1988

Three Experiments in Reliable Transaction Processing in RAID

Bharat Bhargava
Purdue University, bb@cs.purdue.edu

Fady Lamaa

Pei-Jyun Leu

John Riedl

Report Number:
88-782

THREE EXPERIMENTS IN RELIABLE TRANSACTION PROCESSING IN RAID

Bharat Bhargava
Fady Lamaa
Pei-Jyun Leu
John Riedl

CSD-TR-782
June 1988

Three Experiments in Reliable Transaction Processing in RAID *

Bharat Bhargava
Fady Lamaa
Pei-Jyun Leu
John Riedl

Department of Computer Sciences
Purdue University
West Lafayette, IN 47907

Abstract

This paper describes three studies that contribute towards the building of reliable distributed database systems. We measure the transaction processing times in the different phases of its execution in an operational database system called RAID. We find that response times of 300-500 milliseconds are achievable and much of it is due to the communications and commitment software. We have implemented and measured a concurrent checkpointing and rollback algorithm useful for dealing with failures of individual processes in the system. We find that the cost of synchronization for the coordinator process is of the order of the time required for taking a single checkpoint on the stable storage. We find that the concurrent execution does not reduce the message overhead or cpu usage but has the same communication delay as a single checkpoint instance. We show that much parallelism exists in such algorithms. Finally we present two more experiments that were done by implementing a partially replicated database system. They measured the effects of the degree of replication and of the threshold representing the minimum number of copies. We find that a low degree of replication (25%) and a high threshold can provide the same data availability as a fully replicated database but with lower response time.

*This research is supported by NASA and AIRMICS under grant number NAG-1-676, UNISYS, and a David Ross fellowship.

1 Introduction

Although much research has been done in designing algorithms for transaction processing, concurrent checkpointing and rollback, and replication control, few prototypes that incorporate and experiment with them have been implemented. Experimental data from systems doing real processing is scarce. Recently data has been available from the CAMELOT [16] and ARGUS [14] systems. Our mission in the implementation of RAID [6] has been to seek experimental data based on real processing. In this paper, we present experimental data that was obtained in three different studies conducted in the complete transaction processing system RAID.

In the first study, we measured the overheads for subsystems involved in transaction processing such as concurrency control and atomicity control. The atomicity control data contains the timing of the communication software. These studies validate the transaction processing times observed in the CAMELOT [15] system and give a clear indication of the various times in the transaction processing cycle.

In the second study, we report on the implementation and measurements conducted on the concurrent checkpoint and rollback algorithm that was published in 1987 [13]. This algorithm manages multiple global checkpoint and rollback instances that are initiated by different RAID servers. The sizes of a large number of actual object files in UNIX systems were studied to determine the cost of taking a sample checkpoint to stable storage. We ran several real instances of concurrent checkpoint and concurrent rollback trees involving two to ten servers and measured the delays due to synchronization and the sizes of the message queues at each server process. This study experiments with the overheads associated with such algorithms for the first time.

In the third study, we have conducted detailed experiments on the management of partially replicted databases. We measured the effects on data availability of automatically creating new copies of data when the replication level decreases due to site failures. We also measured how the number of out-of-date data objects varies with respect to the level of partial replication. A series of experiments done in a fully replicated database that measured the overheads for recovery, user, control, and copy transaction processing were reported in 1987 [5].

Another study that details a variety of experiments on the communication software including multicast implementation at the kernel and hardware level is reported in a companion paper [3].

We believe that by implementing general purpose algorithms that perform reliable transaction processing in a real distributed database system, we are able to provide a realistic answers to many implementation and performance questions. These evaluations will contribute to alternative choices available to future implementors in industry and academia.

1.1 Experimental Environment

These experimental studies were done either on a complete RAID system or an abstraction of it called Mini-RAID. We briefly describe these systems here, since details have been presented elsewhere [6,5].

RAID is a distributed database system which serves as a test-bed for conducting scientific experiments. RAID provides complete support for transaction processing, including transparency to concurrent access, crash recovery, distribution of data, and atomicity. It is an experimental system running on VAXes and SUNs under the Unix operating system [6]. Database sites in RAID communicate over an Ethernet network. RAID is a modular system and allows for different configurations.

An instance of RAID can manage any number of virtual *sites* distributed among the available physical *hosts*. Each virtual site consists of several servers necessary for transaction processing. Figure 1 depicts the organization of a RAID site. The site is virtual

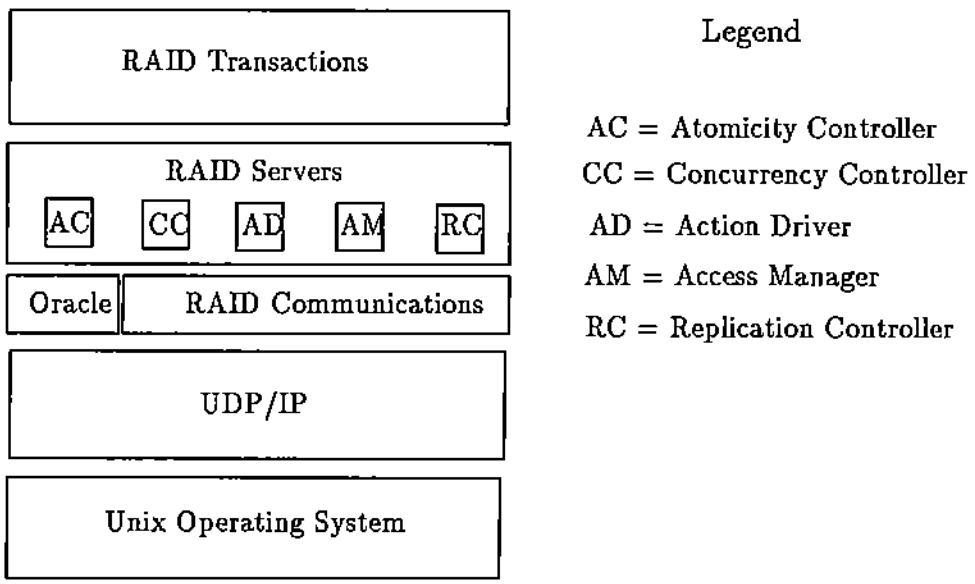

Figure 1: The organization of a RAID site.

since its servers can reside on one or more hosts (machines) and since the site is not tied to any particular host on the network.

Each of the RAID modules is implemented as a separate server, communicating with other modules via datagrams. The servers are currently implemented as single Unix processes. Since the only interaction between modules is through datagrams, individual servers can be changed or completely replaced without affecting other servers. This approach simplifies the incorporation of new algorithms and facilities into RAID, or testing alternate implementations of algorithms.

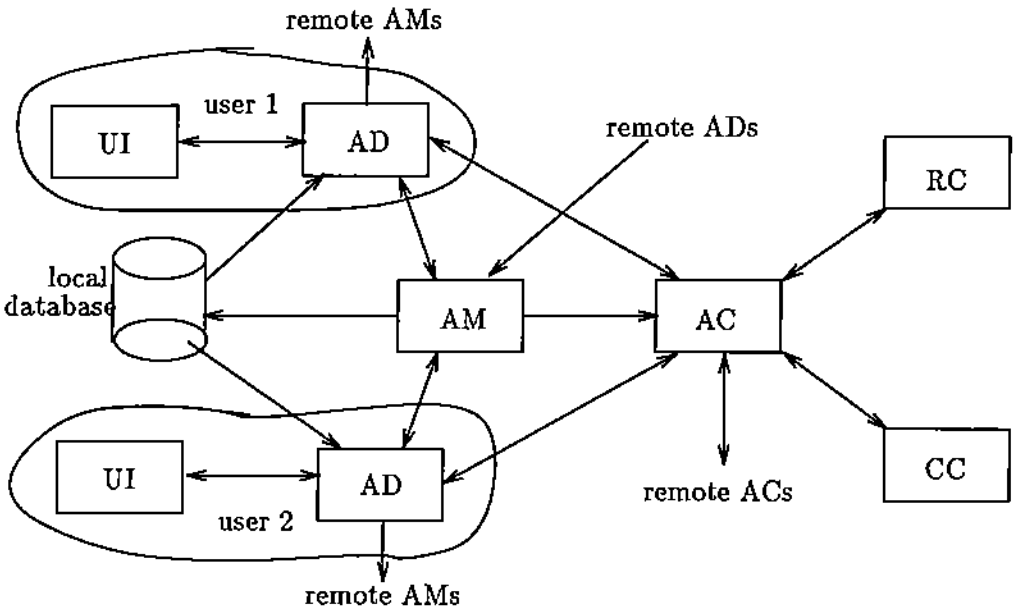

Figure 2: RAID Site Structure.

The following is a brief description of the role of each of the RAID servers (see Figure 2):

User Interface (UI) is a front-end invoked by a user to process relational calculus queries in a QUEL-type language on a relational database.

Action Driver (AD) accepts a parsed query in the form of a tree of actions from its User Interface and executes the transaction, reading data from the local copy of the database. It formats the query as a transaction (read and write actions) for further processing.

Access Manager (AM) provides write access to the local database, and works with AC to ensure that updates are posted atomically to stable storage.

Atomicity Controller (AC) is responsible for ensuring that transactions are uniformly committed or aborted on all sites in the system. This is accomplished by communication between the ACs of all sites, which check with their local Concurrency Controllers (CCs).

Replication Controller (RC) is the server that solves the replicated copy control problem, i.e., it maintains consistency of the replicated copies of the database in the event of multiple site failures.

Concurrency Controller (CC) checks whether a transaction history is locally serializable at a given site.

Database sites in RAID communicate over a 10 million bit per second Ethernet network. RAID servers communicate with each other using high-level operations. These operations are implemented as subroutines that understand and manipulate RAID data structures. The high-level facilities are implemented on top of the low-level RAID transport protocol. We call this protocol *LDG*, for Long DataGram. This protocol is identical to the standard Arpanet UDP/IP protocol except that there is no restriction on packet sizes (many implementations of UDP restrict packet sizes to some maximum length). LDG is currently built on top of UDP. Each LDG packet is fragmented if necessary, and then transmitted using UDP. At the destination, fragments are collected and reassembled. The RAID communication sub-system provides location transparent addressing, and supports multiple virtual sites on a single physical host.

Transaction processing in RAID is separated into one execution phase and two commit phases. In the execution phase the transaction executes on the site to which it was submitted, using only the local copy of the database. The transaction maintains timestamps for its reads, and writes to a copy of the data in volatile memory. During the first commit phase, the executing site communicates with other sites to determine global commitment. The entire read/write set of the transaction is distributed in a single round of messages. Since long messages cost only slightly more than short messages, encapsulating the concurrency control information in a single message has performance advantages over traditional techniques that distribute locking information for each item separately. Phase 2 of commitment is responsible for writing the data to the disk and releasing the commit-locks. It uses one round of messages to communicate the commit decision.

RAID has the following features for experimentatal studies.

1. RAID has built-in support for measurements of elapsed time for each of the phases of distributed transaction processing. The Action Driver checks the value of the time of day clock before and after certain parts of a transaction. Currently measurements can be taken for query interpretation, preparation of concurrency control information, transaction commitment, and posing the updates to the database. The hardware clock resolution is twenty milliseconds, so the measurements are averaged over the execution of several hundred transactions. The clock routines adjust the returned value by one microsecond so that two separate time requests never return the same value, but all that is known about measurements that are in the microsecond range is that they are timing events that took less than twenty milliseconds.

2. The RAID communication package responds automatically to special control messages that instruct it to set the level of debug and timing processing that need to be performed. Another control message causes the communication package to print

the current resource utilization maintained by the kernel for the server, including the number of packets it has sent and received, and its user and system CPU time. Note that once again the clock granularity is than twenty milliseconds. Sometimes servers context switch as often as once per millisecond, so CPU times are only accurate when averaged over a large number of transactions.

3. RAIDTool is a window-based front-end to the RAID system. RAIDTool has a separate window for each site showing the status of the site. A control panel is available to create new sites, cause old sites to fail or recover, and monitor system performance. RAIDTool communicates with the sites via special control messages. Some of the control messages request a simple operation such as changing the timing level, while other messages cause the site to periodically send a summary of certain status information to RAIDTool. In addition, the RAID communication sub-system provides location transparent addressing, and supports multiple virtual sites on a single physical host. The name-server provides a notifier service that automatically informs interested servers of failures or recoveries of other servers. Each server registers a notifier set of servers in which it is interested with the name server.

1.2 Mini-RAID

The Mini-RAID system [5] is an abstraction of RAID. We stripped down the processing of the RAID system in order to run experiments which are relevant to specialized processing such as that needed for replicated copy control. Factored out were the effects of network communications, concurrency control, and data input/output. In the resulting system , which we call Mini-RAID, database sites are implemented as Unix processes (on one processor with one process per site). Due to this, the influence of communication delays on the Ethernet is not considered. Each site keeps a copy of the database, nominal session vector, and fail-locks needed by the protocol [5] to maintain the consistency of these objects.

A managing site provides interactive control of the system's actions. It is used to cause sites to fail and recover and to initiate a database transaction to a site. Site failure was simulated by sending a message to a site to indicate that the site should not participate in any further system actions. A failed site would remain inactive until recovery was initiated from the managing site. The following system parameters are defined through the managing site:

- the database size in terms of the number of data items
- the number of database sites for transaction processing
- the maximum number of operations per transaction, where an operation was defined to be a read or write of a database data item

A database transaction is generated by the managing site and consisted of a random number of operations (from one to the maximum specified for the system).

The Mini-RAID system was initially developed to maintain fully replicated data objects among all of the distributed sites. It has been supplemented with software to manage transaction processing on partially replicated data.

Seciont 4 describes the implementation of partial replication in Mini-RAID, and contains experiments that were conducted on the partially replicated Mini-RAID system.

unique	two	ten	twenty	hundred	thousand
5	0	9	2	64	615
6	1	3	15	38	923
7	0	3	17	68	746
8	1	5	8	80	424
9	0	9	3	59	707
10	0	3	19	32	455
11	1	6	16	20	832
12	1	1	6	79	719
13	1	9	3	19	639
14	1	0	4	41	872
15	1	2	4	84	931

Figure 3: Some example tuples from the thousand relation.

2 Performance Measurements and Analysis of RAID Transaction Processing

This section describes measurements of the performance of the transaction processing protocols in RAID. The servers involved in commitment are the Atomicity Controller (AC) which manages the commit algorithm, and the Concurrency Controller (CC) which determines whether transactions are serializable.

The following series of performance measurements were done on Sun 3/50s (approximately 1 MIPS machines) connected by a 10 megabit/second ethernet. The database for the experiment is 100 tuples from a truncated version of the thousand relation used in [7]. Figure 3 shows a few tuples from this relation. The first column is a unique key for the tuple. The other columns are random numbers selected from the range specified by the column name. For instance, the range for column twenty is 0 to 19. These columns provide for a wide range of selectivity in queries. For example, the query `database thousand, get thousand : thousand.ten = 8;` can be used to select approximately 10% of the tuples.

2.1 Elapsed Time for Transaction Processing

Table 1 shows the time taken by transaction processing for several different database queries on RAID systems with varying numbers of sites. The times do not include the cost of interpreting the database query or the cost of translating the query to a transaction.

Both select queries use a simple predicate that only examines one field in each tuple.

transaction	1 site	2 sites	3 sites	4 sites
select one tuples	0.4	0.4	0.4	0.4
select eleven tuples	0.4	0.5	0.4	0.4
insert twenty tuples	0.7	0.7	0.8	0.8
update one tuple	0.5	0.5	0.4	0.4

Table 1: Execution time for transactions in RAID systems with various numbers of sites (in seconds).

server	CC	
transaction	user	sys
select one tuple	0.04	0.06
select eleven tuples	0.02	0.02
insert twenty tuples	0.12	0.13
update one tuple	0.02	0.02

Table 2: CPU time used by Concurrency Controller in executing transactions (in seconds).

The insert query inserts twenty tuples. The update query updates one field of a selected tuple. The processing time is higher for the insert query because its write set is larger than any of the other read or write sets.

The fact that the processing time is fairly constant as the number of sites increase is due to the use of built-in multicast in the RAID layer of the communications package [3,4]. This lower level multicast only has to format the packet once regardless of the number of sites. Hence, the execution occurs in parallel on each site. Our estimate is that this time will remain constant up to around ten sites if we continue to use UDP as our transport mechanism. We are currently preparing a kernel-level multicast [3] that will help maintain this property for even larger numbers of sites.

2.2 CPU Time for Concurrency Control

Table 2 shows the CPU time used by the Concurrency Controller (CC) for these same queries, averaged over several transactions. Note that the CPU time is a small fraction of the elapsed time for each query. This suggests that multiple queries executing at the same time would be able to overlap significantly. We are considering ways by which the experiment may be extended to multiple simultaneous queries.

	1 site		2 sites		3 sites		4 sites	
transaction	user	sys	user	sys	user	sys	user	sys
select one tuples	0.04	0.14	0.06	0.14	0.04	0.10	0.08	0.24
select eleven tuples	0.04	0.08	0.02	0.04	0.06	0.12	0.06	0.10
insert twenty tuples	0.20	0.16	0.08	0.14	0.10	0.12	0.08	0.10
update one tuple	0.04	0.10	0.06	0.12	0.06	0.16	0.06	0.16

Table 3: CPU time used by RAID Atomicity Controller (AC) in executing transactions on varying numbers of sites (in seconds).

2.3 CPU Time for Atomicity Control

Table 3 shows the CPU time taken by the AC for various numbers of sites. The times show a slight tendency to increase with the number of sites, but the variance in the measurements is too large to permit stronger statements. The variance is probably caused by the granularity of the hardware clock, since individual time slices are likely to be smaller than twenty milliseconds. For instance, sometimes the AC averages almost one context switch per millisecond. Since these numbers are collected directly by the kernel we do not have any means to further control them. In any case, it is again clear that most of the wall clock processing time for an individual transaction is not CPU time.

2.4 Elapsed Time for Communication

Since the transaction processing times include the communication times, we include our measurements of the current RAID communcation software. Table 4 compares the UDP/IP protocol with our own Long DataGram extension (LDG) round-trip communication times

Bytes:	64	512	2048	8192	32768	500000
UDP	7.2	10.6	16.5	48.8	-	-
LDG	13.3	23.7	52.0	170.0	642	10 secs
RAID	14.5	31.2	86.0	300.0	-	-

Table 4: RAID Communication Time by Packet Length (in milliseconds)

for datagrams of various lengths. LDG is about twice the cost of UDP for packets that do not need to be fragmented. In this implementation, LDG datagrams are fragmented into 512 byte packets. Larger datagrams, which UDP transmits as a single packet, are much

more expensive to transmit using LDG. Profiling reveals that about 30% of the additional single-packet cost over UDP of LDG is due to buffer copying, which will be avoided in future releases by replacing the `sendto` and `recvfrom` Unix system calls with `sendmsg` and `recvmsg`, respectively. A further 60% of the cost of a send and 17% of the cost of receive is due to parsing the buffer headers. If communication turns out to be a bottleneck, much of this cost can be avoided by changing to a fixed-format, non-ASCII header. The RAID layer adds a small amount of additional overhead for location transparency. Most of the remaining additional cost of the RAID layer reflected in Table 4 is due to allocating and freeing a buffer. A future implementation will retain a buffer of sufficient size for most messages to avoid this overhead.

2.5 Conclusions from the RAID measurements

A good design should take approximately 230,000 CPU cycles [10] and two rounds of messages for completing a transaction. If round-trip communication delay for a message is approximately 25 milliseconds, the transaction processing time should be around 250 milliseconds for a read or write of a single value [17]. Currently our processing times are around 400 milliseconds for one to seven reads and 500 milliseconds for writes. Our times are slightly higher than CAMELOT times [17], since we deal with an action that reads or writes a tuple in a relation and not just a single value. In addition, we use a separate server for each function on a site, whereas CAMELOT has a single server per site. So our timings include the overhead of communications among the servers. If we communicate among the servers using subroutine calls, our timings will improve by at least 25 milliseconds. We are currently investigating alternative communication strategies such as special-case local message services and hardware multicast that may improve the response time by a factor of two or three [3].

One main conclusion that we draw from this work is that the I/O and communication times dominate the processing time. Using the best algorithms for concurrency control or manipulating data structures in the best possible manner does not show up in the bottom line. We and others are continuing our research in communications [9,4]. Several other projects [17,8] are focussing on reducing the I/O costs on the order of 10-15 milliseconds. These research efforts will have great impact on the performance of database transaction processing.

3 Experiments with Concurrent Checkpointing and Rollback

This section starts with a brief overview of concurrent checkpointing and rollback. It then details an experiment measuring the amount of time it takes a system of communicating processes to checkpoint and recover, including measurements of the delays caused by having multiple processes checkpoint or recover at the same time.

3.1 Brief Overview of Concurrent Checkpointing and Recovery

To allow continuity of transaction processing in the RAID system, we must deal with the failure and restart of individual servers. To recover from failures, a global consistent state must be checkpointed distributively over all servers. In addition, the restoration to a previous global state must be synchronized. We have designed an algorithm [13] that allows concurrent and robust checkpointing and recover in a distributed system. This algorithm utilizes the research of checkpointing [2,12,18], concurrency control, and recovery. In contrast to transaction checkpointing [11] that uses undo/redo logs and concurrency controllers, the problem is mainly concerned with message exchange among the server processes. Message passing establishes a certain kind of execution dependency among multiple processes. We call one instance of the checkpoint algorithm executed on multiple processes a *checkpoint instance*. Similarly, we call one instance of the rollback algorithm executed on multiple processes a *rollback instance*.

The processes that have exchanged messages since their last checkpoints need to take checkpoints or roll back together. Our algorithm synchronizes distributed checkpointing and rollback operations of multiple processes, using the two-phase commit to ensure atomicity. Two-phase commit is performed differently in distributed checkpointing and distributed rollback. In distributed checkpointing, each process, upon a checkpoint request, first takes an uncommitted checkpoint, replies to the coordinator with a "done" message, and then propagates the checkpoint request to some other participants. In distributed rollback, each process, upon a rollback request, first replies to the coordinator with a "done" message, propagates the request to some other participants, and then rolls back. Even if the process fails after replying to the coordinator, we can eventually restore its last saved state from stable storage after the process recovers. Therefore, a rollback operation will always succeed. The rollback commit protocol has a higher degree of parallelism compared to the former one.

Multiple checkpoint instances and rollback instances can interfere with one another. Our algorithm allows concurrent execution of interfering checkpointing and rollback instances. Different instances will not block each other, which gives good response time. Resilience against process failure follows a termination approach. A consistent global state

is maintained among the currently operational processes. When a failed process is up, a new consistent global state is enforced.

3.2 Experiment 1: Synchronization Delays in Concurrent Checkpointing and Rollback

In this section, we evaluate the performance of our distributed checkpointing and rollback algorithm in the experimental setting. In our experiments, several server processes communicate through message queues in Sun Unix. A coordinator initiates a synchronization instance, and sends request messages to some participants. The coordinator and the participant processes take checkpoints or roll back together. The experiment measures the elapsed time, cpu usage, maximum queue length, and maximum queue delay of a process during the execution of the algorithm. Elapsed time denotes the delay represented by time that a process spends during the synchronization of checkpoint operations or rollback operations with other processes. During the period, the process is not allowed to send or receive any normal messages. We also measure the queue length and queue delay to quantify the message traffic. A higher queue length and delay implies high message traffic, and high throughput due to short idle time of processes.

3.2.1 Experimental Design

Here we describe the experimental procedures, message passing, and the delay data for a single checkpoint/rollback used in the experiment. Each experimental scenario performs the following steps:

1. Execute normal processes which send normal messages to one another.
2. Invoke a checkpoint starter or a rollback starter which sends a special message to designated processes. A process that receives this message initiates a checkpoint instance or a rollback instance respectively.
3. Run a special command that stops the normal processes.

Inter-process communication is based on the message queue facility in Sun Unix. Each process is equipped with two queues for incoming messages, one for normal messages, and the other for synchronization messages. Each process sends normal messages randomly (uniform distribution) to all other processes. Upon receiving the first synchronization message, a process joins a synchronization instance, and suspends its normal message passing operations until the end of the synchronization. In the experiments, the size of a synchronization message is 22 bytes.

3.2.2 Timing for Checkpoint and Rollback

Each process in a checkpoint (or rollback) synchronization instance will make a single checkpoint (or rollback respectively). Checkpoint delay is the time to write the image of a process into the disk, while rollback delay is the time to read the image of a process from the disk. We have examined about 900 object files in the UNIX system, some of which are system files, while others are user files. These cover over 90% of all the object files in the UNIX system. An object file is the memory image of a process, and has three segments: *text*, *data*, and *bss*. In taking a checkpoint, we need only write the data and bss segments to the disk, while in rollback, we only read the data and bss segments. The size of these object files (excluding their text segments) in the UNIX system ranges from 4K bytes to 48K bytes. The checkpoint and rollback were measured to take time ranging from 89 ms to 496 ms.

3.2.3 Measured Data

In the experiments we measure the performance of the coordinator and participants separately during the execution of the algorithm. In the performance curves (Fig. 3 to Fig. 6), solid lines display the performance of the coordinator, and dotted lines display the average performance of the participants. A synchronization instance has two to ten processes. We have measured *elapsed time, cpu usage, maximum queue length*, and *maximum queue delay* of a process during the execution of the algorithm. We see little change in cpu usage, maximum queue length, and maximum queue delay when we vary the single checkpoint delay or single rollback delay. Therefore, we take the average performance data over various single checkpoint delays or single rollback delays.

Figure 4 shows elapsed times with respect to three different single checkpoint delays: 89 ms, 251 ms, 496 ms. Figure 4 also shows elapsed times with respect to three different single rollback delays: 89 ms, 251 ms, 496 ms.

Each process has a message queue for incoming synchronization messages. We measure maximum queue length and maximum queue delay during the execution of the algorithm, which are shown in Figures 6 and 7).

We measure cpu usage in the following three cases: a) two checkpoint instances interfere with each other, b) two rollback instances interfere with each other, and c) two checkpoint instances and two rollback instances interfere with each other (Figure 8). Each instance includes the same set of processes. In case c), when a checkpoint instance interferes with a rollback instance, the checkpoint instance is aborted.

3.3 Conclusions and Analysis

Two-phase commit protocols are tailored for efficient synchronization. Our measurement shows the performance of a single instance in terms of synchronization delays and message

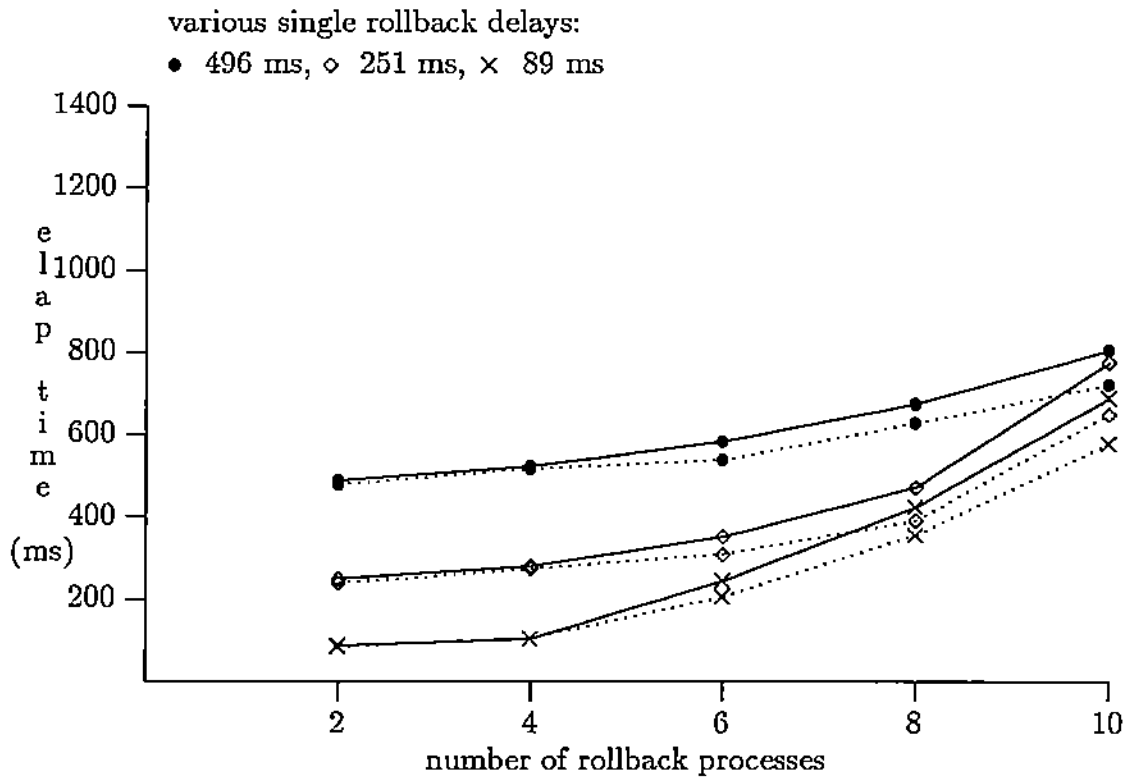

Figure 4: Elapsed Time

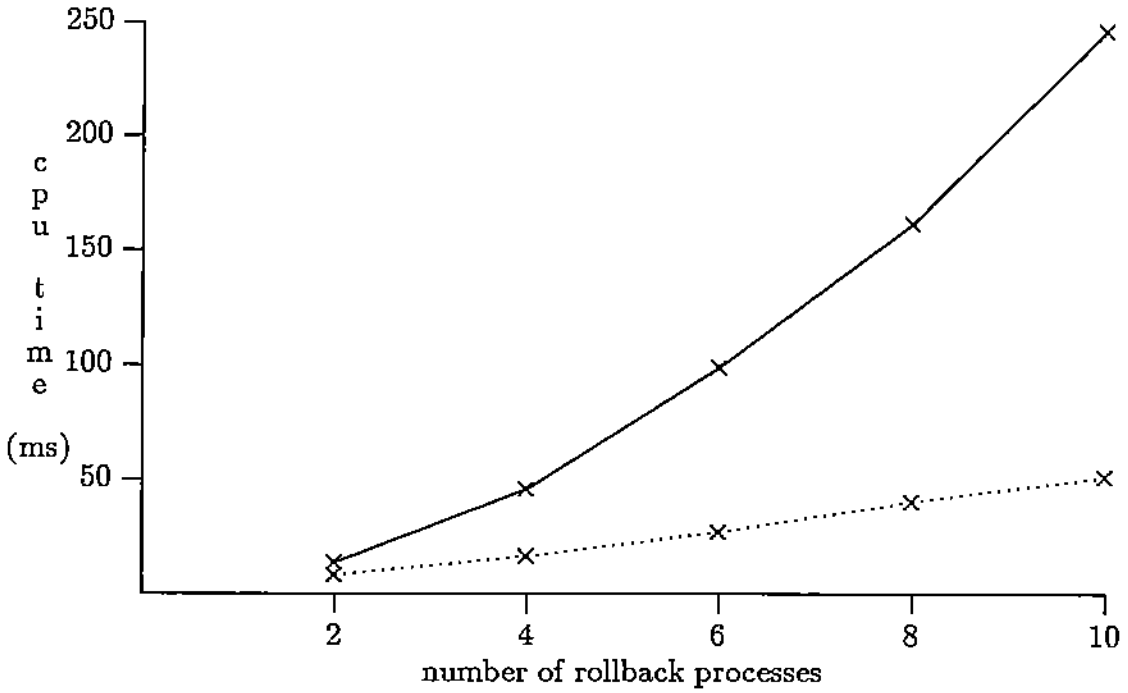

Figure 5: CPU Usage

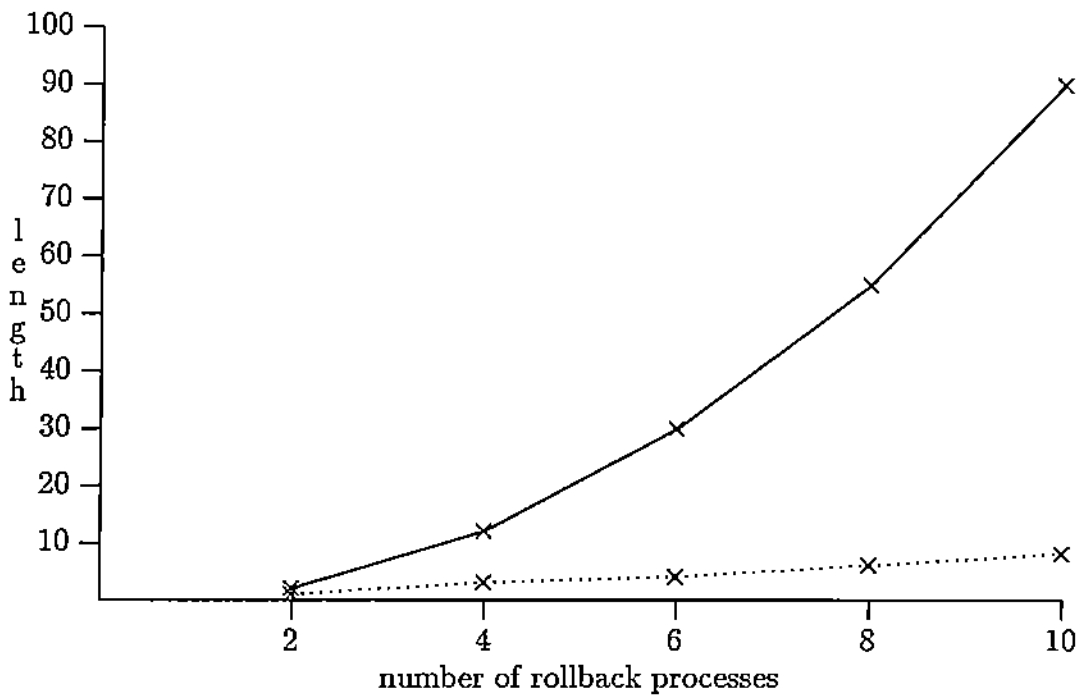

Figure 6: Maximum Queue Length

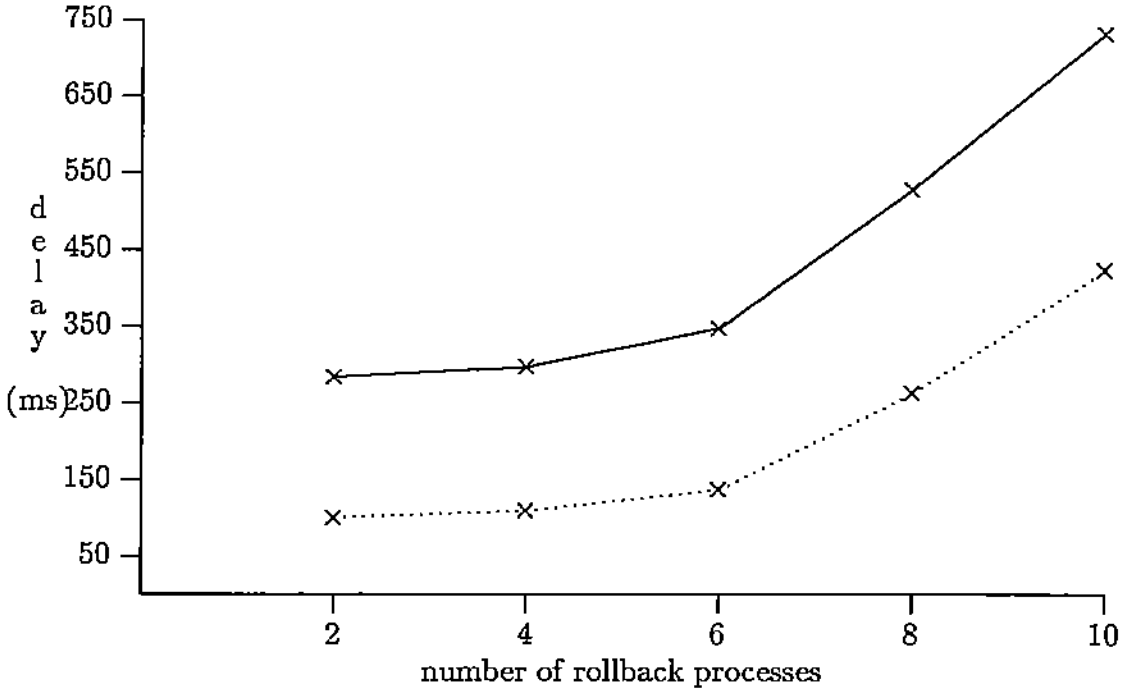

Figure 7: Maximum Queue Delay

overhead. We have also incorporated the concurrency features into the distributed checkpointing and rollback recovery mechanism. Concurrent execution of multiple instances is efficient because of the sharing of checkpoint operations or rollback operations. In the best case, all the operations are shared, and synchronization delay will be the same as in a single instance.

Since two-phase commit performed in a rollback instance has a higher degree of parallelism, elapsed time is smaller than that of a checkpoint instance. The elapsed time of the coordinator of a rollback instance is at least t milliseconds smaller than that of a checkpoint instance, where t is the delay to make a single checkpoint or roll back. Due to the parallelism, elapsed times of the coordinator and participants are very close.

CPU usage in a single participant is insensitive to the total number of participants in a synchronization instance. On the other hand, CPU usage in the coordinator increases faster as the number of participants increases as shown in Figure 5. We note that CPU usage in the coordinator of a rollback instance is 40% higher than that of a checkpoint instance.

Maximum queue length in a participant is linear to the total number of participants in a synchronization instance. But the maximum queue length in the coordinator tends to increase quadratically as the number of participants increases. Maximum queue length in the coordinator of a rollback instance is larger than that of a checkpoint instance. This is because two-phase commit in a distributed rollback instance has a higher degree of

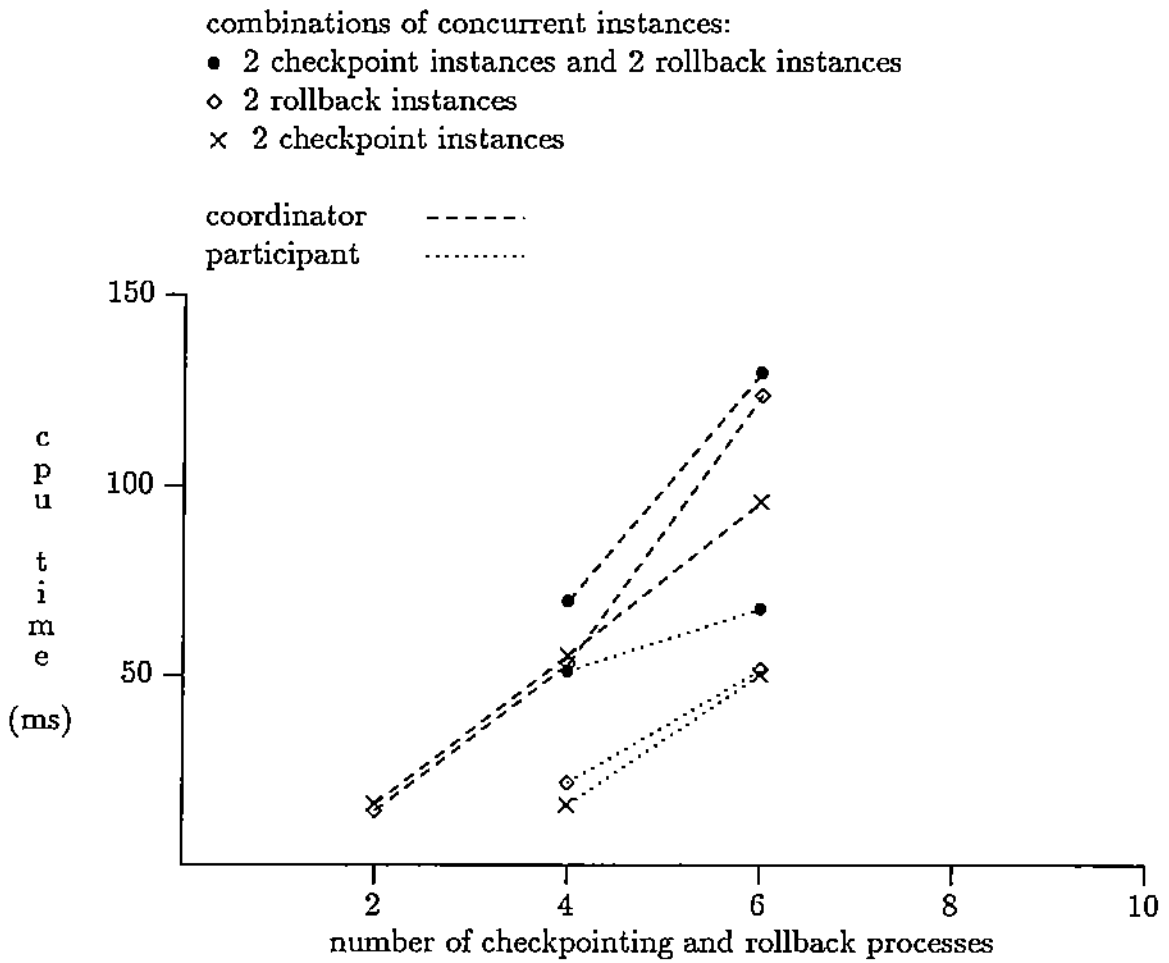

Figure 8: CPU Usage in Concurrent Instances

parallelism. Therefore, inter-arrival time of the messages at the coordinator tends to be small.

Elapsed time of the coordinator gives us a good idea how often a checkpoint instance should be initiated. If the coordinator spends t (elapsed) time to complete a checkpoint instance, then the coordinator should wait for at least t time before initiating a second checkpoint instance. The coordinator has more message overhead in a checkpoint instance or a rollback instance, we can reduce the response time by giving the coordinator a higher priority. In the worst case, the message overhead of the coordinator can increase quadratically as the number of participants increases. The measurements lead to the conclusion that concurrent execution does not reduce the message overhead or cpu usage. However, concurrent execution can reduce the synchronization delay for about 100the same time and have the same participanting processes.

4 Experiments with Partial Replication of Databases

This section describes the implementation of partial replication in the Mini-RAID system and the details of the experiments that were conducted. The first experiment measures the effect of the replica threshold on data availabiliy and the second experiment studies the impact of the degree of replicaion on data availability.

4.1 Implementation of partial replication

A variation of the directory idea presented in [1] was used for the implementation of partial replication. The directories are a fully replicated object on all the sites. Each directory entry for an object contains the following information:

- The sites that currently contain that object (the sites may be operational or in a failed state).
- The number of replicated copies of the object to be maintained within the system as a whole.
- The minimum number of copies to be maintained (i.e. the threshold).

Note that even though data items are partially replicated, fail-locks are still fully replicated on each site. Fully replicating fail-locks entails additional overhead, but partial replication of fail-locks may cause reduced availability and the blocking of processing at recovering sites.

The main changes and additions to the system are described next.

4.1.1 Processing User Transactions

Under partial replication, the read and write operations in a transaction are processed in a slightly different manner. When trying to read an item from a certain site, several cases can arise:

- The item exists on the current site and its value is up-to-date. This value is returned immediately.
- The item exists on the site but a fail-lock is set for it. A copier transaction(s) must be issued to refresh its value.
- The item does not exists on the current site, so a *pseudo* user transaction is created and sent to a site that contains a good copy of that object. The pseudo user transaction is processed by the receiving site in the same fashion as a normal user transaction that is received from the manager.

To process a write operation on an item, at least one copy of the item must exist on an operational site. Note that this is not necessary because new copies can be created using the value in the write operation. If a transaction is committable, only the sites that contain the objects in the write set will have to update their database values. However, since fail-locks are fully replicated, all operational sites must receive the transaction's write set so that they can update their fail-lock tables accordingly.

4.1.2 Maintaining the Threshold

As mentioned earlier, the user is allowed to specify the minimum number of copies to be maintained for objects throughout the distributed system. This value is refered to as the *threshold*. The threshold is stored in the directories and is used later to create new copies of certain objects. When a site detects a failure in Mini-RAID, it informs other operational sites of the failure. It also checks if the threshold is still maintained for each object. If a new object needs to be created, the creation is attempted as follows:

- If there exists a fail-locked copy of the object on an operational site and at least one up-to-date copy of the object on another site, the fail-lock is cleared by updating the fail-lock table.
- If an operational site does not contain a copy of the object, the object is added to the sites' database by updating the directories.
- If neither of the above is possible, then no more copies can be created.

When the creation of a new object is possible, a 2-phase commit protocol is used to send the new fail-locks and directories to other sites. Upon receiving the fail-locks and directories, each site determines whether it should initiate copier transactions for any items. This is done by comparing the old and new copies of the fail-locks and directories. To determine which site to send the copier transaction to, the old copy of the directories and fail-locks is used. It is possible that a site trying to send a copier transaction might discover that all the sites containing an up-to-date copy of the object have just failed. In this case, the site should set a fail-lock for itself and inform other sites about this change. This case was not implemented in the extension being described.

4.2 Experiment 1: Effect of the Threshold on Data Availability

This experiment measured how setting the threshold level in a partially replicated system can affect data availability. The threshold is specified at system configuration time. It refers to the minimum number of copies of each object to be maintained in the system. In our system, transactions can be aborted in one of two situations: when a failure is detected during the 2-phase commit protocol and when a referenced item is not available on any

operational site. Availability is measured in terms of the number of transactions aborted because of site failure and the unavailability of data.

4.2.1 Design of Experiment

The extended Mini-RAID system was used for this experiment. Systems with different threshold levels were used. In each system, half the sites were failed and sets of transactions were processed as more failures and recoveries occurred.

4.2.2 Measured Data

A 12 site system with a degree of replication of 3 was started up. The maximum transaction size used was set to 5, and the number of frequently referenced items in the database was set to 100. The experiment was carried out for three different threshold levels of 1, 2, and 3. Note that a threshold of 1 is the same as 0, because when the number of available copies drops below 1 (i.e. becomes 0), it is not possible to make new copies. The processing scenario for each measurement is as follows:

- Initially, all objects are randomly distributed among the sites.
- 6 random sites are failed before processing the first transaction.
- 20 transactions are processed while the 6 sites are down.
- One of the failed sites is recovered and another operational site is failed.
- The above two steps are repeated 4 more times.
- 3 more sites are failed immediately.
- 20 transactions are processed while all nine sites are down.
- The procedure of recovering one site, failing another, and then running 20 transactions is also repeated 5 times.

For each set of 20 transactions, the number of transactions aborted was recorded. Figure 9 illustrates the results of the three measurements. Overlapping lines have been plotted very close together so that they can be distinguished. The asterisk in the graph corresponds to the number of abortions recorded just after a set of transactions was run with 9 failed sites.

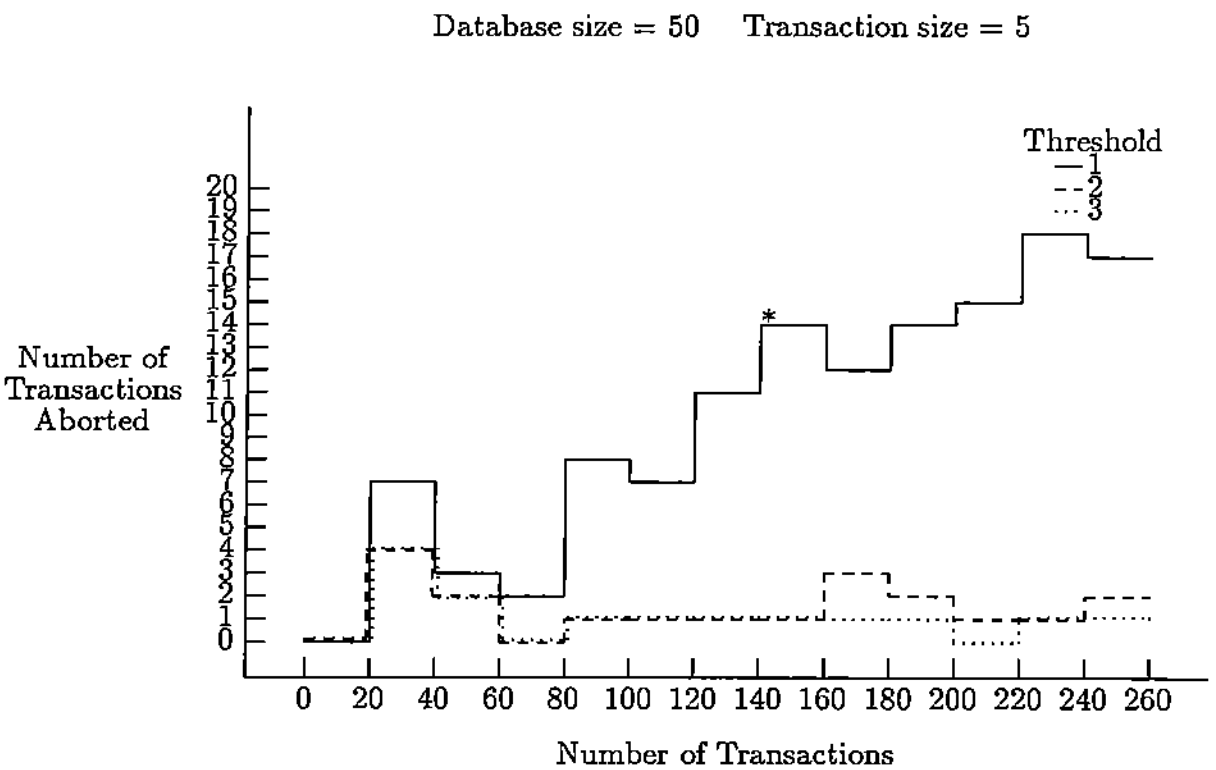

Figure 9: Effect of the threshold on transactions aborted due to data non-availability

4.2.3 Conclusions and Analysis of Experiment 1

This experiment shows that increasing the threshold level in a partially replicated system can improve data availability. The improvement is most significant when the threshold is increased from 1 to 2. As the threshold level is increased further towards the degree of replication, the improvement continues in a decreasing fashion. This is obvious from the following measurements:

- In a system with a threshold of 1, 6 site failures (out of 12 sites), caused an average of 35% of the transaction to be aborted. When 9 sites are failed, the committed transactions cause more fail-locks to be set, thus increasing the average abortion rate to 75%.
- Figure 9 shows that with 9 failed sites and a threshold of 2, only 10% of the transactions get aborted. When the threshold level is increased to 3, the number of aborted transactions drops to 5%.

The threshold of 1 causes a high abortion rate because the new copies of unavailable objects are never generated. With a threshold of 2 or 3, there are always 0, 2, or 3 copies of each object available. Therefore, a single failure does not affect future availability to a large extent. In fact, a large part of the abortions recorded for the thresholds of 2 and 3 are due to failure detection and not the unavailability of data. A degree of replication of N and a threshold of T ($T \leq N$) are enough to maintain consistency as long as there are at most (T - 1) simultaneous failures and at least N operational sites in the system.

In our extensions of the system, the threshold was maintained by checking the number of available copies only when a failure is detected during the processing of a user transaction. We suggest some improvements that can be added to the system in order to enhance its performance.

- Make the recovering site check for possible object creation. This idea will be useful when all sites containing a certain object fail simultaneously, but it will require concurrency control since several sites can recover at once.
- If the site creating the new directories fails, other sites should attempt to continue that task. The sites can compete for access to the directories or the site with the highest ranking can be assigned the job.

4.3 Experiment 2: The Impact of Replication on Data Availability

This experiment measured the behavior of fail-lock setting and clearing for different degrees of replication. Note that fail-locks are set by an update on an operational site if another

site is down. The failed site on recovery gets the fail-locks to identify out-of-date objects. The fail-locks are cleared as fail-locked items are updated by transactions. In a partially replicated database, there is no attempt to read a copy from or send an update to a site that does not have a copy of the object(s) referenced in the transaction. However, when a transaction commits, the write set must be sent to all the operational sites so that they can update their copy of the fail-locks if necessary. Since fail-locks are only set for failed sites that contain an old copy of a certain object, the maximum number of fail-locks that a certain site can have set for it is equal to the number of objects that this site contains.

4.3.1 Design of Experiment

The extended Mini-RAID system described in Section 4.1 is used for this experiment. A 15 site system with a degree of replication of 2 was started up. A random site was failed and transactions were processed until most objects of that site were fail-locked on operational sites. The failed site was then brought up and another set of transactions was processed. The number of fail-locks set after each transaction was recorded. This procedure was repeated for the 15 site system with degrees of replication of 5, 10, and 15. Other parameters used in the system were:

- Maximum transaction size = 5 items.
- Number of frequently referenced items in the database = 100 items.

For the different degrees of replication, the following scenario is used to study the behavior of the recovery process:

- Initially all objects are randomly distributed among the sites according to the degree of replication specified.
- A random site, S, is failed before processing the first transaction.
- 100 transactions are processed while site S is down.
- Site S is brought up.
- 125 transactions are processed while all the sites are operational.

With a database size of 100 objects, it was observed that about 71% of the fail-locks were set for the failed site during the processing of the first 100 transactions. This was true for the different degrees of replication. Also, the number of fail-locks cleared was relatively higher during the first period of the recovery. The results are illustrated in Figure 10. The peak of each graph corresponds to the number fail-locks set at the 100th transaction.

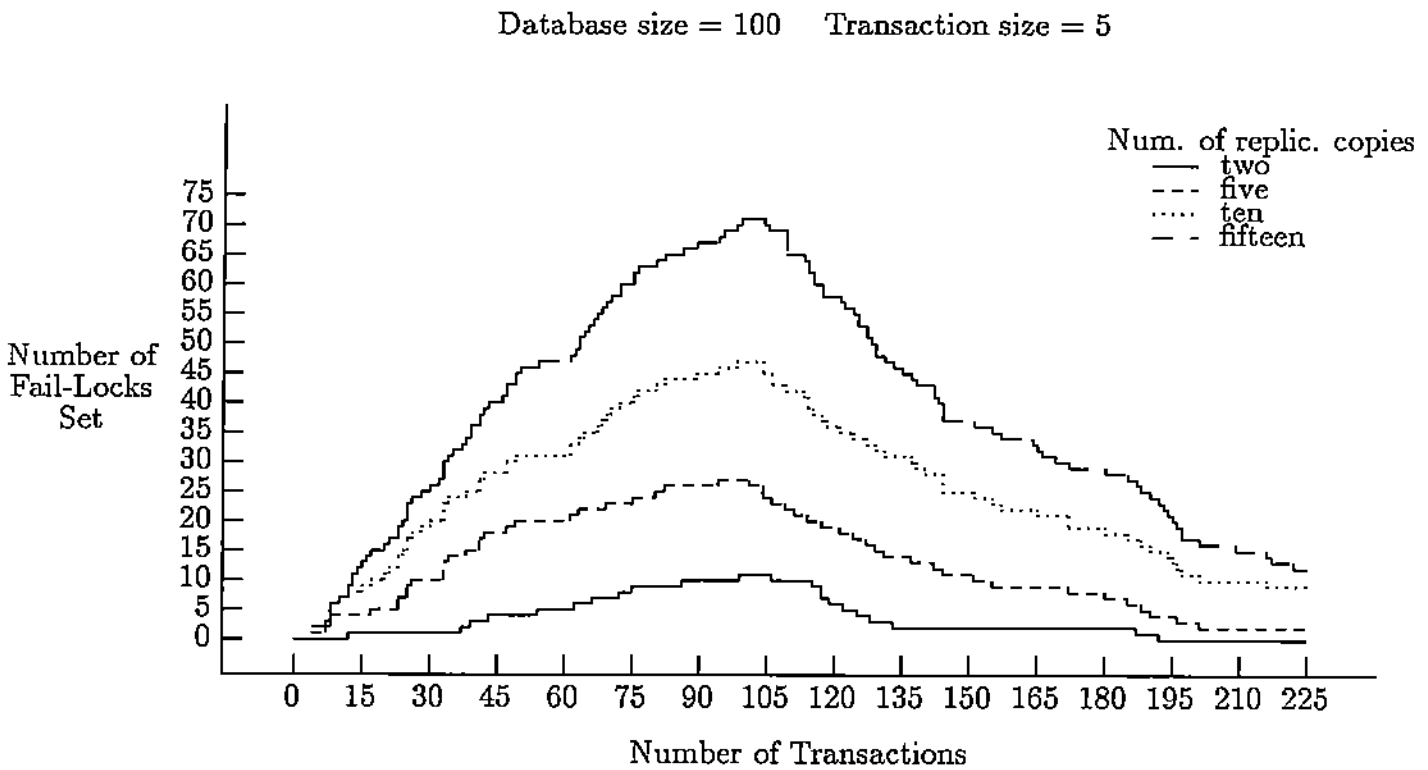

Figure 10: The Impact of Replication on Data Availability

4.3.2 Conclusions and Analysis of Experiment 2

This experiment shows that the degree of replication does not really affect the percentage of data that becomes unavailable on the failed site. With different degrees of replication, about 71% of the objects existing on the failed site become out-of-date after 100 transactions are processed. The 71% corresponds to 71 of the 100 objects in the fully replicated system and 11 of the 15 objects in the system with a degree of replication of 2. Section 4.2 showed that maintaining a high threshold level reduces transaction abortion rate to 5%. Therefore, an N site system with a degree of replication of K and a threshold of T=K, provides data availability that is similar to that of an N site fully replicated system. Figure 10 shows that a lower degree of replication causes less fail-locks to be set which in turn causes less copier transactions to be issued. So from this point of view, transactions can be processed faster.

When deciding on the degree of replication for a distributed system, two of the main concerns are transaction processing speed and data availability. When comparing partially and fully replicated databases, it is true that partial replication requires remote access of non-local data, but this cost is low compared to the costs saved in the write operation. Namely, these costs are due to the reduction in message sizes, disk accesses, and the number of copier transactions. The other fear with having a low degree of replication is the non-availability of data. It was shown in experiment 4.2 that maintaining the threshold automatically increases the availability significantly. In a 12 site system with a degree of replication of 3 and a threshold of 3, 9 site failures caused only 5% of the transactions to be aborted. The top graph in Figure 10 shows that many fail-locks get set in a fully replicated system. If the number of failures is increased, fail-locks will be set on more sites causing substantial delays. Therefore, when designing a distributed database system, partial replication with automatic copy generation should be seriously considered.

5 Conclusions and Future Work

Our experimental studies based on real transaction processing in an operational systems environment provide several insights. We find that the contribution due to concurrency control processing is a small part in the transaction processing response time. Using the standard communication software, the commit protocols that need round trip message exchanges will actually dominate the response time. However, we conclude that response times of the order of 300-500 milliseconds are achievable in a real system such as Raid. Further optimizations that are underway can reduce these times by 50 to 100 milliseconds [3].

Our studies have shown that the overheads due to synchronization in concurrent checkpointing and rollbacks are of the order of a single checkpoint on the stable storage. The smart overlapping of checkpoint instances can terminate such processes with a small elapsed time. The checkpointing/rollback facilities are necessary evils and are critical in the design of truly non-stop systems. We believe that we have shown that such algorithms can be implemented, tested, and we have conducted one of the rare experimental studies on this subject.

Finally we have conducted experiments and provided inferences that are essential to the establishment of the degree of replication and threshold levels in a partially replicated database systems. These results together with our earlier studies [3,5,4] provide the much needed tools for measurements and experimental data for the designers and implementors of distributed systems.

References

[1] R. Attar, P. A. Bernstein, and N. Goodman. Site initialization, recovery, and backup in a distributed database system. *IEEE Trans. Softw. Eng.*, SE-10(6), Nov. 1984.

[2] G. Barigazzi and L. Strigini. Application-transparent setting of recovery points. In *Proceedings of the 13th IEEE Symposium on Fault-Tolerant Computing*, Milano, Italy, June 1983.

[3] B. Bhargava, E. Mafla, J. Riedl, and B. Sauder. *Implementation and Measurements of an Efficient Communication Facility for Distributed Database Systems.* Technical Report CSD-TR-783, Purdue University Computer Science Department, June 1988. Submitted for publication to *IEEE Data Engineering Conference.*

[4] B. Bhargava, T. Mueller, and J. Riedl. Experimental analysis of layered Ethernet software. In *Proceedings of the ACM-IEEE Computer Society 1987 Fall Joint Computer Conference*, pages 559–568, Dallas, Texas, Oct. 1987.

[5] B. Bhargava, P. Noll, and D. Sabo. An experimental analysis of replicated copy control during site failure and recovery. In *Proceedings of the 1988 Data Engineering Conference*, pages 82–91, Los Angeles, CA, Feb. 1988.

[6] B. Bhargava and J. Riedl. Implementation of RAID. In *Proceedings of the Seventh Symposium on Reliability in Distributed Systems*, Columbus, Ohio, Oct. 1988. To appear.

[7] D. Bitton, D. DeWitt, and C. Turbyfil. Benchmarking database systems: a systematic approach. In *Proceedings of the VLDB Conference*, Oct. 1983.

[8] A. J. Borr and F. Putzolu. High performance sql through low-level system integration. In *Proceedings of the SIGMOD International Conference on the Management of Data*, pages 342–349, Chicago, IL, June 1988.

[9] D. R. Cheriton. VMTP: A transport protocol for the next generation of communication systems. In *SIGCOMM '86 Symposium*, pages 406–415, ACM, August 1986.

[10] J. Gray. Private communication, June 1988.

[11] J. Gray. The transaction concept: Virtues and limitations. In *Proceedings of the VLDB Conference*, Cannes, France, Sep. 1981.

[12] R. Koo and S. Toueg. Checkpointing and rollback-recovery for distributed systems. *IEEE Trans. Softw. Eng.*, SE-13(1):23–31, Jan. 1987.

[13] P. Leu and B. Bhargava. Concurrent robust checkpointing and recovery in distributed systems. In *Proceedings of the IEEE Data Engineering Conference*, pages 154–163, Los Angeles, CA, Feb. 1988.

[14] B. Liskov. Distributed programming in ARGUS. *Commun. ACM*, 31(3):300–312, March 1988.

[15] A. Z. Spector, J. J. Bloch, D. S. Daniels, R. P. Draves, D. Duchamp, J. L. Eppinger, S. G. Menees, and D. S. Thompson. The Camelot project. *Database Engineering*, 9(4), Dec. 1986.

[16] *Guide to the CAMELOT Distributed Transaction Facility.* Carnegie Mellon Computer Science Department, 0.98(51) edition, May 1988.

[17] A. Z. Spector, D. Thompson, R. F. Pausch, J. L. Eppinger, D. Duchamp, R. Draves, D. S. Daniels, and J. J. Block. *CAMELOT: A Distributed Transaction Facility for MACH and the Internet - An Interim Report.* Technical Report CMU-CS-87-129, Department of Computer Sciences, Carnegie Mellon University, June 1987.

[18] Y. Tamir and C. H. Séquin. Error recovery in multicomputers using global checkpoints. In *Proceedings of the 13th IEEE International Conference on Parallel Processing*, St. Charles, IL, Aug. 1984.