# Redundancy Management in a Symmetric Distributed Main-Memory Database

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

Technical Report No. CUCS-014-90

## Abstract

We describe the architecture of a symmetric distributed main memory database. The high performance networks in many large distributed systems enable a machine to reach the main memory of other nodes faster than local disks. Thus we introduce *remote memory* as an additional layer in the memory hierarchy between local memory and disks. Exploiting the remote memory (every node's cache) improves performance and increases availability. We have created a simulation program that shows this significant advantage over a wide range of cache sizes and compares the effects of several object replacement policies in the symmetric distributed main memory database.

# Contents

**1 Introduction** **1**

**2 Simulation Study** **2**

2.1 The Architecture . . . . . 2

2.2 The Memory Hierarchy . . . . . 3

2.3 Object Location . . . . . 4

2.4 Update Propagation . . . . . 4

2.5 Replacement Algorithm . . . . . 5

2.6 The Simulation Program . . . . . 5

**3 Analysis of Results** **7**

3.1 Remote Memory Performance Gains . . . . . 7

3.2 Biased Read-Write Access . . . . . 10

3.3 Remote Memory Availability Gains . . . . . 13

**4 Related Work** **15**

**5 Conclusion** **15**

**References** **16**

# 1 Introduction

Data redundancy uses hardware resources to improve performance, as in caching and buffering, or to increase availability, as in mirrored disks. In large distributed systems, vast amounts of hardware are connected together by fast networks ranging from local-area networks to the envisioned National Research and Education Network. These networks introduce a new level in the 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*. 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 call (RPC) implementations over Ethernet take only a few milliseconds round trip [8]. Moreover, the bottleneck in communications protocols is CPU and software overhead. With RISC technology doubling CPU speed every few years, we can expect even better remote memory access times in the near future. Furthermore, faster gateways and higher bandwidth will steadily bring down the communications overhead over wide-area networks.

In these large distributed systems, however, already hard questions such as "*how many* copies should we keep" and "*where* do we put the copies" become even more difficult to answer. We can easily fail to realize the potential of high performance and availability stemming from the redundancy inherent in such systems. For example, many distributed software systems are organized according to the client/server model. Its main advantage is simplicity, since the server resembles a centralized system. However, the asymmetry in the client/server model severely restricts resource sharing. For instance, useful data in a client buffer are inaccessible to other nodes in the system. From the client/server point of view, this problem disappears if we have enough buffer space in the server. This answer, however, begs the question of how to better utilize the ample memory resources in a large distributed system.

In this paper, we study a symmetric architecture in which all nodes of the database system are accessible to one another. Concretely, we distribute data and load, blurring the distinction between clients and servers. In particular, all nodes in the database can "serve" requests if their buffers contain the requested data item. This differs from the client/server model, in that we have a number of relatively small symmetric nodes instead of a powerful central server. We show in this paper that the large amount of remote memory available in the symmetric architecture makes a significant contribution to system performance gains.

Most replication work focuses on a particular update propagation algorithm, or on a replica consistency algorithm that handles node crashes and network partitions. Instead of studying these replication *mechanisms*, we concentrate on replication *policy*, focusing on the relatively

unexplored area of remote memory redundancy. We have chosen a representative set of simple object location, update propagation, and consistency maintenance algorithms. Given these choices, we examine the performance and availability improvements achieved by the management of object redundancy.

# 2 Simulation Study

We have created a simulation program to evaluate the performance and availability gains of a symmetric distributed main-memory database (SDMD). Our simulation model abstracts details from the SDMD scenario that do not affect the performance and availability issues that we are investigating.

The simulation is implemented on top of the SMPL subsystem [5]. The model's building blocks are the CPUs, the network, main memory and disk resources: higher level simulation activities are composed of requests to these simpler resources. For example, a remote disk read is composed of a request to the remote node, a local disk read there, and a reply to the requester. Each resource has its own priority queue, which is serviced according to a FIFO disciplne. Software costs such as cache access and maintenance are charged in terms of CPU service time and the number of network messages.

The simulation model has a fixed skeleton, which is the SDMD, and three movable parts, which are the components of the redundancy management system (described in sections 2.3, 2.4, 2.5), – i.e., the location algorithm, the replacement algorithm, and the consistent update algorithm respectively. We first describe the skeleton architecture, and then discuss the components of the redundancy management system.

## 2.1 The Architecture

The hardware model underlying a SDMD is a collection of computer systems connected by a medium- to high-performance communications network. Each computing system has one or more modern microprocessor CPUs, relatively large main memory (e.g. 16 MBytes per CPU), and sufficient secondary storage (e.g. magnetic or optical disks) to hold a portion of a distributed data base. Such hardware bases are common in academia and industry.

In our simulation, we model each node as a processor and disk. The processor contains a CPU and a portion of local (main) memory available for caching. We chose an ethernet as an instance of the communications network. A sketch is shown in Figure 1. CPUs, disks and the communication network have associated priority queues which are serviced according to FIFO discipline.

Figure 1: Distributed Main-Memory Database Architecture

| cost parameters | **local disk read** | **local disk write** | **cache access** | **RPC overhead** |
|---|---|---|---|---|
| cost values | 50ms (disk) | 55ms (disk) | 1ms (CPU) | 3ms (CPU)<br>1ms (net) |

Table 1: Simulation Performance Parameters

### 2.2 The Memory Hierarchy

The memory hierarchy of a SDMD 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).

In the simulation, an access cost is associated with each level of the memory hierarchy. These costs are listed in Table 1. If the communications network is slow, acting as a bottleneck, then there will be an even greater difference between local and remote storage access costs. This factor is included in simulation parameters, but we have not yet pursued it as a significant source of

SDMD performance gains. We ignore CPU hardware caches completely. Their effects are local, extremely fine-grained and, being implemented in silicon, not subject to software solutions.

Within this memory hierarchy, our simulation manages a set of objects representing the contents of the database. At the disk storage level, the set of objects is completely partitioned: there is no disk-disk redundancy between sites. Memory-disk redundancy of objects is managed locally, in conjunction with the memory-memory redundancy as detailed below in section 2.5.

## 2.3 Object Location

There are three design decisions in a SDMD's redundancy management system: (1) the object location algorithm, (2) the replacement algorithm, and (3) the consistent update algorithm. In the simulation, we have adopted simple and realistic algorithms.

Object location algorithms find object copies in the system. There are two ways to locate objects in the system: either a copy is found in cache or the object is located on disk. If a site fails to find a copy in local cache, we use a broadcast request with individual replies and timeout. After queuing, each broadcast uses the network for half a millisecond. At every remote site, the object request interrupts any local transaction processing to emulate OS support and costs one cache access (one millisecond of CPU). All sites with a copy of the requested object in their caches queue for the network and reply: the requesting node simply discards all replies after the first. Expiration of a five-millisecond timeout period indicates that the object is not cached in any remote node; the object must then be fetched from remote disk. A table lookup determines which site has the object. In other words, when we need a copy of an object we traverse the memory hierarchy looking at local cache, remote cache, local disk and remote disk in turn.

## 2.4 Update Propagation

Consistent update algorithms keep all copies of an object mutually consistent. Most of the recent data replication work focuses on this component of redundancy management, for example, Majority Voting (also called Quorum Consensus). The amount of work required for each kind of object access depends on the particular update algorithm. Also, algorithms may become more complex as more kinds of failures (e.g. network partitions) are considered.

We use broadcast to propagate update values across the system. This takes half a millisecond on the network and a cache access for each node. This is a "read-one, write-all" strategy for maintaining replica control [2, 7]. In this simulation, in which we either read or write only one object at a time, timestamping the update messages suffices to maintain copy consistency.

## 2.5 Replacement Algorithm

Unbounded redundancy will eventually exhaust the SDMD's storage capacity. The criteria for choosing which copies reside at a given level of memory hierarchy are embodied in the replacement algorithms, so named for their similarity with the page replacement algorithms in virtual memory, such as Least Recently Used (LRU). The replacement algorithms are invoked when the storage capacity of a given level in the memory hierarchy (e.g. main memory) is full, to keep the most valuable objects and throw away the others. This component is one of the major parameters of our simulation study: specific replacement strategies are discussed in section 3.

Our "replacement algorithm" has two parts, the object value calculations and the memory management. Object value calculations include information from both the location and update algorithms. The memory management part consists of the resource manager that uses object values to decide which objects to replicate and which ones to throw away. A remote disk read request requires the object to be read into the remote node's memory before transmission. But the memory (cache) management for both the memory-memory and memory-disk redundancies is the same – it is based on assigned values.

Note that an object's value depends on many factors in the system. Each redundancy management component contributes the factors it affects to value calculation. 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.

We made a deliberate decision to define object value as a function of exclusively local parameters. Maintaining the accurate value of a global parameter is costly in a distributed system and even more so when the network scales up. Using an out-of-date global value contradicts the basic idea of value estimation – that an object's value reflects its current worth and adapts to changing situations. The algorithm therefore keeps objects with the highest local values in main memory and throws away those with low local values. The objects of highest value have the highest degree of redundancy and are thus the most readily available.

## 2.6 The Simulation Program

In our simulation, a central transaction server creates transactions with exponential inter-arrival times and distributes them uniformly over the nodes of the SDMD. The probability of a transaction accessing a particular object is specified by a parameter that we term "access locality". This

is similar to the notion of locality of reference in virtual memory in that access to objects is *not* distributed uniformly over the database. Access locality has the effect of creating a "hot-set" of accessed objects within the overall database. In our model of a hot-set we use an exponentially decaying curve, with a small set of objects receiving most of the accesses. Transactions access objects in either read or write mode. The mixture of read and write accesses is specified by a Bernoulli distribution.

In figure 2 we show the execution of a write transaction. The local node determines which part of the database holds the object on disk. A message is sent via the network to that processor to obtain a copy of the object. When the message arrives it is queued at the remote CPU. After processing by the CPU, the request is then queued at the remote disk. Once the object is returned from the remote disk it is sent over the network (queuing on the network queue again) to the local processor. The transaction can then execute at the local CPU: the modified object's value is propagated to all sites by an application of the consistent update algorithm. We optimize the simulation of CPU use by summing a transaction's CPU usage and then charging that usage as one block.

Figure 2: Transaction Overview

In figure 2 we also show the execution paths of a read transaction. A read transaction differs from a write in that is uses the full location algorithm and has no need to invoke the update

propagation algorithm. The object location algorithm is invoked, checking local cache, remote caches and then disks in turn. In order to achieve the goal of fast performance, cache requests must be able to bypass executing transactions. This is done by assigning cache requests a higher priority than either disk or CPU processing jobs. As soon as the object is located, a copy is placed in local cache and used to execute the transaction. Costs are accounted in the same manner as write transactions.

Table 1 highlights the resource performance parameters while table 2 notes the remaining major parameters.

| Parameter | Value(s) |
|---|---|
| Computing nodes | 10 |
| Networks | 1 |
| Objects | 5000 |
| Object Cache | 0 to 1600 objects |
| Read/write mix | read only to write only |
| Transaction cost | 10 ms of CPU |

Table 2: Model Parameters

# 3 Analysis of Results

In reporting both mean transaction response-time and availability, the batched-means method was used. We ran the simulations until the relative half-width of the confidence interval for the response time statistic, averaged over a minimum of ten batches, was 0.05 or less.

## 3.1 Remote Memory Performance Gains

The database contains 5000 objects, numbered from 1 to 5000. Figure 3 shows the effectiveness of remote memory for a representative cache size (500 objects at each node), plotting the number of accesses for each object. The outermost negative exponential curve sums up all the object accesses during the simulation run and indicates the "access locality". In other words, the negative exponential models a non-uniform object access pattern, with the hot spot made up by objects with small numbers. Roughly, we denote an object's "temperature" by its number, the smaller the hotter.

Because of the existence of the hot-set, if a site can get most of the hot objects into its cache, performance will improve. The second, lower solid curve (starting just above 200) in figure 3 showing negative exponential decay, describes the number of local cache hits for each

Figure 3: Object Access in Memory Hierarchy

object. The replacement algorithm used here is Not Frequently Used with aging, i.e., each site approximates classical LRU behavior. As expected of a successful local caching algorithm, the hotter an object is, the more likely we will find it in the local cache. However, since the local cache is limited by its moderate size (10% of the total objects), the curve drops off quickly.

As the local curve drops off, the remote cache becomes increasingly important. Even if a hot object was swept out of a given site's cache, it is quite likely that it will be found in another site's cache. The successful hits on remote cache are shown by the dotted curve in figure 3. Since the remote cache is much larger than the local cache (9 times in this simulation), it is not surprising that the total number of remote hits is much higher than the local hits. The hottest objects have about the same number of local hits as the remote hits, but the object number 1000 has more than 4 times remote hits than local hits. As the network grows, the remote cache size grows and we can expect even higher gains.

In contrast, the residual disk access is shown as a bell-shaped curve. On the hot end, only a few requests had to be satisfied by going to disk. This is due to the success of both the local and remote caches. On the cold end, the bulk of requests had to be met by going to disk. Overall, greater total cache sizes tend to flatten the bell-shaped disk access curve. Intuitively, since the area under the curves represents the total time spent on object access,

Figure 4: All Read Transactions with Caching

the area between the outermost curve and the bell-shaped disk curve would show the system performance improvements obtained through caching.

Instead of throughput (figure 3), we use the mean response time to compare the performance gains for different transaction arrival rates. Figure 4 shows the mean response time of read-only transactions as a function of cache size. The curves represent different loads on the system, from transactions arriving every 6ms to every 20ms.[1] The mean response time decreases markedly with increased cache size. Since write transactions do not benefit from caching, with 50% of the transactions writing we have fewer benefits from caching, shown in the figure 5.

Once we have determined the effectiveness of remote cache, the next step is to investigate how the remote cache works. In figure 6 we study the degree of object replication as a function of cache size. Every object either is resident only on disk (0 copies in cache) or is cached (1 to 10). On one axis, we have the number of copies. For each number of copies (the second axis) we have the number of objects with that many copies (population size, divided by 1000 for display in the graph). On the third axis we vary the cache size (divided by 100 for display). At one extreme (cache size 0), we have 0 for 1 to 10 copies and 5000 for 0 copies (off scale). As the cache size grows, we see the objects have small number of copies (up to cache size 1000). When

[1] Transaction interarrival rates are exponentially distributed. The range specified is that of the means.

Figure 5: Half Read Transactions with Caching

the cache size approaches 2000, the copy population curve shows two humps. The prominent hump at 1 copy simply reflects the drop of objects not in cache when the cache size increases. The smaller hump between 7 to 9 copies has not been explained.

From figure 6 we can see that remote cache management is not a simple matter. The decision between allocating more copies to a hot object or a single copy to several warm objects is certainly a non-obvious tradeoff. More copies of a hot object wins because local accesses are cheaper. But a single copy can serve the entire database at remote cache cost, without the penalty of maintaining the many copies during updates. This and other complications that arise in a multi-level memory hierarchy with varied performance parameters led us to the experiment on valued redundancy, to be described in the next section.

### 3.2 Biased Read-Write Access

In this experiment, we compare several replacement algorithms. First, in the above investigation of the benefits of memory-memory hierarchy, we used "Not-Frequently Used with aging" (approximate LRU, simply called LRU from now on). One way of looking at this algorithm is that each site dynamically updates a snapshot of the hot-set curve. An object will tend to have a high value dynamically if it also has a high access frequency in the static hot-set curve. On

Figure 6: Degree of Object Replication vs Cache Size

the other hand, because the global access curve is distributed over the entire system, and the sites update their caches locally, if an object is not resident in a given site's memory it may still be memory-resident in another site.

Both of the access frequency and the remote residency effects may be seen in figure 3 where the number of requests for each object is divided into local cache, remote cache, and disk accesses to satisfy the request. Although both the local main memory and remote main memory approximate the hot-set curve, over time more requests are met by remote cache. As a result, the first algorithm takes advantage of the one non-random object access characteristic –access frequency– without having to hardwire this information into the database.

Now, instead of modeling all objects as having identical read/write characteristics we assume that objects can belong to one of a number of read/write categories. If two objects have the same access frequency, then a site will want to maintain in main memory the object with greater read access because a write request must always go to the disk anyway.[2] An object's access pattern now has two components –overall access frequency and read/write frequency– and we seek to incorporate both components in the replacement algorithm. In addition, we want each site to continue to do this estimate in a distributed and dynamic fashion.

[2] This heuristic depends on the particular update propagation algorithms and location algorithms.

Figure 7: Performance of Three Replacement Algorithms

To investigate whether a replacement algorithm can take advantage of this non-random object access characteristics, we have the second replacement algorithm, called *valued redundancy*. We calculate an object's value by weighting its LRU estimate with its read/access ratio (number of reads divided by total number of accesses). The read/access ratio is accumulated dynamically and aged in the same way as the LRU estimate. If the object access is all-writes, its value will be zero, indicating that despite its high access rate, it does not pay to keep it in cache. At the other extreme, a read-only object's value will simply be the usual LRU estimate. In general, an object's value may use more sophisticated formulas, taking into account other factors.

The third algorithm is what we call "static optimal", because the sites simply maintain in cache the most frequently accessed objects as determined by the global hot-set curve. It is optimal in the sense that the objects in cache are by definition the most accessed over the entire history of the simulation. It is static in the sense that the hottest objects are locked into the cache and never changed.

In figure 7 we contrast these three replacement algorithms in terms of mean response time as a function of increasing cache size for each replacement algorithm. The mean transaction arrival rate is fixed at 6ms while all other simulation parameters varied as shown in figures 4 and 5. The solid line represents the static optimal. The other two (dotted–LRU and dashed–valued

redundancy) are dynamic algorithms, which do much better than the static optimal. This happens because the short-term access pattern varies significantly, departing from the static hot-set curve, thus penalizing the static algorithm. Generally, dynamic algorithms capture the access pattern variance in time and in space commonly known as locality of reference.

In our simulation experiment, instead of a uniform $P_{read}$ (Bernoulli read probability) of 0.5 for all objects, as in figure 5, each object belongs to one of three read/write categories. Object $i$ is assigned to category $i \bmod 3$. Objects in categories 0, 1 and 2 are assigned $P_{read}$ values of 1.0, 0.3, and 0.2 respectively. The average $P_{read}$ is still 0.5, but now substantial variance exists between the objects of different categories. In figure 7, we see that for cache sizes between 10 and 250 objects, valued redundancy does better than LRU by up to 10%. As cache size gets larger, the performance of the two dynamic algorithms is virtually identical, since everything useful is in cache for both.

Our investigation on valued redundancy is much broader than the current 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 modeled with values straightforwardly. 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.

### 3.3 Remote Memory Availability Gains

We have instrumented the simulation program to collect availability data. The basic idea is to use the memory-disk redundancy inherent in a SDMD to continue answering queries even when nodes go down. Even though SDMD 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 SDMD.

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

Figure 8: Availability vs. Cache Size vs. Down Nodes

(read) availability increases significantly as the cache size increases. Because our SDMD 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 [7], we would have system write availability as high as the read availability.

Figure 8 shows the availability as a function of cache size and the number of nodes down. The simulation parameters are exactly the same as in Section 3.2. We show only the results for the transaction interarrival time of 10ms. Simulation results for other transaction interarrival times with manageable queueing effects are essentially the same. For small cache sizes, availability is reduced by exactly the number of disks that can no longer be accessed. In stark contrast, for larger cache sizes availability approaches that of a fully functioning system. As a result of the hot-set phenomenon, even at moderate cache sizes many of the valuable objects are resident in caches when the host site crashes. Consequently high availability is achieved without very large cache. We plan to perform further availability experiments with object values that take this factor into account.

## 4 Related Work

In a centralized system, evaluation of caching mechanisms only focus on interactions between the slow main memory and the fast cache memory. The situation changes in multi-processor systems that share a common memory, because each processor may have its own cache of a given object. Since the memory is shared, we need to keep the caches consistent, a problem called *cache coherence*. A typical solution for cache coherence is "snooping", in which each processor observes the memory traffic to find values of interest. Several snooping protocols [1] (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, and not from main memory. In this respect, our memory-memory redundancy is analogous to snooping caches, but at a different level in the memory hierarchy.

Recent work on page placement has studied very simple memory hierarchies in Non-Uniform Memory Access machines [3]. Their main conclusion is that simple algorithms can obtain essentially all the benefits of caching in the case of a simple memory hierarchy. For example, in the IBM ACE multiprocessor workstation, global memory access time is approximately twice that of local memory. Because of the small difference in access time, the placement mechanism must be simple and fast to be worthwhile. In contrast, we are interested in richer memory hierarchies.

A common way to organize a distributed file system is to make it a server and the readers clients. In the Sprite network operating system [6], 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 [4] 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. In our symmetric model, nodes act both as servers and clients.

## 5 Conclusion

We have described a symmetric distributed main-memory database architecture and studied the effects of careful redundancy management on its performance and availability. The salient feature of the architecture is node symmetry, because nodes provide each other data from their main memory through a fast network. The architecture captures today's environment where networks connect many powerful workstations. Even though we know that simple algorithms suffice for

simple memory hierarchies [3], the general symmetric architecture will require better caching and replication management to realize the potential for higher performance and availability.

We have written a simulation program to study the potential performance and availability gains of redundancy based on remote memory in a symmetric distributed main-memory database. For a wide range of cache sizes and object access patterns, we find the hit ratio of remote memory cache to be several times the hit ratio of local cache. This result demonstrates the importance of remote memory cache in the symmetric architecture, due to both its size and speed. Since the size of remote memory grows larger as the network grows and remote memory access becomes faster as the network evolves, remote memory cache will become ever more important.

Another interesting result is the significant availability gains with remote cache. With 30% of the objects residing on down nodes, we can answer more than 90% of the queries with a moderately-sized cache (local cache containing less than 20% of the total number of objects). In a large distributed system, there will always be nodes temporarily inaccessible. Since remote memory stores only the "hot" objects in volatile memory, we have gained added availability at no extra storage cost.

Finally, we have tested a new cache replacement method. We use a value-based redundancy policy in order to maintain the most valuable objects in cache. With a simple value formula, valued redundancy shows excellent promise in an experiment running biased read/write requests, gaining 10% over a significant range of cache sizes compared to a very good approximation of LRU.

## References


[1] 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.

[2] 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.

[3] W.J. Bolosky, R.P. Fitzgerald, and M.L. Scott.
Simple but effective techniques for NUMA memory management.
In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, pages 19–31, December 1989.

[4] 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.

[5] M.H. MacDougall.
*Simulating Computer Systems.*
Computer Systems. MIT Press, 1987.

[6] 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.

[7] 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.

[8] M. Schroeder and M. Burrows.
Performance of Firefly RPC.
In *Proceedings of the Twelfth Symposium on Operating Systems Principles*, pages 83–90. ACM/SIGOPS, December 1989.