.00	1.00
5%	10	1.60	1.60	1.60	1.00	1.00	1.00
5%	50	1.23	1.23	1.23	1.00	1.00	1.00
5%	100	1.15	1.15	1.15	1.00	1.00	1.00

performance degradations (as a function of decreasing η) when ρ = 100. The relative performance of the static greedy algorithm also degrades for smaller values of ρ. When sites can benefit from each other's cache, then the penalty for local optimization is not too high—active sites can still utilize the contents of less active sites' cache. As correlation among sites' hot-sets decreases, then local optimization does increasingly (relatively) worse for the case with smaller hot set, because less active sites cannot benefit the other sites in the system. As before, the distributed local algorithm does better than static greedy.

VI. CONCLUSION AND FUTURE WORK

The high performance networks in many large distributed systems enable a site to reach the main memory of other sites more quickly than the time to access local disks. *Remote memory* can serve as an additional layer in the memory hierarchy between local memory and disks, but optimizing performance in the remote cache architecture is complicated by the fact that local sites may make replication decisions independently of other sites.

Remote caching architectures offer immediate benefit because of the opportunity to take advantage of objects that are cached at remote sites. *Efficient* use of the memory resources in such a system depends critically on replica management. A tradeoff exists between simplistic replication of valuable objects (eliminating the need to pay the extra cost of remote access), and using local cache storage to cache unreplicated objects. This paper shows that the optimal selection of objects for caching is a function of the hot-set curve, available cache storage, differences between the access patterns of the sites, and the criterion for optimal performance.

In this paper we have:

1) Introduced the idea of a remote caching architecture.
2) Analyzed the issues affecting its performance.
3) Developed optimal replica management algorithms.
4) Examined the issues of *cost function* and *remote caching information* as they effect algorithm performance.
5) Analyzed the interaction of "object importance" factor with "memory hierarchy" factor.
6) Developed a distributed global optimization algorithm with performance very close to optimal.
7) Developed a distributed local optimization algorithm (that maintains site autonomy) with mean access times that are generally close to optimal.
8) Devised greedy algorithms for replica management.

We identified and analyzed the factors that are critical to system performance. The performance of two optimal algorithms was used as an upper bound on remote caching architecture performance: Optimality results from the fact that sites make coordinated decisions. Two greedy algorithms are used as a lower bound on system performance. These algorithms do not factor information about the state of other sites into local site decisions. We showed that while locally "greedy" decisions can lead to far worse performance than "optimal" decisions, the degree of performance degradation depends on the amount of cache storage available, the kind of access pattern, and the variation among the sites' access patterns. Two distributed algorithms are then developed which provide performance that is close to the optimal—even though decisions are made in distributed fashion. The algorithms work by exchanging information between sites. This information is used as input for local cache decisions. One algorithm does local optimization, and the other does global optimization. The performance differences between the two point to the autonomy tradeoffs in a remote caching architecture.

This paper demonstrates the potential of remote memory to reduce the number of disk acesses, and thus to improve performance. It also discusses distributed algorithms that, given knowledge of object access rates, enable sites to achieve close to optimal performance. Optimality refers here to *average* object access time, and assumes that object accesses are "static" for significant periods of time. A major direction of future research in this area is *dynamic* replica management. A key issue for remote caching is the development of an LRU analog that captures global—in addition to local—object access patterns [18]. As this paper shows, algorithms which only capture local object value have order-of-magnitude performance gaps compared to the optimal. Another area of research is applying the distributed algorithms to a nonsymmetric network topology. In a symmetric topology, the "arithmetic" of these algorithms is greatly simplified because, from the perspective of any given site, all sites can be characterized as either "local" or "remote'.' This categorization is, of course, valid for LAN-like topologies. In more complicated topologies, it might be necessary to modify the algorithms to reduce the amount of information that must be stored to keep track of n sites' constraints. In addition, synchronous site decisions would be less reasonable in such topologies, and algorithms

that work despite asynchronous site decisions would need to be devised. Implementation of remote memory requires that these issues be addressed; this paper shows that RCA potential is sufficiently great to make implementation worthwhile.

ACKNOWLEDGMENT

We thank the anonymous reviewers for their suggestions, which have greatly improved the quality of the paper.

REFERENCES

[1] E. A. Arnould, F. J. Bitz, E. C. Cooper, H. T. Kung, R. D. Sansom, and P. A. Steenkiste, "The design of Nectar: A network backplane for heterogeneous multicomputers," in *Proc. ASPLOS III*, Apr. 1989, pp. 205–216.

[2] J. Bennett, J. Carter, and W. Zwaenepoel, "Munin: Distribiuted shared-memory based on type-specific memory coherence," in *Proc. 1990 Conf. Principles and Practice of Parallel Programming,* ACM Press, New York, NY, 1990, pp. 168–176.

[3] D. P. Bertsekas and D. A. Castinon, "The auction algorithm for the transportation problem," *Ann. Oper. Res.,* pp. 67–96, 1989.

[4] M. J. Carey, M. J. Franklin, M. Livny, and E. J. Shekita, "Data caching tradeoffs in client-server DBMS architectures," in *Proc. 1991 ACM SIGMOD Int. Conf. Management of Data,* May 1991, pp. 357–366.

[5] R. G. Casey, "Allocation of copies of a file in an information network," in *Proc . AFIPS 1972 Spring Joint Comput. Conf.,* AFIPS Press, 1972, pp. 617–625.

[6] D. Comer and J. Griffoen, "A new design for distributed systems: The remote memory model," in *Proc. Summer USENIX,* 1990.

[7] A. Dan and D. Towsley, "An approximate analysis of the LRU and FIFO buffer schemes," in *Proc. ACM Sigmetrics,* 1990, pp. 143–152.

[8] A. Dan and P. S. Yu, "Performance analysis of buffer coherency policies in a multisystem data sharing environment," *IEEE Trans. Parallel Distributed Syst.*, vol. 4, pp. 289–305, Mar. 1993.

[9] ——, "Performance analysis of coherency control policies through lock retention," in *Proc. ACM SIGMOD,* 1992, pp. 114–123.

[10] G. Delp, "The architecture and implementation of Memnet: A high-speed shared memory computer communication network," Doctoral dissertation, Univ. of Delaware, 1988.

[11] L. Dowdey and D. Foster, "Comparative models of the file assignment problem,"*ACM Comput. Surveys,* vol. 14, no. 2, pp. 287–313, June 1982.

[12] N. M. Downie and R. W. Heath, *Basic Statistical Methods.* New York: Harper and Row, 1965.

[13] E. W. Felten and Z. Zahorjan, "Issues in the implementation of a remote memory paging system," Tech. Rep. 91-03-09, Univ. of Washington, Mar. 1991.

[14] H. Garcia-Molina and B. Kogan, "Node autonomy in distributed systems," in *Proc. Int. Symp. Databases in Parallel and Distributed Systems,* Dec. 1988, pp. 158–166.

[15] R. S. Garfinkel and G. L. Nemhauser, *Integer Programming.* New York: Wiley, 1972.

[16] D. E. Knuth, *The Art of Computer Programming, Volume 3: Sorting and Searching.* Reading, MA: Addison-Weseley, 1973.

[17] A. Leff, C. Pu, and F. Korz, "Cache performance in server-based and symmetric database architectures," in *Proc. ISMM Int. Conf. Parallel and Distributed Comput. and Syst.,* Oct. 1990.

[18] A. Leff, J. L. Wolf, and P. S. Yu, "LRU-based replication strategies in a LAN remote caching architecture," in *Proc. 17th Annu. Conf. Local Comput. Networks,* Minneapolis, MN, Sept. 1992.

[19] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy, "The directory-based cache coherence protocol for the Dash multiprocessor," in *Proc. 17th Int. Symp. Comput. Architecture,* IEEE CS Press, Los Alamitos, CA, 1990, pp. 148–159.

[20] K. Li and P. Hudak, "Memory coherence in shared virtual memory systems," *ACM Trans. Comput. Syst.,* vol. 7, no. 4, pp. 321–359, Nov. 1989.

[21] K. R. Pattipati and J. L. Wolf, "A file assignment problem model for extended local area networks," in *Proc. Distributed Comput. Syst.,* 1990, pp. 554–561.

[22] C. Pu, J. D. Noe, and A. Proudfoot, "Regeneration of replicated objects: A technique and its Eden implementation,"*IEEE Trans. Software Eng.,* vol. SE-14, no. 7, pp. 936–945, July 1988.

[23] C. Pu, A. Leff, F. Korz, and S-W. Chen, "Redundancy management in a symmetric distributed main-memory database," Tech. Rep. CUCS-014-090, Columbia Univ., 1990.

[24] C. Pu, D. Florissi, P. Soares, K. L. Wu, and P. S. Yu, "Performance comparison of dynamic policies for remote caching," *Concurrency: Practice and Experience,* vol. 5, pp. 239–256, June 1993.

[25] C. V. Ramamoorthy and B. W. Wah, "The isomorphism of simple file allocation," *IEEE Trans. Comput.,* vol. C-32, no. 3, pp. 221–232, Mar. 1983.

[26] B. N. Schilit and D. Duchamp, "Adaptive remote paging for mobile computers," Tech. Rep. CUCS-004-91, Columbia Univ., 1991.

[27] M. Schroeder and M. Burrows, "Performance of Firefly RPC," in *Proc. 12th Symp. Oper. Syst. Principles,* Dec. 1989, pp. 83–90.

[28] D. F. Thiebaut, H. D. Stone, and J. L. Wolf, "Improving disk cache hit-ratios through cache partitioning," *IEEE Trans. Comput.,* June 1992, pp. 665–676.

[29] B. W. Wah, "File placement in distributed computer systems,"*IEEE Computer,* vol. 17, no. 1, pp. 23–32, Jan. 1984.

[30] Y. Wang and L. A. Rowe, "Cache consistency and concurrency control in a client/server DBMS architecture," in *Proc. 1991 ACM SIGMOD Int. Conf. Management of Data,* May 1991, pp. 367–376.

[31] J. L. Wolf, "The placement optimization program: A practical solution to the disk file assignment problem," in *Proc. ACM Sigmetrics,* 1989, pp. 1-10.

[32] J. L. Wolf, D. M. Dias, and P. S. Yu, "A parallel sort-merge join algorithm for managing data skew," *IEEE Trans. Parallel Distributed Syst.*, vol. 4, pp. 70–86, Jan. 1993.

Avraham Leff (S'90–M'91) received the B.A., M.S., and Ph.D. (in computer science, 1991) degrees from Columbia University, New York, NY.

He works in the high-speed interconnect group at IBM Research, Hawthorne, NY. His current research includes high-speed channels and distributed systems.

Joel L. Wolf (S'93), for a photograph and biography, see p. 86 of the January 1993 issue of this TRANSACTIONS.

Philip S. Yu (S'76–M'78–SM'87–F'93), for a photograph and biography, see p. 86 of the January 1993 issue of this TRANSACTIONS.

Prediction of Performance and Processor Requirements in Real-Time Data Flow Architectures

Sukhamoy Som, *Member, IEEE,* Roland R. Mielke, *Member, IEEE,* and John W. Stoughton, *Member, IEEE*

Abstract—**The purpose of this paper is to present a new data flow graph model for describing the real-time execution of iterative control and signal processing algorithms on multiprocessor data flow architectures. Identified by the acronym ATAMM, for Algorithm to Architecture Mapping Model, the model is important because it specifies criteria for a multiprocessor operating system to achieve predictable and reliable performance. Algorithm performance is characterized by execution time and iteration period. For a given data flow graph representation, the model facilitates calculation of greatest lower bounds for these performance measures. When sufficient processors are available, the system executes algorithms with minimum execution time and minimum iteration period, and the number of processors required is calculated. When only limited processors are available or when processors fail, performance is made to degrade gracefully and predictably. The user off-line is able to specify tradeoffs between increasing execution time or increasing iteration period. The approach to achieving predictable performance is to control the injection rate of input data and to modify the data flow graph precedence relations so that a processor is always available to execute an enabled graph node. An implementation of the ATAMM model in a four-processor architecture based on Westinghouse's VHSIC 1750A Instruction Set Processor is described, and the performance of a real-time space surveillance algorithm on this system is investigated.**

Index Terms— **Algorithm to Architecture Mapping Model (ATAMM); iterative control and signal processing algorithms; multiprocessor data flow architectures; periodic, nonpreemptive, dynamic multiprocessor scheduling; real-time systems; time performance and processor requirement prediction.**

I. Introduction

MULTIPROCESSOR computing systems are being used to obtain high-speed computing performance through concurrency, while at the same time achieving a high level of fault tolerance and reliability [1]. The data flow strategy is gaining wide acceptance as an excellent computational model for multiprocessor systems [2]. In the data flow paradigm, an algorithm is expressed as a collection of tasks which are to be executed according to a set of precedence constraints. The algorithm is represented by a data flow graph, a directed graph in which the nodes represent tasks and the arcs represent communication paths between nodes [3]. The presence of data on an arc is denoted by the placement of a token on that arc. A node is said to be enabled when all incoming arcs contain a token. An enabled node is executed (fired) by an available processor by encumbering one token from each incoming arc, delaying for a time equal to the execution of the node, and the depositing one token on each outgoing arc. A number of experimental data flow multiprocessor architectures have been developed and tested [4].

An emerging area of considerable interest is to use data flow computers for real-time computing applications such as the implementation of control and signal processing algorithms for aerospace, factory automation, and remote sensing [5]. Real-time control and signal processing algorithms possess unique features often not shared with general-purpose computing problems. First, these algorithms periodically process infinite sequences of input data and produce infinite sequences of output data. The process of consuming one input token, executing all algorithm tasks once, and producing one output token is called an iteration. Because control and signal processing algorithms repetitively perform algorithm iterations, computing concurrency is achieved in two ways. Different processors can be assigned to simultaneously perform different tasks for the same iteration. This intraiteration concurrency is referred to as parallel concurrency because it is the result of inherent parallelism in the algorithm. In addition, however, different processors can be assigned to simultaneously perform tasks for different iterations. This interiteration concurrency is referred to as pipeline concurrency because the algorithm is repeated periodically for successive iterations, like a pipeline. Thus, real-time algorithms have an additional degree of freedom for achieving concurrency. Second, real-time algorithms generally require consideration of at least two time performance measures. The time which elapses between the encumbering of an input token and the production of the corresponding output token for a single iteration is called the iteration execution time, or simply the execution time. Execution time is important in real-time control and signal processing algorithms because it corresponds to time delay or phase lag. The time which elapses between the production of output tokens for successive iterations is called the iteration period. The inverse of the iteration period is the iteration frequency or sample frequency, a measure of algorithm throughput. The sample frequency is important because it limits the bandwidth of input and output signals. When task execution schedules for successive algorithm iterations are allowed to overlap, the performance measures execution time and iteration period

Manuscript received January 29, 1991; revised June 1, 1992. This work was supported in part by the NASA Langley Research Center under Grants NAG1-683 and NCC1-136.

S. Som was with the Department of Electrical and Computer Engineering, Old Dominion University, Norfolk, VA 23529. He is now with NASA Langley Research Center, Mail Stop 473, Lockheed Engineering and Science Company, Hampton, VA 23681–0001.

R. R. Mielke and J. W. Stoughton are with the Department of Electrical and Computer Engineering, Old Dominion University, Norfolk, VA 23529.

IEEE Log Number 9213610.