PERFORMANCE OF A MIRRORED DISK IN A REAL-TIME TRANSACTION SYSTEM

Shenze Chen, Don Towsley
Department of Computer and Information Science
University of Massachusetts
Amherst, MA 01003

COINS Technical Report 91-07

Performance of a Mirrored Disk in a Real-Time Transaction System *

Shenze Chen   Don Towsley

Department of Computer & Information Science
University of Massachusetts
Amherst, MA 01003

Abstract

Disk mirroring has found widespread use in computer systems as a method for providing fault tolerance. In addition to increasing reliability, a mirrored disk can also reduce I/O response time by supporting the execution of parallel I/O requests. The improvement in I/O efficiency is extremely important in a real-time system, where each computational entity carries a deadline. In this paper, we present two classes of real-time disk scheduling policies, RT-DMQ and RT-CMQ, for a mirrored disk I/O subsystem and examine their performance in an integrated real-time transaction system. The real-time transaction system model is validated on a real-time database testbed, called RT-CARAT. The performance results show that a mirrored disk I/O subsystem can decrease the fraction of transactions that miss their deadlines over a single disk system by 68%. Our results also reveal the importance of real-time scheduling policies, which can lead up to a 17% performance improvement over non-real-time policies in terms of minimizing the transaction loss ratio.

1 Introduction

In many computer systems, reliability is highly required and of critical importance. This requirement can be found, among others, in the area of database applications, where the database is usually stored on one or more disks. Because of the mechanical nature of disk drives, they are one of the weakest components in a computer system. In order to gain fault tolerance, these systems are typically designed to provide redundancy so as to survive single component failures. This forms the basis behind the concept of *a mirrored disk subsystem*, whereby each data item is stored on a pair of disks [3,12,9]. It is indicated in

*This work is supported, in part, by the Office of Naval Research under contract N00014-87-K-796, by NSF under contract IRI-8908693, and by an NSF equipment grant CERDCR 8500332.

the literature [4] that by today's technology, the mean time between failure of a mirrored disk subsystem will be more than 30,000 years. Besides providing fault tolerance, disk mirroring is also expected to improve the system performance.

A real-time transaction system, where each transaction carries a deadline when submitted to the system, often requires highly reliable service. Here by deadline, we mean the time by which a transaction is expected to terminate. In such a real-time system, if a transaction cannot commit before its deadline expires, it is said to be lost. For a real-time transaction system, the performance metric of most interest is the transaction loss ratio, and the primary design goal is to minimize the loss ratio. This is quite different from that of traditional database systems, where the goal is that of reducing the mean transaction response time. Since the timing constraint is a main concern in a real-time system and I/O devices are orders of magnitude slower than CPU's, the improvement in I/O efficiency, that may arise from the introduction of disk mirroring, is extremely important.

Previous work on the performance evaluation of a mirrored disk system and/or replicated database has mainly concentrated on non-real-time systems. Towsley et.al. [19] discussed several scheduling policies for a non-real-time mirrored disk system, where I/O requests are served in FCFS order. Matloff [13] developed an approximate analysis of a different form of disk redundancy, where three or more disks are used to maintain two copies of each data item. Bitton and Gray [4] examined a similar policy in the context of k disks. Nelson and Iyer [14] described and analyzed two protocols for a replicated non-real-time database system. Other work on replicated database systems can be found in [7,8,2]. For real-time systems, Chen et.al. [6] introduced two new real-time disk scheduling policies. Abbott [1] and Carey et.al. [5] each proposed SCAN based real-time disk scheduling policies. These three studies considered a single disk system. Son [17] presented an algorithm for maintaining consistency with replicated data in distributed real-time systems; this work only considered read-only transactions. To the authors' knowledge, no discussion on the performance evaluation of mirrored disk for real-time systems has appeared in the literature.

In this paper, we present two policies suitable for disk mirrored real-time systems, and examine their performance in a real-time transaction system. Each policy maintains two queues for storing I/O requests. The first policy, labeled RT-DMQ, maintains a queue at each disk. Write requests are placed in both queues and read requests are placed into one of the queues. A variation of the SSEDO policy, first proposed in [6], is used to schedule requests at each queue. The second policy, RT-CMQ, maintains a single queue for *both disks*. All read and write requests are entered into this queue upon arrival. A second queue is required for the disk that lags behind while servicing write requests. Again, a variation of SSEDO is used at the common queue.

The basic real-time transaction system simulation model we use has been validated on an actual real-time database system testbed, called RT-CARAT. Our experimental results using this model show that the real-time policies , RT-DMQ and RT-CMQ, can lead to a 17% performance improvement over non-real-time policies (termed DMQ and CMQ) with respect to the transaction loss ratio. Also, by providing an additional disk, a disk mirrored system can outperform a single disk system by as much as 68%. Other performance metrics of interest include the mean response time for successfully committed transactions, and the mean I/O queue length.

Finally, based on our results, we consider a non-real-time setting and observe that a modified DMQ policy that uses the shortest seek time first (SSTF) at each queue outperforms a modified CMQ policy that also uses SSTF. This contrasts with the results in our earlier study [19], where CMQ outperforms DMQ. Here the original CMQ and DMQ policies used FCFS.

The remainder of this paper is organized as follows. Section 2 describes the real-time transaction system model. Section 3 introduces the two real-time scheduling policies, RT-DMQ and RT-CMQ, for disk mirrored I/O subsystems. The performance results are presented in Section 4. Section 5 summarizes this paper.

2 The Real-Time Transaction System Model

The real-time transaction system is modeled as a closed system (Figure 1), which consists of multiple users, a CPU, and a mirrored disk I/O subsystem. While our intention is to specifically study real-time disk I/O scheduling policies for a mirrored disk subsystem, we do so in a complete system setting. The overall goal is to minimize the transaction loss probability.

In this system model, each user spends a random amount of time in a *think* state before generating a transaction. Each transaction is considered to have the same importance and is assigned a deadline when submitted to the system. If a transaction cannot commit before missing its deadline, it is lost and immediately removed from the system. The user who submitted the lost transaction returns to the *think* state. Each user is assumed to have a private buffer which is large enough to accommodate all pages necessary for a transaction.

A transaction consists of a random number of operational steps, which are executed sequentially. When a transaction is submitted, it waits for the CPU in order to initialize execution of its operational steps. During each step, the transaction needs to access the database once. In our database model, in order to maintain database consistency, serializ-

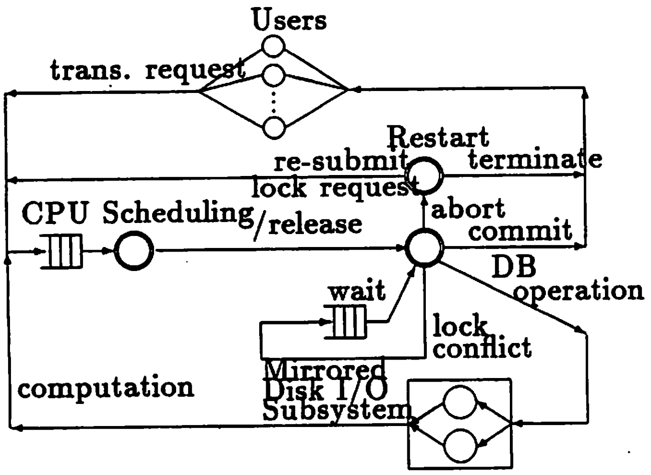

Figure 1: The Real-Time Transaction System Model

ability is enforced by a *two phase locking* protocol. Specifically, each page is associated with a lock. If during a step, a transaction wants to access a database page, it must request and hold the lock for that page before being allowed to access the disk. For any transaction, all locks obtained at previous steps are held until it commits or aborts. After obtaining a lock for a step, a transaction queues up in the I/O subsystem waiting to be scheduled if the page is not in memory. While receiving service, it brings the desired page to its memory buffer, after which it competes for the CPU in order to manipulate the retrieved data. This process is repeated until all steps have finished, after which it is prepared for commit. If there are any dirty pages in its buffer, they are written back to the database. A transaction commits only when all of these writes complete.

It is possible that a transaction's deadline will expire before it commits. Two cases are identified. If a transaction is lost before it completes all of its steps, it is simply removed from the system and the user is notified. Otherwise, if a transaction misses its deadline while performing the write operations, it remains in the system until all writes complete. In this case, a notice is sent to the user, indicating his transaction missed its deadline, but that the database has been updated.

For the lock mechanism in our model, we distinguish between two types of locks. A *shared lock* can be granted to multiple users for read operations, and an *exclusive lock* is granted to only one user for an update operation. A lock conflict may occur when a requested lock has been granted to other transactions, and either the requesting or the holding transaction is to perform an update operation. In this case, we always suspend the requesting transaction, i.e., it joins a queue waiting for the lock. There is no preemption, since all transactions are assumed to be of the same importance. Although there may be better concurrency control policies [10,11], we are only interested in the effects that I/O scheduling have on performance, hence this policy is adequate for our purpose.

The locking scheme used may cause deadlocks. A simple cycle detection deadlock detection algorithm is employed in the model. When a deadlock is detected, the requesting transaction is aborted. In this case, it releases all resources held and proceeds according to a restart strategy. The restart strategy in our model is simple. A transaction aborted from a deadlock is restarted as long as its deadline has not yet expired.

When a transaction aborts or commits, it releases all locks held. For each of these released locks, if there are any transactions waiting for it, we select the one with earliest deadline to wakeup, and grant it the associated lock.

CPU scheduling is another important issue when dealing with real-time transaction processing [10]. The earliest deadline scheduling policy is used in our model.

Last, a real database system will include logging operations to a separate I/O device. We account for it as part of the CPU cost in our model.

Our basic transaction system model, in which the I/O subsystem is replaced by a single disk that services I/O request in a FCFS fashion, has been validated on a real-time database testbed, called RT-CARAT [10], running on a VAX Station II/GPX [1]. Since RT-CARAT uses a fixed number of steps for each transaction, we selected workloads by setting the number of users to 8 and varying the number of steps for each transaction from 4 to 20. In particular, we ran each of these workloads on RT-CARAT and gathered statistics on the computation time required for a transaction step (including operating system overhead). We then ran the same workloads with these step computation times and algorithm settings (including: CPU scheduling, lock conflict resolution, deadlock resolution, and restart strategy) in our simulation. The results from the simulation match quite well with the measurements taken from the testbed.

3 The Mirrored Disk I/O Subsystem

It is our main interest to examine the impact of mirrored disk I/O subsystem on the overall performance of the real-time transaction system. In our model, each I/O request arriving to the I/O subsystem carries a deadline which is inherited from the transaction issuing the request.

In our previous work [19], we studied the performance of a non-real-time mirrored disk system under different FCFS based scheduling policies. It was observed that two multiple queue policies, DMQ (*for Distributed Multiple Queue*) and CMQ (*Centralized Multiple*

[1]Since the VAX Station II, where the RT-CARAT is running, does not support mirrored disk, we have no way to validate the various policies discussed in this paper on RT-CARAT

Queue), outperform various single queue policies in terms of minimizing the mean read and update response time, and that CMQ slightly outperforms DMQ. On the other hand, in [6] we proposed two real-time disk I/O scheduling policies, SSEDO (*Shortest Seek and Earliest Deadline by Order*) and SSEDV (*Shortest Seek and Earliest Deadline by Value*), for a single disk system. They exhibit better performance over all other suggested real-time disk scheduling policies. In the following, as a supplement to our previous studies on non-real-time mirrored disk systems, we first describe two modified versions of DMQ and CMQ policies. Then we merge the features from the non-real-time mirrored disk policies and the real-time single disk policies to provide two real-time scheduling policies for a mirrored disk I/O subsystem. Finally, we describe another proposed single disk real-time scheduling policy, FD-SCAN [1], and apply it to our mirrored disk context. The performance of all these policies are compared in next section.

3.1 Non-Real-Time Policies DMQ and CMQ

These two policies were first introduced in [19]. In this section, we describe a modified version of these two policies.

- **The Modified DMQ Policy:**
  The DMQ policy maintains two separate queues, one for each disk. Every update request generates two write requests that enter the two queues. Consider the arrival of a read request. If both disks are idle then it goes to the disk where the current arm position is closest to the track required by the read. If only one disk is idle, the read request always goes to that disk. Otherwise, it joins a queue waiting to be served. We consider several possible strategies for determining which queue it should join. The first strategy is to let reads randomly choose one queue to enter (DMQ-RD). In the second alternative, when a read arrives to the I/O subsystem, it selects the shortest queue to enter (DMQ-SQ). In the third strategy, the read request joins both queues. Whenever the first read begins service, its peer is removed from the other queue. We refer to this alternative as the *minimum read* strategy (DMQ-MR). Unlike the original DMQ policy described in [19], where the I/O requests are served in a FCFS order, the modified version applies the *shortest seek time first* (SSTF) policy to each queue to select the next request for service [2]. Since the disk service time under SSTF is less than that of FCFS [18], the disk I/O response time, and therefore the overall system performance, is expected to improve.

[2]Other alternate policies applicable include SCAN and C-SCAN.

Among the three alternatives, since the performance of the *shortest queue* strategy is close to that of the *minimum read* strategy and the former is easier to implement, we use the DMQ-SQ in most our experiments in order to compare it with other policies. In the following we simply use DMQ to refer to the *shortest queue* strategy. The other two alternatives will be indicated explicitly when we describe their performance.

- **The Modified CMQ Policy:**
  The CMQ policy maintains a common queue for all arriving requests. When any disk becomes free, the scheduler selects one request from the queue for service. If the request is an update, it spawns two write requests, one of which begins service at the idle disk and the other which either initiates service on the second disk if it is idle, or queues up at that disk if it is busy. Hence an auxiliary queue of write requests may dynamically develop at the disk that lags behind under this policy. Similarly, in the modified CMQ policy, SSTF is used on both the common queue and the auxiliary queue for scheduling rather than FCFS.

Corresponding to the two policies, we present two real-time disk scheduling policies, RT-DMQ and RT-CMQ, for a mirrored disk system, which take into account not only the disk seek time but also the timing constraint of each I/O request when making decisions.

3.2 The RT-DMQ Policy

The RT-DMQ policy maintains separate queues for the two disks in the same way as the DMQ policy. These policies differ from each other in the scheduling rule used at each queue. Under RT-DMQ, requests are scheduled from each queue according to the SSEDO policy, first introduced in [6]. The SSEDO policy sorts the requests in a queue according to their deadlines. Consider one queue and let

r_i : be the I/O request with the i-th smallest deadline at a scheduling instance;

d_i : be the distance between the current arm position and request r_i's position.

A window of size m is defined as the first m requests in the queue, i.e., the m requests with smallest deadlines, $r_1, r_2, ... r_m$. The scheduling rule assigns each request in the window a weight, say w_i for r_i, (where $w_1 = 1 \leq w_2 \leq ... \leq w_m$), and chooses the one with the minimum value of $w_i d_i$. This quantity $w_i d_i$ is referred to as the *priority value* associated with request r_i. If there is more than one request with the same priority value, the one with the earliest deadline is selected. Although there are many ways to assign these weights w_i, in our experiments, they are simply set to

$$w_i = \beta^{i-1} \quad (\beta \geq 1) \qquad i = 1, 2, ..., m.$$

where β is an adjustable scheduling parameter. Observe that when $\beta = 1$, the policy schedules requests within the window according to the SSTF rule. On the other hand, as $\beta \rightarrow \infty$, the policy converges to the *earliest deadline* policy. In this way, a request with a loose deadline is allowed a higher priority only when it is "very" close to the current arm position.

Corresponding to the three non-real-time DMQ strategies, we can define RT-DMQ-RD, RT-DMQ-SQ, and RT-DMQ-MR policies in the same way. Since most of the results in the next section are for the RT-DMQ-SQ, we refer to it as RT-DMQ in the remainder of the paper except indicated explicitly.

3.3 The RT-CMQ Policy

The RT-CMQ policy, like the CMQ policy, maintains a common queue for all arriving requests. This common queue is also sorted by the requests' deadlines. Whenever a disk becomes free, the SSEDO policy described in the last subsection is applied to schedule the next request for service. The same auxiliary queue mechanism, as in CMQ, is adopted if the request is an update. Depending on the service discipline used for the auxiliary queue, there are also three variants. The simplest way is to service the auxiliary queue in a FCFS manner (RT-CMQ-F). Another way is to perform the SSTF (or SCAN) strategy for the auxiliary queue (RT-CMQ-S). The third alternative is to apply the window strategy to the auxiliary queue (RT-CMQ-W). As we shall see, there is no significant difference between these alternatives, since the queue length for the auxiliary queue is very short. Again, we use RT-CMQ to refer the RT-CMQ-S strategy in the remainder of the paper because it is used in most of our experiments.

3.4 The Modified FD-SCAN Policy

The FD-SCAN policy was first introduced in [1] for a single disk system, and is based on the classical SCAN policy. In FD-SCAN, the track location of the request with the earliest feasible deadline is used to determine the scan direction. A deadline is *feasible* if we estimate that it can be met. Determining the feasibility of a request's deadline is simple since once the current arm position and the request's track location are known, its service time can be determined. A request's deadline is feasible if it is greater than the current time plus the request's service time. At each scheduling point, all requests are examined to determine which has the earliest feasible deadline. After selecting the scan direction, the arm moves toward that direction and serves all requests along the way. A potential problem with this policy is the high cost to run it. For a mirrored disk, the DMQ policy can be modified to use FD-SCAN on each disk.

4 Performance Results

In this section, we illustrate the performance benefits gained by applying the previously described real-time scheduling policies on a mirrored disk. In section 4.1, we first describe the system parameter settings for our experiments, including: system configuration, transaction characteristics, deadline settings, and the parameters for the two real-time scheduling policies, RT-DMQ and RT-CMQ. The experimental results are given in section 4.2. In particular, we assess the system performance over a wide range of workloads, determine the impact of varying transaction deadline settings, and examine the system behavior by varying the read probability. Finally, we investigate the behavior of the modified policies DMQ and CMQ in a non-real-time environment.

4.1 System Parameter Settings

4.1.1 System Configuration

In our model, the disk has 1000 tracks. The database consists of 6000 pages, which are uniformly distributed on the disk with each page corresponding to a disk block. The disk service time is the sum of seek time, latency, and transfer time. The latency is assumed to be uniformly distributed among [0, 16.7] milliseconds, and the transfer time is considered to be a constant (0.8 ms). The seek time T_s is defined by,

$$T_s = \begin{cases} a + b\sqrt{i} & i > 0; \\ 0 & i = 0; \end{cases} \tag{1}$$

where a is the arm moving acceleration time (8 ms), b is the seek factor (0.5 ms), and i is the number of tracks for the arm to move [16,4].

4.1.2 Transaction Characteristics

Transaction characteristics and the load to the system are defined as follows: the number of users, N_u, which limits the maximum number of transactions in the closed system, ranges from 4 to 20. Each user may think for a random of time, which comes from an exponential distribution with mean $1/\mu_T = 1$ second, before generating a transaction. A transaction consists of a random number of operational steps, N_s, which is uniformly distributed between 1 and 20. Each step needs to access the database once and is followed by a manipulation of those data items fetched. The computation time of each step, T_C, is assumed to be 15 ms, and the time required to abort a transaction, T_A, is 5 ms. With these parameter settings, the workloads we use exhibit a I/O bound characteristic, which allows us to examine the difference in the disk scheduling policies.

In most of our experiments, the read probability, p_r, is set to 0.6, since more often users are querying the database. We also investigate the effect of varying the read probability from 0 to 1.

4.1.3 Deadline Settings

The deadline setting for each transaction in our model depends on the system load and the transaction's length. The system load can be characterized by the number of users in the closed system (with the mean think time $1/\mu_T$ fixed), and the transaction length corresponds to the number of steps. Specifically, we define T_{min} to be the average system time when there are no additional transactions in the system; it is given as

$$T_{min} = (T_C + T_D) * N_s$$

where T_C is the average step CPU time and T_D is the average disk service time during the execution of a single transaction. The value of T_D is approximately 25 ms.

The transaction deadline is set by,

$$Trans\_Deadline = T_{min} * \eta$$

where η is a *r.v.* drawn from a uniform distribution on $[DL\_LOW, DL\_UP]$, where DL_LOW is the *deadline lower bound* selected to be proportional to the number of users in the system, $DL\_LOW = min\{1, k * Num\_Users\}$ [3], and the DL_UP is the *deadline upper bound* which is a linear function of the lower bound, $DL\_UP = \alpha DL\_LOW$. In most of our experiments, k is equal to 0.25 and α is 3. However, we also examine the case where transactions' deadlines are very tight and/or very loose by varying DL_LOW.

The deadline setting for each I/O request is inherited from that of the transaction issuing the request in our experiments.

All of the above parameters are summarized in Table I, and II.

4.1.4 Parameters for SSEDO

The scheduling parameters β and the window size for the RT-DMQ and RT-CMQ policies are set to 2 and 3, respectively. These values are shown to be suitable for the window strategy operating in a single disk environment [6]. Certainly, a further tune up of these parameters may achieve some additional performance improvement.

[3] In any case, the deadline lower bound may not be less than 1. Otherwise, all the transactions might be lost

Table I: Disk Parameters

Tracks	1000
Seek Factor	0.5
Arm Acce. Time	8 ms
Latency	Uniform in [0, 16.7]
Transfer Time	0.8 ms

Table II: Transaction Characteristics

Num_Users	4 – 20
Mean Think Time	1 sec
Steps/per Trans.	Uniform in [1, 20]
Step Comp. Time	15 ms
Abortion Time	5 ms
p_r	0.6
DL_Lower Bound	0.25 * Num_Users
DL_Upper Bound	3 * DL_Lower Bound

4.2 Experimental Results

In this section, we report our experimental results. The performance of RT-DMQ and RT-CMQ policies are compared with their non-real-time counterpart DMQ and CMQ, as well as to the real-time policy SSEDO [6] and non-real-time policy FCFS for a system with single disk. Results of each experiment are averaged over 20 runs. In each run 1050 transactions are executed. 95% *confidence intervals* are obtained by using the method of independent replications. Confidence interval widths are less than 11% of the point estimates of the loss probability in all cases (these intervals are not shown in our figures in order to clearly show the results). For each run, the execution of the first 50 transactions are considered the transient phase and is excluded from our statistics.

4.2.1 Performance of RT-DMQ and RT-CMQ Policies

In this experiment, we explore the system performance under different I/O subsystems and disk scheduling policies.

Single Disk vs. Mirrored Disk: Figure 2 illustrates the performance is improved by introducing a mirrored disk. This improvement is due to the fact that the mirrored disk can support concurrent reads which has the potential of increasing the disk bandwidth and decreasing the I/O response time. The improvement is significant; The RT-DMQ and RT-CMQ can yield up to a 68% improvement over the real-time SSEDO policy and an

81% improvement over the FCFS policy for a single disk I/O subsystem. The two real-time policies, RT-DMQ and RT-CMQ, perform basically the same.

Real-Time Policies vs. Non-Real-Time Policies: In Figure 3, we plotted the performance of different real-time and non-real-time policies in combination with the DMQ policy. The original DMQ policy uses the FCFS discipline for its two queues, whereas the modified DMQ applies the SSTF discipline to each queue. The two real-time policies adopt SSEDO and FD-SCAN, respectively.

From this figure, we observe the importance of using policies that account for real-time constraints on a mirrored disk subsystem. As a transaction's deadline becomes closer, it is reasonable to assign it a higher priority to let it go first. As we can see, DMQ coupled with FCFS is the worst, since it takes care of neither the time constraint nor the service time. DMQ coupled with SSTF is better because it accounts for the service time factor when scheduling. The RT-DMQ coupled with SSEDO and FD-SCAN outperform the non-real-time policies, since they consider not only the disk service time but also the deadline information of I/O requests in making their decisions. These RT-DMQ policies differ in the weight placed on the two factors. While using FD-SCAN puts more weight on reducing the service time, using SSEDO puts more weight on the time constraint (e.g., it will schedule the request with earliest deadline most of the time). The curves show that the SSEDO variant performs slightly better than the FD-SCAN variant with the only exception being at an extremely high workload. Henceforth, RT-DMQ will refer to the RT-DMQ coupled with SSEDO.

Mean I/O Queue Length: The mean I/O queue lengths for different policies are illustrated in Figure 4. From this figure we observe that the I/O queue length for a single disk policy (SSEDO) is much longer than that for mirrored disk policies. The queue length for RT-DMQ-MR is greater than the queue length for RT-CMQ because of the duplicate read requests. An important observation is the small mean queue length (less than 0.2) for the auxiliary queue under RT-CMQ (also true for non-real-time CMQ). In fact, the probability that the auxiliary queue length exceeds 1 is less than 0.03. Thus, there is no benefit provided by the use of a scheduling policy such as SSTF for the auxiliary queue.

Mean Disk Service Time: The mean disk service time is of interest (Figure 5). When the system is lightly loaded, all of the mirrored disk policies can take advantage of allowing a read to go to the disk with the smallest seek time when both disks are idle at the time of its arrival. As system load increases, this advantage disappears since most new arrivals may not see both disks idle. However, all of the real-time policies begins to take advantage of the shortest seek time component inherent in the window strategy when the queue length is greater than one. The disk service time under the SSEDO policy decreases quickly as the load first increases and then remains at the same level. This is because the window size in our experiments is set to 3 and the I/O queue length exceeds 3 when the number of users is equal to 8 (Figure 4).

It is not surprising that the mean disk service time is greater under RT-CMQ than under RT-DMQ. This is because each disk is scheduled using separate SSEDO policies under RT-DMQ whereas a single SSEDO policy is used to schedule both disks under RT-CMQ. Recall that SSEDO attempts to reduce seek times as the number of the requests in the queue increases. Thus SSEDO operating under RT-DMQ is able to do so at both disks. However, in the case of RT-CMQ, SSEDO is only able to reduce the seek times of a write at one disk, when the queue length is large. The other write enters the auxiliary queue associated with the second disk. Unfortunately this queue rarely contains more than one request and so it is impossible to reduce the seek time of a write at this disk. Hence the overall disk service times are larger under RT-CMQ.

As expected, the mean disk service time is smallest under the DMQ policy that uses SSTF, especially at high loads, and FCFS has the highest mean disk service time.

Comparison of Variations of RT-DMQ and RT-CMQ Policies: The performance of the different variations of RT-DMQ and RT-CMQ policies are plotted in Figure 6 and Figure 7, respectively. From Figure 6, we observe that RT-DMQ-MR performs better than the other two policies. This is consistent with what we observed for a non-real-time environment in [19] where DMQ-MR exhibits the smallest mean response time from among the three variations. The RT-DMQ-SQ performs reasonably well, particularly as the system load increases. However, for the RT-CMQ policy, there is no big difference between the three variants, RT-CMQ-S, RT-CMQ-F, and RT-CMQ-W, due to the short queue length of the auxiliary queue under CMQ.

In Figure 6, we also plot RT-CMQ in order to compare it with the different RT-DMQ variants. Although our previous studies [19] for a non-real-time environment show that CMQ exhibits better performance than DMQ when each queue is served in a FCFS order, the reverse is true here. This is due to the fact that the mean disk service time is smaller under RT-DMQ than under RT-CMQ. The DMQ and CMQ policies studied in [19] used FCFS policies for all of the queues. Consequently, there was no difference in the mean disk service times under both policies and the benefit of having a common queue under CMQ was observed.

Mean Response Time for Committed Transactions: The mean response time for committed transactions under RT-DMQ and RT-CMQ are almost the same (Figure 8). The SSEDO policy for a single disk system, as shown in Figure 8, exhibits the smallest mean response time as the system load increases. This is because the window strategy used by SSEDO favors those transactions with tight deadlines, which, according to the way deadlines are determined (see in 4.1.3), are often associated with relatively short transactions. In high load cases, the transaction loss probability under SSEDO is high, and only short transactions are likely to commit. Therefore the mean response time for a committed transaction is the smallest. This conclusion is verified by Figure 9, where the size of committed transactions are plotted.

4.2.2 Varying Transaction Deadline Settings

This experiment is designed to examine the effects of the deadline settings on the performance of different disk scheduling policies. At one extreme the deadline is loose so that every transaction is successfully served and none are lost. At the other extreme, the deadlines are tight so that most transactions miss their deadlines. From the results shown in Figure 10, we observe that a mirrored disk subsystem outperforms a single disk subsystem, and that the RT-DMQ and RT-CMQ policies perform the best over the entire range of deadline settings. The average transaction response time for committed transactions is depicted in Figure 11. For the case of loose deadline settings, when nearly all the transactions can make their deadline, the mean transaction response time for a mirrored disk is almost half of that for single disk. This is because the I/O throughput is nearly doubled for a mirrored disk subsystem, especially when the read probability is high. On the other hand, when the deadline is tight, the mean transaction response time for a single disk subsystem is low because only short transactions are able to commit. It is interesting to observe that the transaction response time under DMQ is less than that of the RT-DMQ and RT-CMQ when the system loss ratio is small (less than 10%, see Figure 10 and Figure 11). This is reasonable since in this case, almost every transaction can commit successfully and the DMQ has been shown to exhibit the smallest disk service time (Figure 5). This results in a smaller response time for each transaction.

4.2.3 Varying the Read Probability

In this experiment, we study how the various disk scheduling policies behave as a function of the read probability, p_r. In our database model, an increase in update probability (i.e., a decrease in read probability) results in more lock conflicts since update requests require exclusive locks. This will result in more transactions being blocked waiting for locks and more deadlocks which in turn causes more transactions to be aborted. The increase in update probability results also in an increase in I/O load because more writes occur at the end of transaction execution. Both of these phenomena result in a decrease in the loss ratio as a function of p_r. The experimental results are shown in Figure 12. The transaction loss probability decreases for all of the I/O scheduling policies as a function of p_r. However, the mirrored disk subsystems remain more beneficial than single disk subsystems even as p_r approaches 1. This is again due to the presence of concurrent reads provided by the mirrored disk subsystem.

4.2.4 Mean Transaction Response Time in a Non-Real-Time Environment

Finally, we compare the performance of the modified version of DMQ and CMQ presented in this paper to that of original DMQ and CMQ policies introduced in [19] in a non-real-time environment. In our previous studies [19], we observed that the mean I/O response time for CMQ is less than that of DMQ when FCFS is used to service I/O requests. Figure 13 plotted the improvement of mean transaction response time produced by the modified DMQ and CMQ policies in a non-real-time database environment, where no time constraint is imposed on transactions. This is because SSTF used by the modified DMQ and CMQ reduces the disk service time, which in turn speeds up transaction processing. In contrast to our observation in [19], the mean transaction response time under the modified CMQ policy is greater than that of the modified DMQ policy. This is because the mean disk service time for CMQ is greater than that for DMQ due to the short queue length of the auxiliary queue in CMQ, for the reasons discussed in section 4.2.1. Among the three variations of DMQ, DMQ-MR provides the best performance, followed by DMQ-SQ. These results are consistent with the our previous observations in [19] where each queue is served in FCFS order. As the performance of CMQ(FCFS) is only marginally better than that of DMQ(FCFS), it is not included.

5 Summary

In this paper we examined the performance a mirrored disk I/O subsystem in an integrated real-time transaction system. Specifically, we first presented two real-time disk scheduling policies, RT-DMQ and RT-CMQ, for such a mirrored disk I/O subsystem, and then compared their performance with their non-real-time counterparts, DMQ and CMQ, as well as policies for a single disk I/O subsystem. In addition, we also considered another proposed SCAN based real-time policy, FD-SCAN. The results show:

- A mirrored disk I/O subsystem not only provides fault tolerance, but also improves system performance significantly for a real-time system in terms of the transaction loss ratio.

- If a mirrored disk is used in a real-time environment, then real-time disk scheduling policies should not only take into account the disk service time but also the time constraint of each I/O request.
- The RT-DMQ policy exhibits slightly better performance than the RT-CMQ policy. These qualitative results also hold between the DMQ and CMQ policies that use the SSTF policy while operating in a non-real-time environment.
- The difference in performance between a mirrored disk and a single disk increases as the probability that a request is a read increases.

Although our study focused on a specific real-time database system using specific CPU scheduling and concurrency control policies, we believe that the results will remain qualitatively the same for other database architectures and for other real-time applications. Finally, many interesting problems remain to be solved in the design of I/O scheduling policies for high performance real-time computer systems. We are currently studying the problem of scheduling disk arrays [15] in real-time systems.

Acknowledgment:

The authors wish to thank J. Huang for his invaluable help with the experiments on the RT-CARAT testbed.

References

[1] Abbott, R. and Garcia-Molina, H., "Scheduling I/O Requests with Deadlines: A Performance Evaluation," *to appear on RTSS 1990*, Dec. 1990.

[2] Baccelli, F. and E.G. Coffman, Jr., "A data base replication analysis using an M/M/m queue with service interruptions", *Performance Evaluation Review* Vol.11 (4), pp.102-107, 1982-1983.

[3] Bartlett, J. F., "A Nonstop Kernel", *Proc. Eighth Symp. on Operating System Principles*, pp.22-29, 1981.

[4] Bitton, D. and Gray, J., "Disk Shadowing," *Proc. of the 14 VLDB Conf.*, 1988, pp.331-338.

[5] Carey, M. J., Jauhari, R. and Livny, M., "Priority in DBMS Resource Scheduling," *Proc. of the 15th VLDB Conf.*, 1989.

[6] Chen S., Stankovic, J. A., Kurose, J. F., and Towsley, D., "Performance Evaluation of Two New Disk Scheduling Algorithms for Real-Time Systems," *COINS Tech. Report 90-77, UMass*, Aug. 1990.

[7] Coffman, E. G. Jr., H.O. Pollak, E.Gelenbe, and R.C. Wood, "An Analysis of Parallel-read Sequential-write Systems", *Performance Evaluation* Vol. 1, pp.62-69, 1981.

[8] Coffman, E. G. Jr., E.Gelenbe and B. Plateau, "Optimization of the number of copies in a distributed data base", *IEEE Trans. on Software Engi.*, 7(1) pp.78-84, 1981.

[9] Grossman, C. P., "Evolution of the DASD Control", *IBM Systems Journal*, Vol. 28, No. 2, 1989.

[10] Huang, J., Stankovic, J. A., Towsley, D. and Ramamritham, K., "Experimental Evaluation of Real-Time Transaction Processing," *Proc. Real-Time System Symposium*, pp.144-153, Dec. 1989.

[11] Huang, J., Stankovic, J. A., Ramamritham, K., and Towsley, D., "On Using Priority Inheritance in Real-Time Databases", *COINS Tech-Report 90-121*, Nov. 1990.

[12] Katzman, J., "A Fault-tolerant Computing System", in *The Theory and Practice of Reliable System Design*, D. Siewiorek and R. Swarz Eds., Digital Press, 1982.

[13] Matloff, N. S., "A Multiple-Disk System for Both Fault Tolerance and Improved Performance", *IEEE Trans. on Reliability*, Vol. 36, 2, pp. 199-201, June 1987.

[14] Nelson, R. D. and B.R. Iyer, "Analysis of a Replicated Data Base", *Performance Evaluation* Vol. 5, pp.133-148. 1985.

[15] Patterson,D., Gibson, G., and Katz, R. H., "A Case for Redundant Arrays of Inexpensive Disks (RAID)", *Proc. of ACM SIGMOD Conf.*, Chicaco, IL, June, 1988.

[16] Seltzer, M., Chen, P. and Ousterhout, J., "Disk Scheduling Revisited," *Proc. of USENIX, Winter'90* pp. 313-323.

[17] Son, S. H., "Using Replication for High Performance Database Support in Distributed Real-Time Systems," *Proc. Real-Time System Symposium*, Dec. 1987, pp.79-86.

[18] Teorey, T. J. and Pinkerton, T. B., "A Comparative Analysis of Disk Scheduling Policies," *ACM Communications*, Vol.5, No.3, March 1972, pp.177-184.

[19] Towsley, D., Chen S., and Yu S-P., "Performance Analysis of a Fault Tolerant Mirrored Disk System," *Proc. of Performance'90*, Sept. 1990.

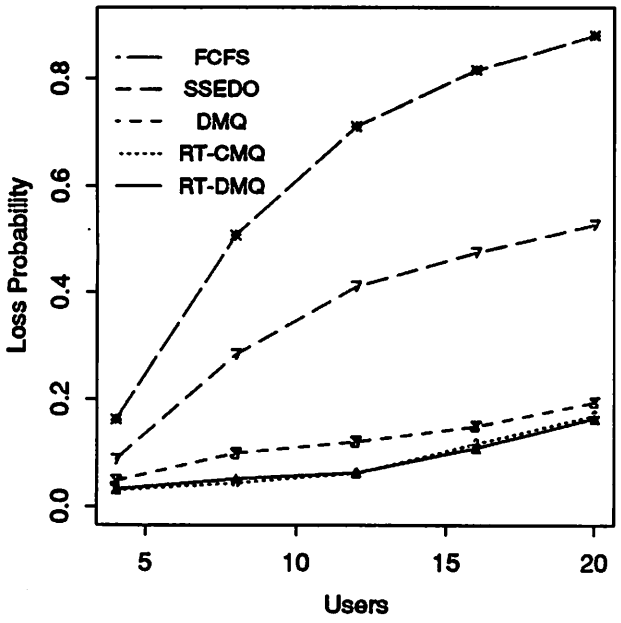

Fig.2: Performance of Single Disk vs. Mirrored Disk (Pr=0.6).

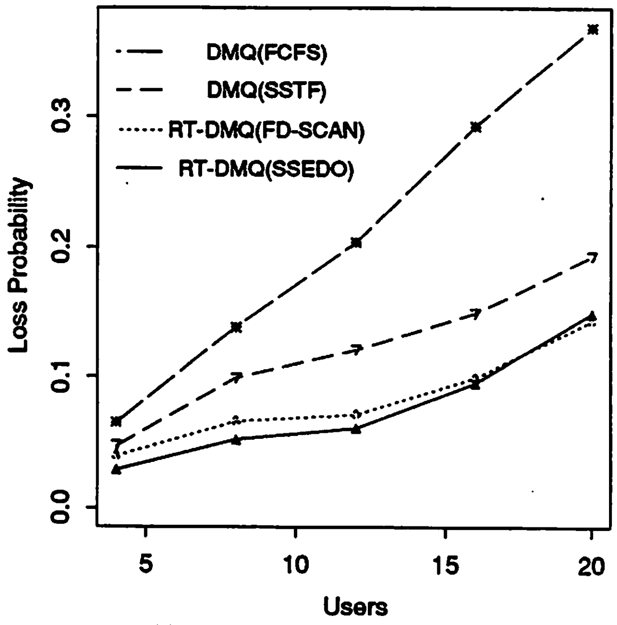

Fig.3: Performance of Real-Time vs. Non-Real-Time Algorithms. (Pr=0.6)

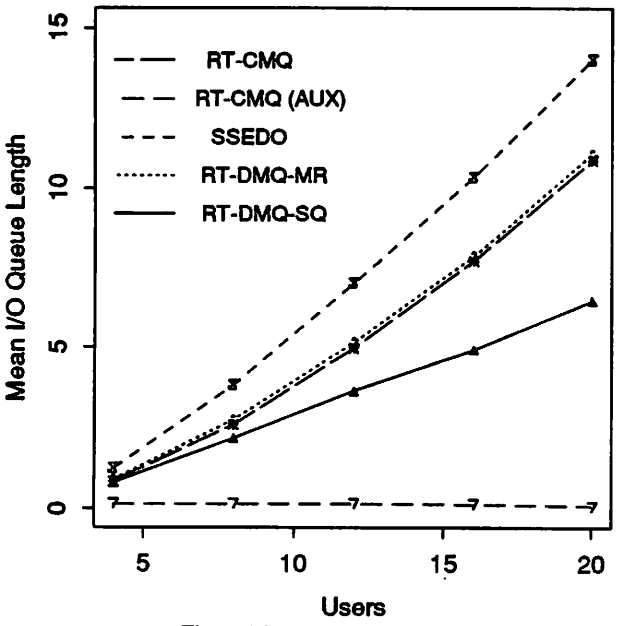

Fig.4: Mean I/O Queue Length. (Pr=0.6)

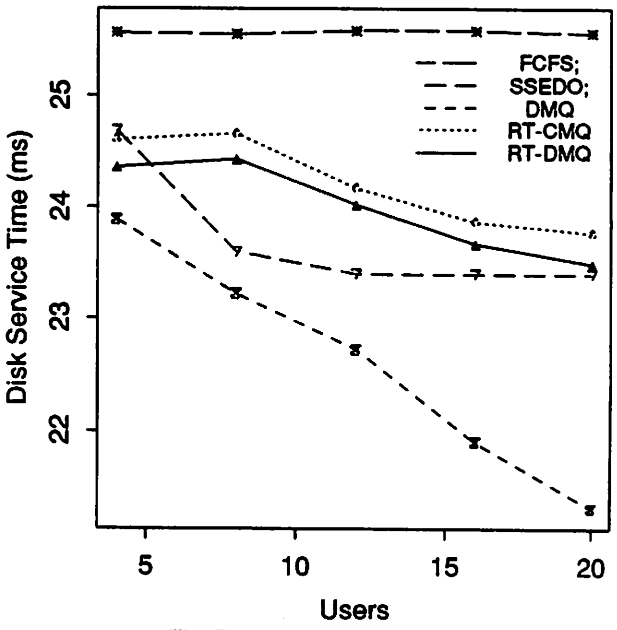

Fig.5: Mean Disk Service Time. (Pr=0.6)

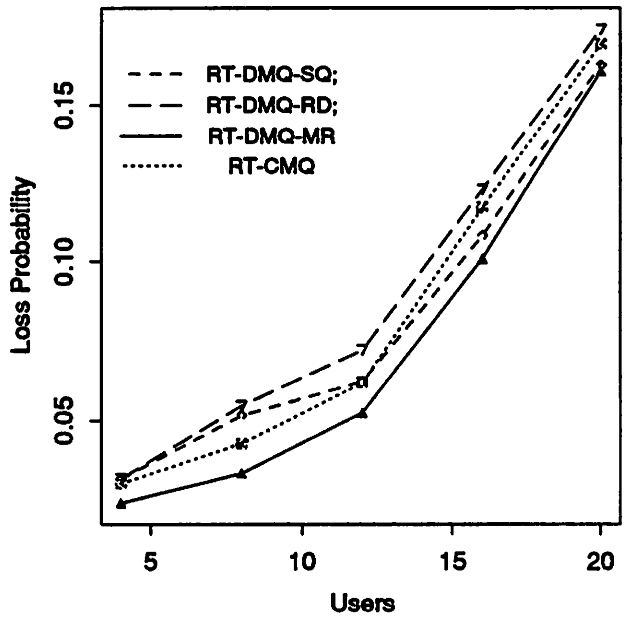

Fig.6: Comparison of Various RT-DMQ Algorithms.(Pr=0.6)

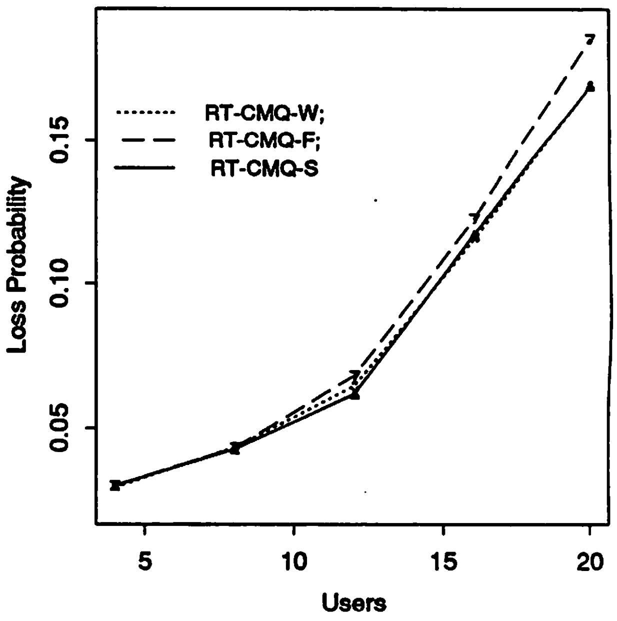

Fig.7: Comparison of Various RT-CMQ Algorithms. (Pr=0.6)

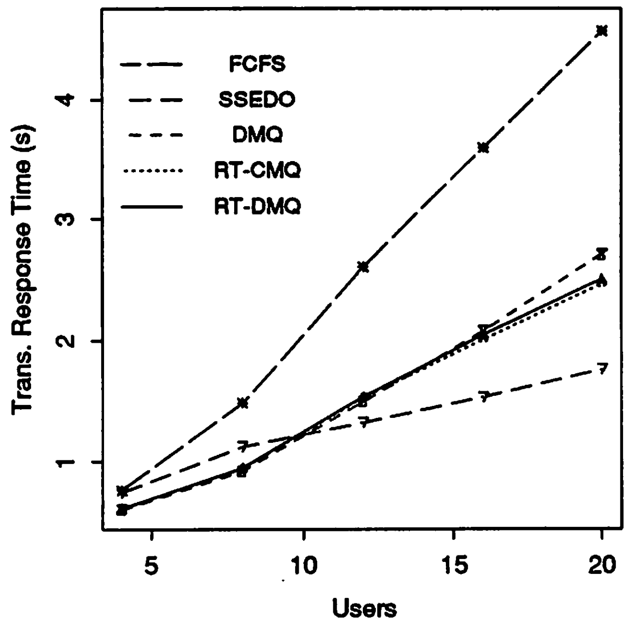

Fig.8: Mean Transaction Response Time.(Pr=0.6)

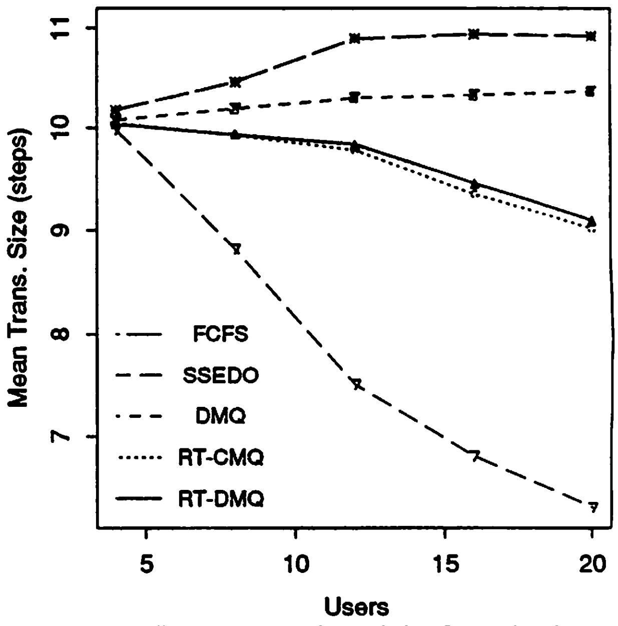

Fig.9: Average Length for Committed Transactions. (pr=0.6)

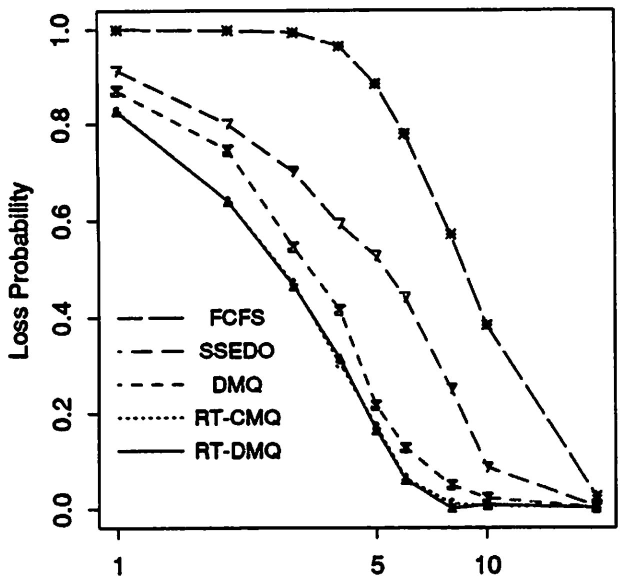

Fig.10: Varying Deadline Settings.
(Pr=0.6, U=20)

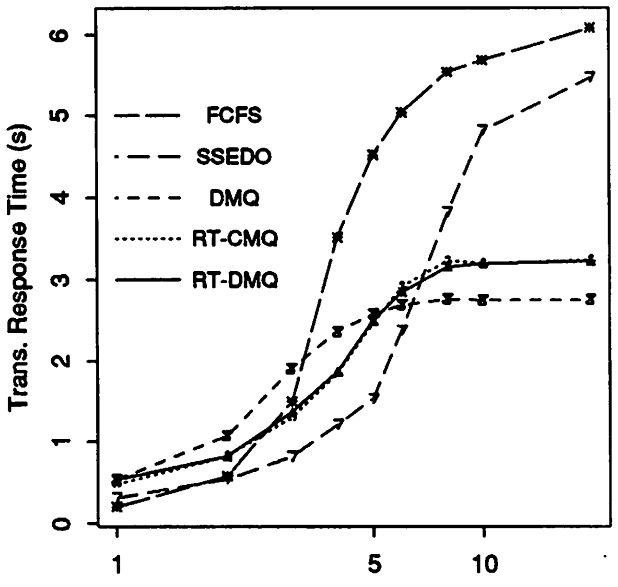

Fig.11: Mean Trans. Resp. Time under
Various Deadline Settings.(Pr=0.6,U=20)

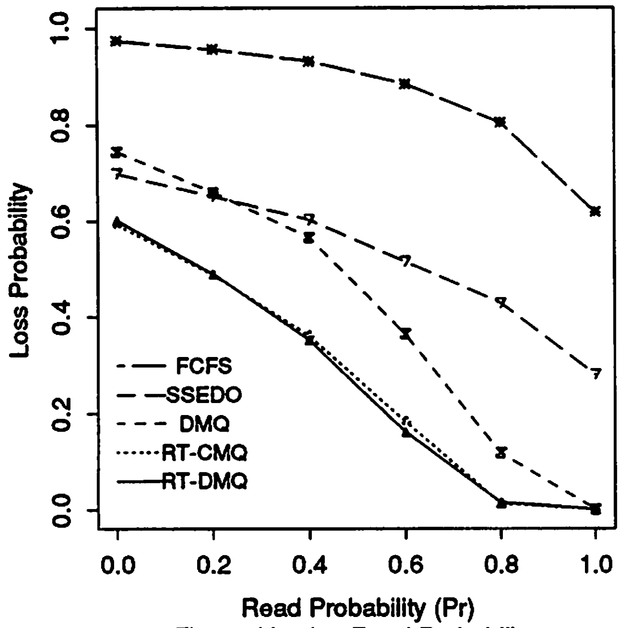

Fig.12: Varying Read Probability.
(Pr=0.6, U=20)

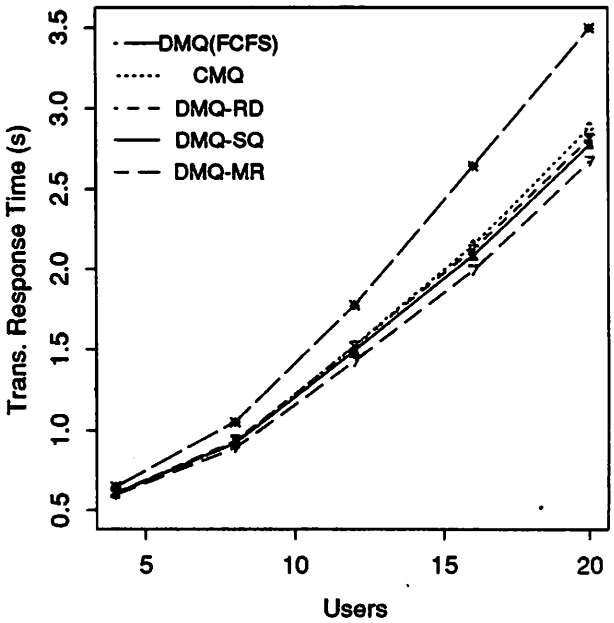

Fig.13: Trans. Resp. Time in a
Non-Real-Time Environment. (Pr=0.6)