Valued Redundancy

Calton Pu, Avraham Leff, Shu-Wie Chen, Frederick Korz, Jae Wha
Department of Computer Science
Columbia University
New York, NY 10027

Technical Report No. CUCS-453-89

Abstract

Replicated objects increase distributed system performance and availability. An object is more valuable to the system if it contributes more to system performance (e.g., it is frequently accessed) and availability. Similarly, an object is less valuable if it is expensive to maintain (e.g., it is a large object). By replicating only the most valuable objects we use redundancy to maximize system performance and availability at low cost. A simulation study of a distributed main-memory database shows substantial performance and availability gains with valued redundancy.

Contents

1 Introduction

Data redundancy in centralized systems shows up in two ways. Horizontally, copies of an object may be placed on storage media of comparable speed, say mirrored disks, for increased availability. Vertically, copies of an object may appear in storage media with order-of-magnitude differences in performance characteristics, for example, between main memory and disks. Vertical redundancy typically increases performance, as in caching or buffering.

Data redundancy in distributed systems naturally combines horizontal redundancy with vertical redundancy. Since copies are stored on different nodes, accessing a local copy increases performance while the presence of multiple copies increases availability. The communications network, however, complicates the analysis of performance and availability gains, since its latency and bandwidth add more levels of performance differences and its own failure modes. Already hard questions such as "*how many* copies should we keep" and "*where* do we put the copies" become even more difficult to answer.

Redundancy for all its benefits has its costs. Poorly managed redundancy can be worse for system performance than having no redundancy. It is also a common belief that redundant systems with higher availability will carry a higher price tag. However, we believe that an integrated redundancy management system can handle both horizontal and vertical redundancy to achieve improved performance *and* availability in a distributed system's rich environment. The redundancy management should minimize the storage cost and the system overhead, at the same time it maximizes the performance and availability.

Our idea is *valued redundancy*. An object's value increases with its usefulness in the system and decreases with its maintenance cost. By replicating only the most valuable objects (the cheapest objects that improve system performance and availability the best) we achieve the redundancy objective of maximizing the cost/performance ratio of redundancy.

The implementation of valued redundancy requires cooperation of the three existing components of a replication mechanism: the location algorithms that find objects and their copies, the replacement algorithms that manage local storage, and the update propagation algorithms that keep an object's copies mutually consistent. Most of the existing rich literature on data replication focuses on one of these three components. Valued redundancy provides the glue that implements a smart replication policy integrating these three components.

Because the weights in weighted voting [6] are analogous to the values in valued redundancy, we note here their differences. Weighted voting is an algorithm to maintain consistency among the copies of an object. Valued redundancy is an approach to manage the degree of redundancy for replicated data. More technically, weights are assigned to the copies in weighted voting to

represent the static differences relative to each other, e.g. a heavier weight for a copy on a faster machine. Values are assigned to the copies to represent the dynamic properties relative to other resources in the system, e.g. a higher value for a copy would keep it longer in the local cache.

2 The Idea And Its Implementation

2.1 Valued Redundancy

Replication in distributed databases has received considerable attention from two groups of researchers. For those interested in performance, replication decreases system response time since reading a local copy is faster than reading a remote copy. For those interested in reliability, replication increases system availability since reading from any of several copies is more likely to succeed. Frequently, a naive redundancy mechanism for performance reduces system availability (e.g., by requiring that all copies be accessible for consistent updates). Similarly, a simplistic redundancy method for availability reduces system performance (e.g., by requiring the access of more than one copy which worsens response time).

The idea of *valued redundancy* increases system performance and availability at low cost by replicating the most valuable objects of the system. Since the redundancy management system explicitly maintains a replicated object's cost/performance ratio as its value, it is clear that valued redundancy maximizes system performance and availability objectives while minimizing maintenance cost.

In the calculation of an object's value, we take into account the object's costs, performance contributions, and performance goals. Important cost parameters include the object creation costs (in terms of resources consumed) and maintenance costs, such as storage, consistent update across copies and garbage collection. The performance contributions include the access patterns such as read and write frequency. Finally, the main performance goals are object access time (averaged over the copies) and the apparent object availability (calculated over the attempted accesses).

These value calculation parameters are closely related. Between the performance goal of an object and its performance contributions, we have a positive correlation. For example, to maximize system throughput we would set high performance goals for frequently accessed objects. Between the object cost and performance goals, we also have a positive correlation. For instance, we would carry more copies of an object that is read by many nodes. Between the object cost and the performance contributions, we have a trade-off. For example, replica update cost will be high for objects that are modified often, while queries do not carry additional object maintenance cost.

With valued redundancy, in principle we can handle specific performance goals for each object. In this paper, we will focus on system-wide performance goals, set in terms of the average behavior. More concretely, we want a system with high throughput, fast response-time, and high availability. Valued redundancy will let us concentrate on the objects that have high performance contributions, replicating them to the extent we can afford. As we will see from our discussion in the next section on the implementation of valued redundancy, the interaction between the components will influence the value calculations strongly. Therefore, it is unlikely that one single value calculation formula will be the best for all the different combinations of algorithms. Part of our ongoing research is to determine how the value calculations are affected by the components of redundancy management.

2.2 Implementation Issues

Value calculations are the glue that holds together the three parts of a redundancy management system: object location algorithms, replacement algorithms, and consistent update algorithms. (For a particular redundancy management system, one concrete algorithm for each component will do.) Although these components interact strongly with the object value calculations, they are sufficiently independent to be described individually. In this informal discussion, we emphasize the functions of each component rather than rigorously analyze their interactions.

Object location algorithms find the objects and their copies in the system. Although we call them "algorithms" for consistency with the other components of a redundancy management system, they may use persistent data, such as directories, to store location information. An example of a pure location algorithm is broadcast, a frequently used mechanism for locating objects in a network. Since the location algorithms can improve their performance with caches, they may use the redundancy management recursively to store the object or copy location information.

Redundancy growing unboundedly will eventually exhaust the system resources, in particular the storage capacity. The decision criteria for choosing the copies to reside at a given level of memory hierarchy are embodied in the replacement algorithms, named for their similarity with the page replacement algorithms in virtual memory. For example, Least Recently Used (LRU) is a well-known replacement algorithm. The replacement algorithms are invoked when the storage capacity of a given memory hierarchy (e.g. main memory) is nearly full, to keep the most valuable objects and throw away the others.

Consistent update algorithms keep the copies of the same object mutually consistent. Most of the recent data replication work focuses on this component of redundancy management. For ex-

ample, Majority Voting (also called Quorum Consensus) is a well-known consistent read/update algorithm that keeps object consistency. Depending on the particular update algorithm, more or less work is required for each kind of object access. Also, algorithms may become more complex as more kinds of failures (e.g. network partitions) are considered.

For simplicity, we concentrate on data replication as the main form of redundancy in our database. Each object is represented by its copies. To read an object, the location algorithm will point to the (presumed) best copy. The replacement algorithm finds room in the local memory to store the copy. And the consistent update algorithm will make sure the copies remain mutually consistent when the object is modified.

The implementation of valued redundancy impacts primarily the replacement algorithm. Our "replacement algorithm" now has two parts, the object value calculations and the memory management. As we will see in the next paragraph, object value calculations involves the other two components as well. The memory management part consists of the resource manager that uses the object value to decide which objects to replicate and which ones to throw away.

An object's value calculations may depend on many factors in the system. Each redundancy management component contributes with the factors it affects. The replacement algorithm, for example, tabulates the object access patterns such as read frequency. The consistent update algorithm accumulates useful modification statistics; for instance, a frequently changed object carries a high maintenance cost and therefore should not be replicated too widely. The location algorithm keeps interesting object location information; for example, a frequently moved object increases its value and the probability of being replicated.

After this abstract discussion on the implementation issues of valued redundancy, we apply the idea of valued redundancy to a specific and concrete problem to test its validity. The demonstration environment we have chosen is a distributed main-memory database (DMMDB), which we describe now.

3 Distributed Main-Memory Database

3.1 Architectural Assumptions

We assume the architecture underlying the DMMDB to be a network of computer systems. Each node has one or more modern microprocessor CPUs (e.g. a 25 MHz MC68030), relatively large main memory (e.g. 16 MBytes per CPU), and sufficient secondary storage (e.g. magnetic or optical disks). We assume that the network latency and bandwidth are about an order of magnitude faster than the secondary storage (e.g. an Ethernet compared to a current technology Winchester drive). Many modern workstations meet these assumptions.

The key to our database study is the memory hierarchy in the DMMDB. For this study, we will ignore the CPU cache memory completely. We focus our attention on the memory hierarchy represented by the local main memory, remote main memory, and the disks. In terms of access time, there are single order of magnitude differences between the local memory (tenth of a millisecond), remote memory (millisecond), and disks (tens of milliseconds). If the network is a bottleneck, then there will be a larger difference between local and remote accesses to memory and disks. We will take this factor into account, but will not consider this as a significant source of performance gains. Archival storage is another level in the memory hierarchy postponed to future study.

In this architecture, we apply valued redundancy systematically along two dimensions. Vertically, each node will manage redundancy in the caching/buffering sense. In the DMMDB this is between the main memory and secondary storage (memory-disk). Horizontally, the network will manage redundancy in the data replication sense. In the DMMDB this is both at the main memory level (memory-memory) and at the secondary storage level (disk-disk). Much work has been done on using the memory-disk redundancy to increase performance (usually called a buffer, see Section 5.2). Also, many papers have been written on disk-disk redundancy to increase availability (see Section 5.3). Therefore, we will focus on the relatively unexplored area of memory-memory redundancy and use simplistic algorithms to manage the other two kinds.

3.2 Memory-Memory Redundancy

The memory hierarchy between the local main memory and remote main memory is peculiar to the DMMDB. Snooping caches in shared-memory multi-processors are very similar to our memory-memory redundancy, but the different performance characteristics between the local main memory and remote main memory have not been intensely explored.

To focus on the memory-memory redundancy, we simplify the DMMDB. Our model of DMMDB is a set of objects. We assume there is no disk-disk redundancy, i.e., the DMMDB is completely partitioned and each object resides on exactly one disk. Also, the memory-disk redundancy is managed locally, in conjunction with the memory-memory redundancy. A remote disk read request requires the object to be read into the remote node's memory before transmission. But the memory (buffer) management for both the memory-memory and memory-disk redundancies is the same, that it is based on assigned values.

We made a deliberate decision to define the object value as a function of exclusively local parameters. To maintain the accurate value of a global parameter is costly in a distributed system and even more so when the network scales up. To use an out-of-date value would seem

to contradict the basic idea of valued redundancy, which intends the object value to reflect the current situation adaptively. Therefore we will avoid global parameters in valued redundancy until we solve the maintenance cost problem.

Each object copy in the main memory has a Local Usage Value (LUV). The replacement algorithm will keep the objects with the highest LUV in the main memory and throw away those with low LUV. From the theoretical point of view, adding up all the copies' LUV gives us the object's value. It is easy to see that the objects of highest value have the highest degree of redundancy and are therefore also the most readily available.

To demonstrate the usefulness of valued redundancy, we need to specify the three components of our redundancy management system. For object location, we use broadcast and individual replies, which is one of the simplest location algorithms for a LAN-based system. For consistent update propagation, we use versions [1], which simplifies concurrency control. Finally, we assume a homogeneous system in which all the nodes have the same hardware and software.

4 The Simulation Study

4.1 The Simulation Model

We have created a simulation program to evaluate the performance and availability gains from valued redundancy in the DMMDB. Our simulation model abstracts away details from the DMMDB scenario that may not affect the subjects of our evaluation. Therefore, the simulation model has a fixed skeleton, which is the DMMDB, and three moveable parts, which are the components of the redundancy management system described in Section 2.2, namely, the location algorithm, the replacement algorithm, and the consistent update algorithm. The plan is to plug in different algorithms and see how they affect the system performance and availability.

The fixed skeleton in our simulation model reflects a network of computers. Each node in the network contains a CPU, some local memory that can cache a varied number of objects, and disks. The hardware resources in the system are the CPU, the disks, and the network. Requests may come in for each of them and are serviced on a FIFO basis. The software resources in the simulation model are the cached objects in the local memory, which the replacement algorithm and consistent update algorithms maintain and the location algorithm uses.

The simulation model has considerable detail and builds more sophisticated services out of more elementary ones. For example, a remote disk read is composed of a request to the remote node, a local disk read and a reply to the requester. Besides taking into account the queueing effects on the hardware resources, we also charge for the use of software resources, such as cache access and maintenance.

To make the results more predictable, we use the most common statistical distributions to generate the events. A central transaction server creates transactions with exponential inter-arrival times and distributes them uniformly over the nodes of the DMMDB. Read and write transactions are created under binomial distribution using the specified ratio, and they are always the same (a certain number of reads followed by writes). The hot-set curve represents the degree of access locality; currently we are using an exponentially decaying curve (a small hot set getting essentially all of the access).

The simulation is event-based. A transaction is created and queued at a node's CPU. It consumes its share of CPU and then requests data. (This inverts the real situation but should make no difference for the simulation.) Then we use the node's object location algorithm to service the data requests. Essentially we look at local cache, remote cache, local disk and remote disk in turn. The different location algorithms carry different costs for finding the objects in the system.

4.2 The Simulation Program

The simulation program that implements the model above is based on the SMPL written by MacDougall [13], which simulates caching between fast cache memory and main memory. But the DMMDB memory-memory boundary is sufficiently close that we were able to use SMPL as is. In our simulation program, his system bus is our network, his cache memory is our main memory, and his main memory is our disk. There is an independent SMPL queue for each CPU and disk; there may be more than one network connecting the nodes, each network having its own queue. All these queues are serviced according to FIFO discipline.

In addition to the three redundancy management components, the simulation program can also easily accommodate changes in some other important parameters. For example, the input to the simulation includes the distribution of requests over the database. Currently we are using a negative exponential distribution to determine object use frequency. This reflects a hot-set curve, whose parameters can be tuned easily. Of course, the LUV calculation formula is easy to change and the effects tested.

The run-time simulation parameters will expand as the simulation program is refined. Currently they include:

1. Total number of nodes and number of objects.

2. Maximum memory cache size.

3. Read/write request ratio.

cost parameters	local disk read	local disk write	cache access	RPC overhead
cost values	50ms (disk)	55ms (disk)	1ms CPU	3ms CPU

Table 1: Simulation Parameters

4. Network topology (e.g. bus or ring).

5. Costs of elementary operations, such as local cache location (CPU), local disk read (disk), and messages (network).

Currently, we assume that an update creates an immutable version, which can be replicated without further concern about replication consistency. Since we want to use valued redundancy to improve availability as well as performance, we will experiment with different algorithms to maintain replication consistency, such as available copies [3] and regeneration [16].

4.3 Performance Results

We made some additional simplifications with our initial simulation experiments. The read transactions query only one object and the write transactions update only one object. This makes the transaction cost uniform and the object creation cost uniform. Although these are potentially important value parameters, we made those assumptions to study each parameter in isolation.

Before we discuss our simulation results, we note that the simulation model is very robust and batched runs of the program (10 each time) produce figures that are very close, usually within 1% of each other. We will perform the usual statistical analysis for the final version of the paper.

Our first set of graphs show performance gains due to caching. We have assumed that the DMMDB access is not uniform over the objects, i.e. it exhibits locality of reference. This phenomenon, called hot-spots, is often verified in practice. In the simulation program, the objects are labeled from 1 to n, where $n = 5000$ for the curves shown in this paper. Our object reference pattern is a negative exponential curve shown in figure 1. The graphs in figures 2 and 3 show a simple value calculation algorithm for baseline study. The algorithm is essentially Not Frequently Used with aging. Each object access increases its value by a fixed amount; and periodically all the object values are divided by a fixed factor.

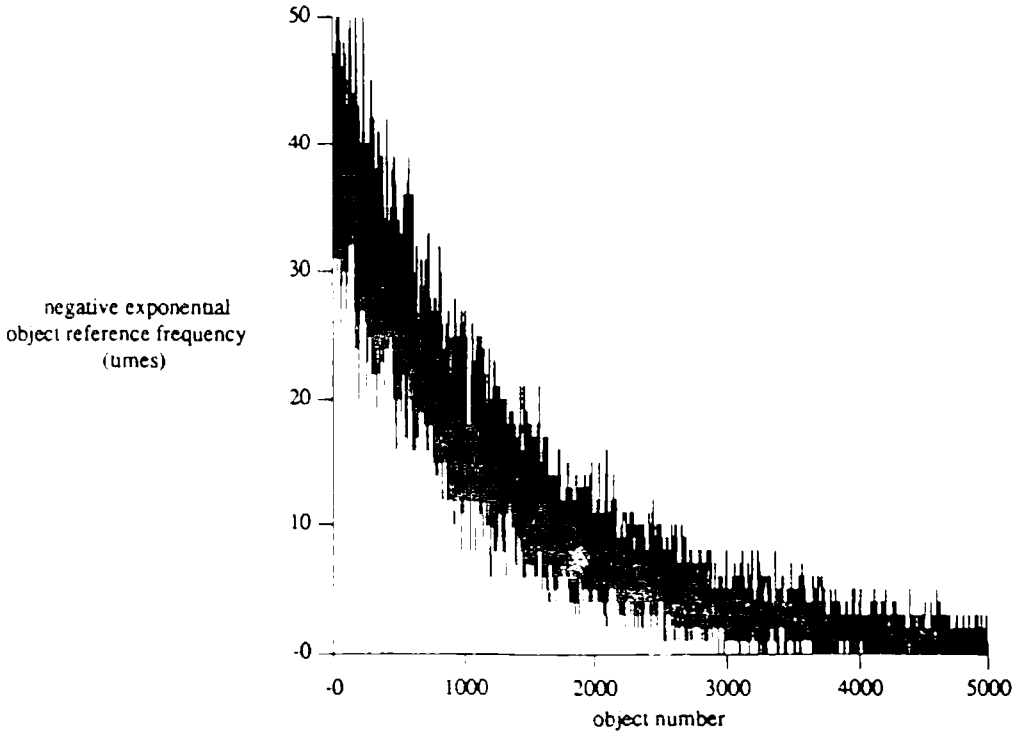

Figure 1: Object Access Pattern

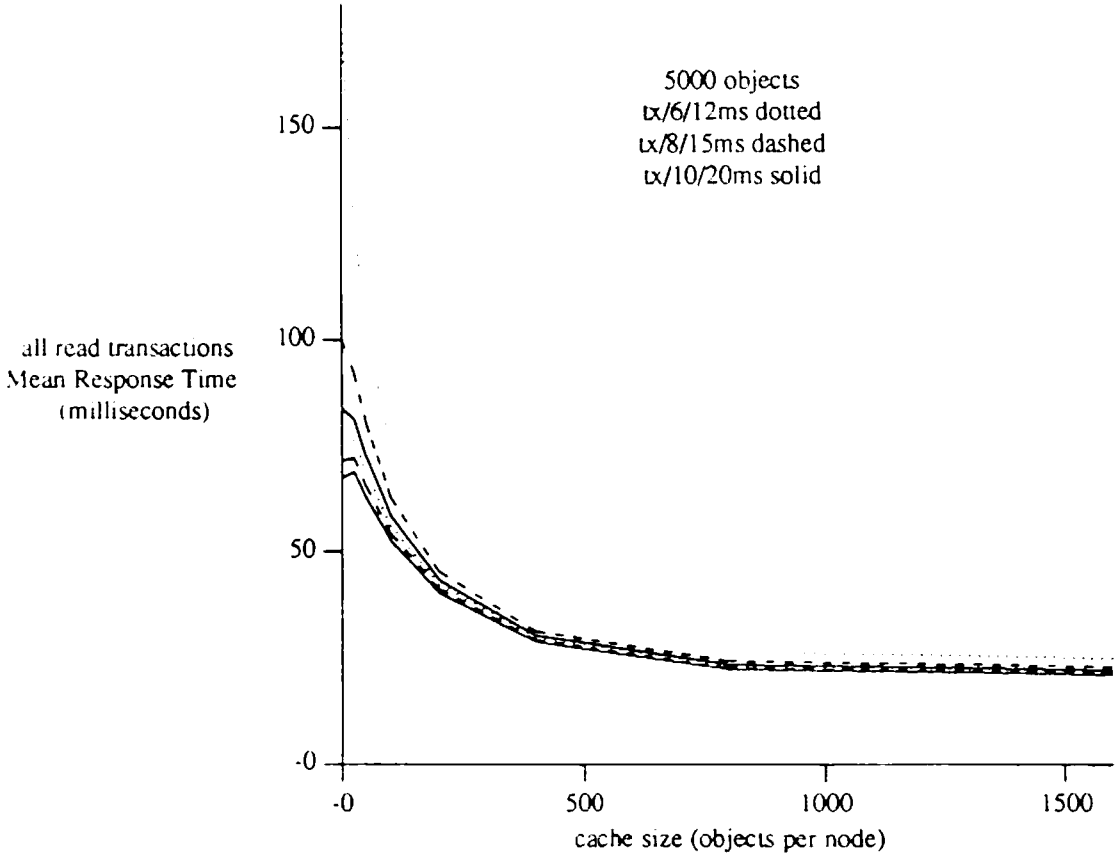

Figure 2: All Read Transactions with Caching

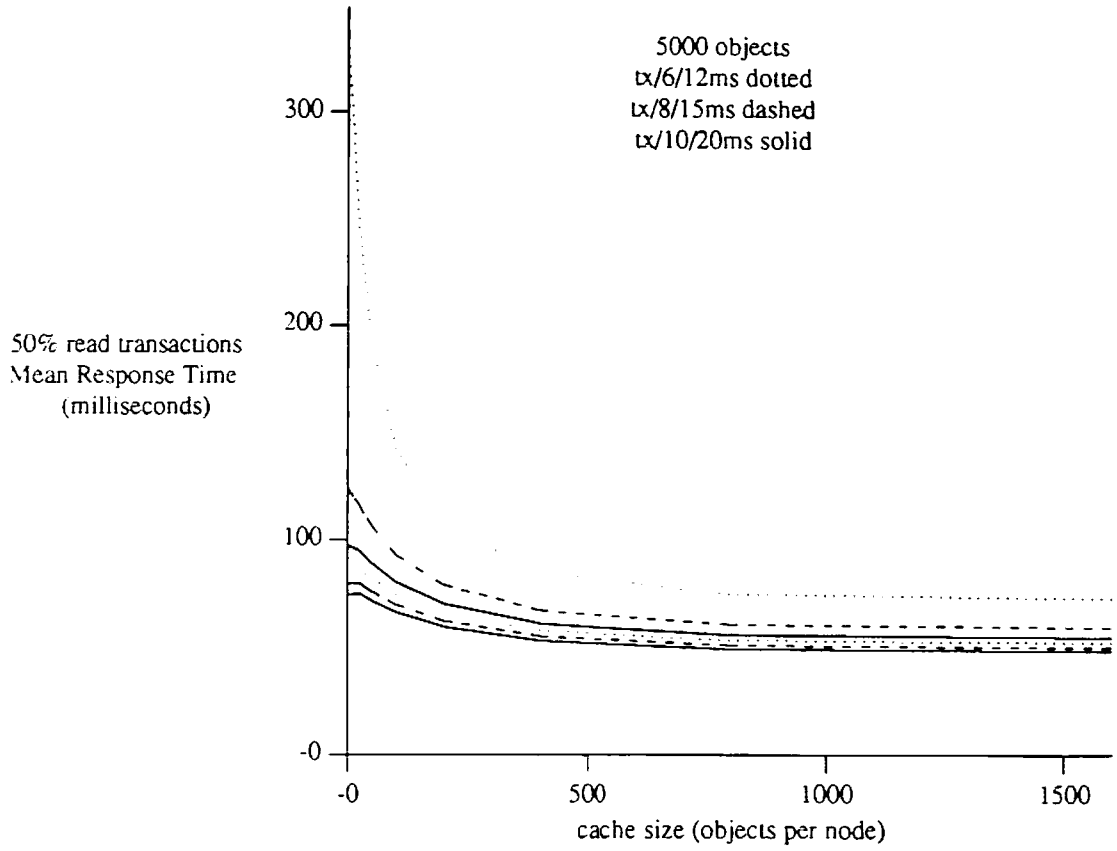

Figure 3: Half Read Transactions with Caching

Figure 2 shows the mean response time of read-only transactions as a function of cache size. The database consists of 5000 objects distributed over 10 nodes (500 objects each). For each simulation run, a node stores from 12 to 1600 objects in its main memory cache. Other important system parameters are included in table 1. The curves represent different loads on the system, from transactions arriving at every 6ms (exponentially distributed transaction interarrival rates, with the number being the mean) to 20ms. We can see that the mean response time decreases remarkably with increased cache size. Since write transactions do not benefit from caching, with only 50% of the transactions writing we have less benefits from caching, shown in the figure 3.

The system cost parameters can be changed for each simulation run. We have chosen the values according to the current workstation architecture. Each transaction consumes 10ms of CPU in its user code. Each message over the network takes 0.5ms to be delivered (occupies the network for that amount of time). Local cache requests take 1ms of CPU to be serviced. This explains the slight downward curve at cache size zero, since we avoid the cache management cost when the cache is turned off.

So far, we have shown that caching wins when there are hot-spot. Now we want to show that valued redundancy wins even when simple replacement algorithms do not. One way to defeat the replacement algorithms is to make the object access uniformly distributed, eliminating the

hot-spot. The only value parameter we will consider is the distinction between read access (redundancy desirable) and write access (redundancy undesirable). We have modified the page replacement algorithm to increase object value for read and decrease it for write, if not a local object. Therefore, even though the object access is randomly distributed over the database, the object value will be able to distinguish those to be replicated from the others. Unfortunately, at the time of this writing the simulation program was not able to produce the results due to programming problems.

Our current investigation is much broader than the simulation program. The additional directions include the cost factors, the usage patterns, and the performance and availability goals. Important cost factors that we want to explore are object size and object creation cost. Disparate usage patterns such as reference locality and sequentiality can be easily modeled with values. Performance goals, such as guaranteed response times can be assured with object values high enough to keep them in the main memory. Availability goals can be achieved with the same mechanism.

4.4 Availability Results

We have instrumented the simulation program to collect availability data. The basic idea is to use the memory-disk redundancy inherent in a DMMDB to continue answering queries even when nodes go down. Even though DMMDB is relatively new, this kind of memory-disk redundancy exists in many existing distributed systems. For example, in a client/server environment, the client cache could be used to improve system availability when the server goes down. It is necessary that the client be able to answer queries, which is the case with a DMMDB.

At this moment, we have a very simple failure model. The nodes are fail-safe (either up or down) and we have not considered network partitions. The simulation program starts up running normally, with all the (10) nodes sending and receiving transactions. After it has reached the stable state, with the caches filled, we "crash" a fixed number of nodes. The crashed nodes stop sending transactions and refuse to answer any queries regarding the data residing on them. But the up nodes can answer the queries if they have the object in memory cache.

As it turns out, since the memory cache contains the most valuable objects, the system (read) availability increases significantly as the cache size increases. Because our DMMDB does not include any redundancy at the disk level, write availability is limited by the accessible nodes. With object redundancy at the disk level and a consistent update algorithm such as regeneration [16], we would have system write availability as high as the read availability.

Figure 4 shows the availability as a function of cache size and the number of nodes down. The

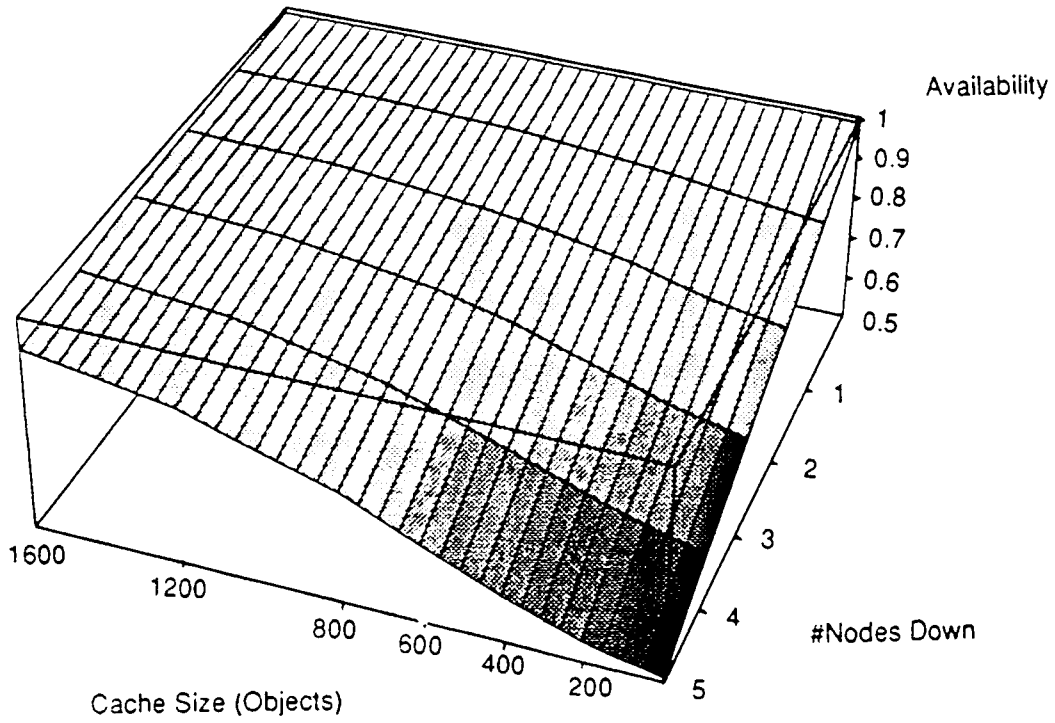

Figure 4: Availability vs. Cache Size vs. Down Nodes

simulation parameters are exactly the same as in Section 4.3. We show only the results for the transaction interarrival time of 20ms; the simulation results for the other transaction interarrival times with manageable queueing effects (from 15ms to 35ms) are essentially the same. Figure 5 shows the increase in availability due to memory-memory redundancy, as a function of cache size and number of down nodes. Both figures show availability gains with a simple caching algorithm. We plan to perform further availability experiments with object values that take this factor into account.

5 Related Work

5.1 Caching

In this section we summarize the cache work in the context of computer architecture caching and distributed file system caching. Another relevant area is database query value caching, to be discussed in the next section (5.2).

Traditional architecture caching is a good example of vertical redundancy. In a centralized system, evaluation of caching mechanisms studies only the interactions between the slow main memory and the fast cache memory. The situation changes in multi-processor systems that

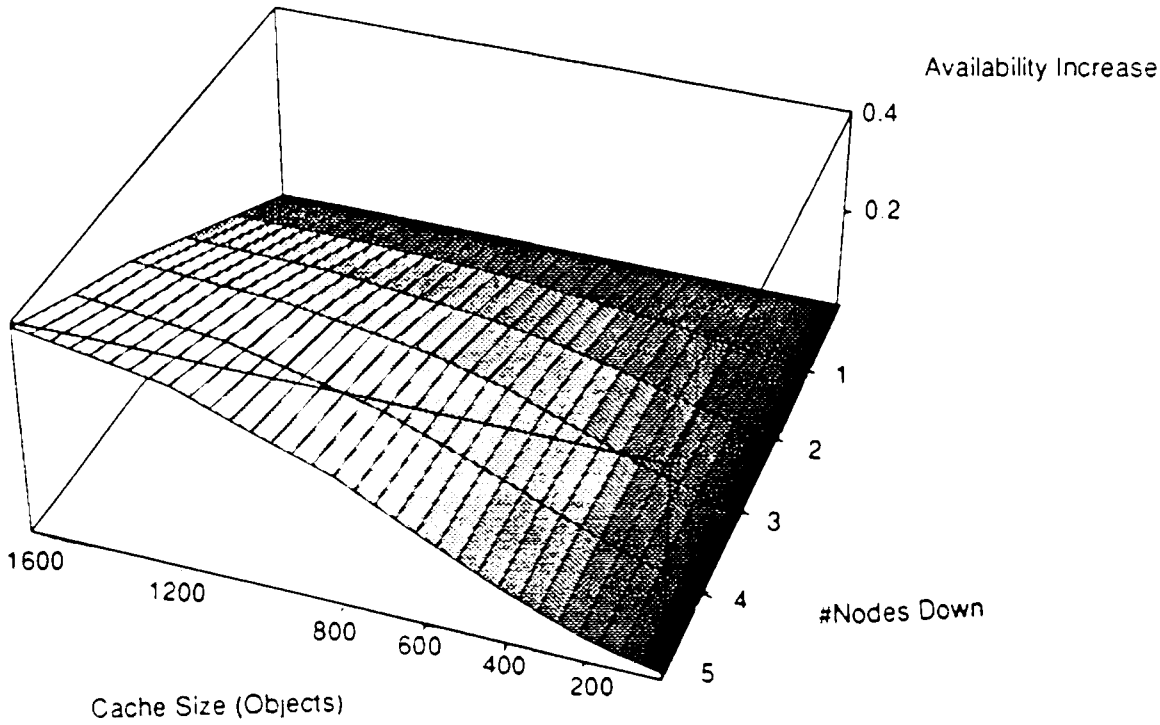

Figure 5: Caching Increases Availability

share a common memory bus, where each processor may have its own cached value of a memory location. Since the memory is shared, we need to keep the caches consistent, in a problem called *cache coherence*. A typical solution for cache coherence is called snooping, in which each processor observes the memory bus traffic to find values of interest. Several snooping protocols [2] (such as those in DEC Firefly and Xerox Dragon) use direct cache-cache transfer for shared blocks (if a requested block is in another cache, the block is loaded from that cache, not from main memory). In this respect, our memory-memory redundancy is analogous to snooping caches, but at a higher level in the memory hierarchy.

A common way to organize a distributed file system is to make it a server and the readers clients. In the Sprite experimental network operating system [14], blocks of files are cached in the main memory of both the server and the client. Sprite caches data on demand and uses the Least Recently Used replacement algorithm. Another example is the Andrew [9] file system, which caches entire files on local disks. However, neither Sprite nor Andrew make explicit use of the memory-memory redundancy provided by the servers caching from their local disks. Sprite emphasizes redundancy between the client memory and server disk, while Andrew emphasizes disk-disk redundancy between the server and clients. Valued redundancy can be seen as a generalization of this kind of caching, since we take into account not only read and

write frequency but also other parameters such as the cost of object creation.

5.2 Redundancy for Performance

A buffer management system speeds up data access in databases as it does for operating systems in general. However, specialized database buffers [4, 5] can take advantage of the data structures and access patterns in a database management system. Most of the database buffer work has concentrated on the memory-disk boundary in centralized databases.

Redundancy improves database performance since queries can be answered more quickly. An example of more recent work is the idea of field replication [18], which refines the replication granule to individual data fields. Since a frequently accessed data field may be used in conjunction with several other fields, grouping a copy of the frequently used field with each of them will eliminate some I/O operations and functional joins. Unlike valued redundancy, field replication was presented as a static redundancy method.

Directly caching previously calculated query results also increases database performance. PACKRAT [10] stores the intermediate results of queries on disk (called a derived database) to shorten the processing of future queries. Periodically PACKRAT reorganizes the derived database to adapt to current query patterns. Another example is database procedures, which stores queries in the database. A performance analysis of database procedure processing [7] shows that redundancy wins depending on factors such as frequency of updates and size of objects. Valued redundancy takes into account these factors (and other relevant ones) to dynamically adapt to maximize the performance benefits with minimal cost.

5.3 Replication for Availability

On the availability side, most of the work focuses on consistency maintenance algorithms, such as weighted voting [6], available copies [3], or regeneration [16], when nodes go down or the network partitions. As noted in the discussion on the implementation issues (Section 2.2), valued redundancy is orthogonal to the consistent update algorithms.

Another active area of replication research is the analysis of availability provided by the update propagation algorithms (for example, see [12]). Our work uses simulation to evaluate the availability provided by memory-memory redundancy, independent of the consistent update algorithms. A natural extension of our work will study the influence of the three components in the redundancy management model (including valued redundancy) on system availability.

5.4 Replication Policies

Good memory allocation is important for performance in shared-memory multiprocessors. Since optimal static allocation is difficult, dynamic allocation may be the best solution. One example is the pivot mechanism [17] which regulates dynamic migration of pages. The pivot mechanism is implemented as a set of counters per memory page which measure the "direction of imbalance." When a page's counters reach a preset value, that page is migrated one neighbor in the appropriate direction. The use of the counters is very similar to that of values in valued redundancy, the difference being that the counters are used exclusively for adaptiveness whereas values can be used to maintain the correct degree of redundancy and to incorporate adaptiveness. The full range of information provided by values helps the migration policy to make better decisions [8].

Valued redundancy assumes that the database access exhibits some kind of regularity. This assumption is made by all research on virtual memory, caching and buffering. In the database field, this assumption translates into the existence of hot-spots. Empirically, different kinds of regularity have been observed in database transaction trace analysis work. A recent trace analysis [11] made on 25,000,000 block references from 350,000 transactions shows that several kinds of behavior (e.g. locality of reference and sequentiality) have been observed. Even though locality of reference conflicts with sequentiality, each behavior is stable over a period of five days. Therefore, the ideal redundancy management system should be able to identify these different and possibly conflicting kinds of behavior and "lock on" to them. Valued redundancy can do just that.

6 Conclusion

We have introduced the idea of valued redundancy. The goal of valued redundancy is to replicate only the most valuable objects in the system, i.e., those that contribute the most to system performance and availability. The object's value explicitly represents its cost/performance ratio for the redundancy management system. This way, we maximize *both* performance and availability while minimizing the redundancy cost.

We have written a simulation program to study the potential performance and availability gains of valued redundancy in a distributed main-memory database. The simulation models today's environment of a network connecting many powerful workstations. We have investigated the memory-memory redundancy between the local memory and remote memory of the database resident in the main memory of the workstations. Even though we just started taking data, the results are very encouraging.

The simulation shows substantial performance and availability increases provided by an integrated redundancy management system. For example, in a database of 5000 objects distributed uniformly over 10 nodes and object access determined by a negative exponential probability function (modeling a small number of hot-spots and relatively uniformly distributed access over the rest of the database), we used a simple aging algorithm to manage a local cache of objects. For a cache size of 800 objects per node, we obtain an apparent availability of over 90%, even though 3 nodes have gone down (30% of the database inaccessible). For the same parameters (but all nodes up), a configuration of all-read transactions runs at the mean response time of 22ms in contrast to non-cache baseline case of the same load that runs at the mean response time of 67ms. In other words, a substantial cache (a total cache size that is 160% of the database size) can improve the system response time (in this simulation also the throughput) 3 times.

Valued redundancy integrates the main components of a redundancy management system: the object location algorithm, the page replacement algorithm, and the consisten update algorithm. This integration implements a smart replication policy of maximizing performance and availability while minimizing the maintenance costs. We are currently exploring the space of redundancy management systems formed by different combinations of these components and the interactions between valued redundancy and each component.

Much work remains to be done in performance and availability analysis, since we have only begun our simulation study. Once we are satisfied that we have found a good combination of algorithms and value calculation formula, we intend to implement the redundancy management system in the database manager on top of the Synthesis operating system [15] being developed at Columbia.

References

[1] D. Agrawal and S. Sengupta.
Modular synchronization in multiversion databases: Version control and concurrency control.
In *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data*, Portland, 1989. ACM/SIGMOD.

[2] J. Archibald and J-L. Baer.
Cache coherence protocols: Evaluation using a multiprocessor simulation model.
ACM Transactions on Computer Systems, 4(4):273–298, November 1986.

[3] P.A. Bernstein and N. Goodman.
An algorithm for concurrency control and recovery in replicated distributed databases.

ACM Transactions on Database Systems, 9(4):596–615, December 1984.

[4] W. Effelsberg and T. Haerder.
Principles of database buffer management.
ACM Transactions on Database Systems, 9(4):560–595, December 1984.

[5] K. Elhard and R. Bayer.
A database cache for high performance and fast restart in dataase systems.
ACM Transactions on Database Systems, 9(4):503–525, December 1984.

[6] D.K. Gifford.
Weighted voting for replicated data.
In *Proceedings of the Seventh Symposium on Operating Systems Principles*, pages 150–162. ACM/SIGOPS, December 1979.

[7] E.N. Hanson.
Processing queries against database procedures: A performance analysis.
In *Proceedings of 1988 ACM SIGMOD Conference on Management of Data*, pages 295–302, June 1988.

[8] M.A. Holliday.
Reference history, page size, and migration daemons in local/remote architectures.
In *Proceedings of the Third Symposium on Architectural Support for Operating Systems and Programming Languages*, pages 104–112, Boston, April 1989.

[9] J.H. Howard, M.L. Kazar, S.G. Menees, D.A. Nichols, M. Satyanarayanan, R.N. Sidebotham, and M.J. West.
Scale and performance in a distributed file system.
ACM Transactions on Computer Systems, 6(1):51–81, February 1988.

[10] N.N. Kamel.
The Use of Controlled Redundancy in Self-Adaptive Databases.
PhD thesis, Department of Computer Science, University of Colorado, 1985.

[11] J.P. Kearns and S DeFazio.
Diversity in database reference behavior.
In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems*, pages 11–19. ACM/SIGMETRICS, May 1989.

[12] D.D.E. Long and J-F Paris.
Regeneration protocols for replicated objects.

In *Proceedings of the Fifth International Conference on Data Engineering*, pages 538–545, Los Angeles, 1989.

[13] M.H. MacDougall.
Simulating Computer Systems.
Computer Systems. MIT Press. 1987.

[14] M.N. Nelson, B.B. Welch, and Ousterhout J.K.
Caching in the sprite network file system.
ACM Transactions on Computer Systems, 6(1):134–154, February 1988.

[15] C. Pu, H. Massalin, and J. Ioannidis.
The Synthesis kernel.
Computing Systems, 1(1):11–32, Winter 1988.

[16] C. Pu, J.D. Noe, and A. Proudfoot.
Regeneration of replicated objects: A technique and its Eden implementation.
IEEE Transactions on Software Engineering, SE-14(7):936–945. July 1988.

[17] C. Scheurich and M Dubois.
Dynamic page migration in multiprocessors with distributed global memory.
In *Proceedings of the Eighth International Conference on Distributed Computing Systems*, pages 162–169, September 1988.

[18] E. Shekita and M. Carey.
Performance enhancement through replication in an object-oriented dbms.
In *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data*, Portland, 1989. ACM/SIGMOD.