Buffer Management in Real-Time Databases *

Jiandong Huang[1] and John A. Stankovic[2]
[1]Department of Electrical and Computer Engineering
[2]Department of Computer and Information Science
University of Massachusetts
Amherst, MA 01003

August 1990

Abstract

Buffer management plays a very important role in database systems. But little work has been done to study buffer management in real-time database systems. In this work, we propose and evaluate algorithms for real-time oriented buffer allocation and buffer replacement based on the existing organization of a real-time database testbed. Our goal is to increase the percentage of transactions meeting their deadlines. The experimental results obtained from the testbed indicate that under two-phase locking, the real-time oriented buffer management schemes do not significantly improve system performance; rather, other integrated processing components such as conflict resolution and CPU scheduling play a more important role in the system. We have shown that data contention is a constraint on the performance improvement of buffer management. Under data contention, conflict resolution becomes a key factor in real-time transaction processing. In addition, CPU scheduling is more important than buffer allocation, even if the system is not CPU bound. We discuss reasons for these results and give suggestions as to where and how real-time buffer management may improve real-time transaction performance.

*This work was supported by the National Science Foundation under Grant IRI-8908693 and Grant DCR-8500332, and by the U.S. Office of Naval Research under Grant N00014-85-K0398.

1 Introduction

Data buffering plays an important role in database systems where part of the database is retained in a main memory space so as to reduce disk I/O and, in turn, to reduce the transaction response time. The principle of buffer management is based on transaction reference behaviors [12]. In terms of *locality*, there are basically three kinds of reference strings in database systems:

1. *intra-transaction locality*, where each transaction has its own reference locality, i.e., the probability of reference for recently referenced pages is higher than the average reference probability.
2. *inter-transaction locality*, where concurrent transactions access a set of shared pages.
3. *restart-transaction locality*, where restarted transactions completely repeat their previous reference strings.

Buffer management policies should capitalize on one or more of these three types of locality.

Buffer allocation and buffer replacement are considered to be two basic components of database buffer management [8]. Buffer allocation strategies attempt to distribute the available buffer frames among concurrent database transactions, while buffer replacement strategies attempt to minimize the buffer fault rate for a given buffer size and allocation. The two schemes are closely related to each other and are usually integrated as a buffer management component in database systems.

In this work, we consider the buffer management in real-time database systems where transactions have timing constraints, such as deadlines. In a real-time environment, the goal of data buffering is not merely to reduce transaction response time, but more importantly, to increase the number of transactions meeting their timing constraints. To achieve this goal, buffer management should consider not only transaction reference behaviors, but also the timing requirements of the referencing transactions.

This study investigates one buffer organization which is based on the system structure of RT-CARAT, a real-time database testbed [10]. On RT-CARAT, we implemented a global buffer in connection with a transaction recovery scheme using after-image journaling. Based on the overall system structure, we study both buffer allocation and buffer replacement for the management of this global buffer which captures inter-transaction locality and restart-transaction locality. We propose several buffer management schemes that attempt to support real-time transactions in meeting their timing constraints. Our goal is to improve the system performance in terms of the percentage of transactions that meet their deadlines. The experimental results obtained from the testbed show, however, that under the concurrency control scheme using two-phase locking, the real-time oriented buffer management does not significantly improve the system performance over typical non real-time buffer management. We identified, through experiments, that in the integrated system, it is the conflict resolution and CPU scheduling, not the real-time oriented buffer management, that are the dominant factors for real-time transaction processing.

The rest of this report is organized as follows. In section 2, we give a brief review of the related work in buffer management where transaction timing constraints were addressed. Then, we present our overall buffer organization in Section 3. The proposed buffer allocation and replacement schemes are described in Section 4. Integration and implementation of our buffer management schemes are discussed in Section 5. In Section 6, we discuss the performance studies of the buffer management conducted on RT-CARAT. Finally, we make concluding remarks in Section 7.

2 Related Work

Even though buffer management in traditional database systems has been extensively studied [8, 14, 9, 6, 7, 3, 4, 5], little work has been done on time-constrained data buffering. Most recently, Carey et. al. [1] investigated some priority-based buffer management schemes which took an initial step towards the buffer management in real-time environment. [1]

The scheme in [1] incorporates transaction admission policies and buffer management policies. The later involves buffer allocation and replacement policies. When a transaction arrives at the system, the transaction admission policy checks whether there is enough buffer space available to allow the transaction to be *admitted* into the system. The arriving transaction may be blocked outside the system until sufficient memory becomes available. Once a transaction begins running and submits a request for a page that is currently not in the buffer pool, the buffer manager must determine the set of candidate buffers from among which one is to be *allocated* to the transaction. If there are no free buffers, the buffer manager determines which of the data pages currently in the buffer pool should be *replaced*. Different from buffer management in traditional database systems, priorities are considered in some of the three components when making decisions.

Besides priority-based buffer management, priority-based CPU and disk scheduling were also investigated in [1]. The performance of their proposals were evaluated through simulation. The results show that, regardless of whether the system bottleneck is the CPU or the disk, priority scheduling of the critical resources must be complemented by a priority-based buffer management policy.

With respect to [1], we note the following:

- It is the transaction admission control, not the allocation or replacement scheme, that leads to the performance gain. Indeed, considering the two proposed buffer management schemes, *Priority_LRU* and *Priority_DBMIN*, there is no special concern at all for priority-based buffer allocation and replacement in *Priority_DBMIN*, and the proposed buffer allocation and replacement policy in *Priority_LRU* does not perform well.

- The investigated CPU scheduling policy is a *non-preemptive priority-based round-robin* scheme. This scheme might not be well suited in real-time database environments.

[1] The concept of cache partitioning applied in [13] can be useful for buffer management in real-time database systems. Since its emphasis is on hardware architecture, we will not discuss it here.

Under these circumstances, it is doubtful whether buffer management is more important than CPU scheduling.

- [1] uses transaction response time as a performance metric. There is no special concern for transaction timing constraints, such as deadlines. How to use transaction timing information to manage buffer replacement or allocation has not been addressed.
- Since [1] focused on query-only environments, the buffer management did not deal with updates. It is not clear how the update operations would affect their performance results.

Different from [1], we will not consider the admission control for buffer management in this work. Instead, we focus on buffer allocation and replacement. We want to know how these two processing components perform in supporting real-time transactions. In addition, we explore buffer management in such an environment where update transactions exist. In other words, data contention is a main concern in our performance studies.

3 The Buffer Model Used in RT-CARAT

In RT-CARAT, an after-image journaling approach is used for database recovery. Under such a scheme, each transaction is allocated a piece of memory as its working space. If an (update) transaction is successfully committed, the "image" of this memory space will be written back to the permanent storage (disk) of the database; otherwise the "image" will be discarded. We view each piece of this memory space as a *private buffer (P_Buffer)* dedicated to each active transaction. We assume that there will always be enough memory space for the private buffers of concurrent transactions in the system. Clearly, the working space has twofold use. First, it is used for after-image journaling in terms of database recovery. Second, it acts as a buffer that captures the intra-transaction locality discussed in Section 1.

While keeping the existing structure of RT-CARAT, we implemented a *global buffer (G_Buffer)* which lies in between the pool of private buffers and the database disk. Figure 1 illustrates the system architecture with respect to data buffering. Here DM is a data manager process that carries out transaction execution. The global buffer is shared among the concurrent transactions. The global buffer also has twofold use. First, it may buffer the data shared among the concurrent transactions. Second, it can hold transaction working data such that a restarted transaction (aborted due to any reason) can fetch its data from the global buffer directly, eliminating the disk I/O. In other words, the purpose of using the global buffer is to capture the inter-transaction locality **and** the restart-transaction locality.

Based on the buffer organization presented here, we have designed algorithms for global buffer management.

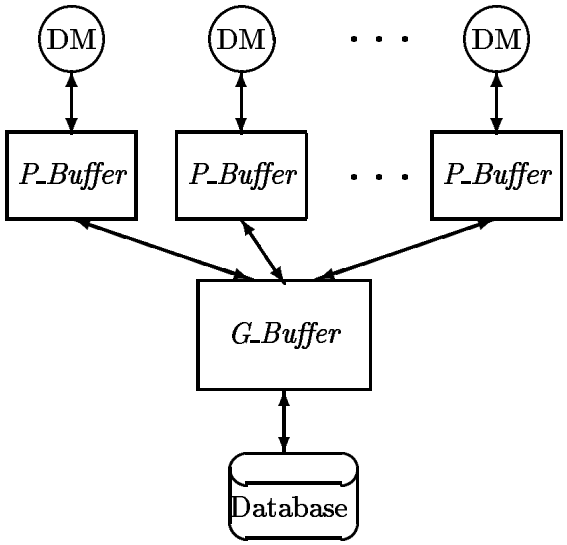

Figure 1: The buffer model

4 Buffer Management

As noted in the previous section, the buffer organization consists of two types of buffers - a pool of private buffers and a shared global buffer. For the private buffers, since they are used as transaction working space for the purpose of database recovery and it is assumed that there will always be enough space, there is no need to be concerned with allocation and replacement of private buffers. Therefore, we will not discuss the private buffer any further. In this work, we focus on global buffer management.

The global buffer management consists of two processing components:

- *buffer allocation*, which distributes global buffer space among concurrent transactions;
- *buffer replacement*, which is responsible for accessing of the global buffer and page replacement operations.

Upon arrival of a new transaction, the *buffer allocation* component decides if, and how much, global buffer space can be allocated to the transaction, according to a certain buffer allocation policy. When a transaction makes a request for a page, it will first access its private buffer (the working space). If the requested page is not found, the *buffer replacement* component searches the global buffer space allocated to it. If there is a hit in the global buffer, the transaction will use the page by copying it to its working space. Otherwise, the buffer replacement component will fetch the requested page from the disk to the global buffer, where page replacement operation may take place if there is no free space available in the buffer. Note that data consistency between database and global buffer must be maintained. This is guaranteed by the underlying concurrency control mechanism (e.g.

two-phase locking protocol) and a buffer write-through scheme. When a transaction comes to its commit stage, it checks if the original copy of any page modified in its working space exists in the global buffer. If yes, it will write the updated page back to the global buffer as well as the database.

In this study, we investigate buffer allocation and buffer replacement for the global buffer management. Although the two components are closely related to each other, the problem of allocating buffer space to real-time transactions in an optimal way is logically different from the problem of selecting a page for replacement. Thus, we treat buffer allocation and buffer replacement separately. In the following, we first discuss allocation schemes and then replacement schemes.

Here we define some notations that are used in the following subsections.

t - the current time of the system;
T_i - transaction i;
$T_i.dl$ - the deadline of transaction T_i;
$T_i.ws$ - the working set (pages) of transaction T_i;
$P\_buffer_{T_i}$ - the private buffer of transaction T_i;
G_buffer - the global buffer;
G_buffer_size - the global buffer size.

4.1 Buffer Allocation

The function of buffer allocation is to distribute the available buffer space among concurrent transactions. Buffer allocation appears more important when there is contention for the global buffer. Here the problem is how to let the concurrent real-time transactions make use of the limited buffer space. The basic idea for buffer allocation is to allocate the global buffer space to the real-time transaction(s) in such a way that the overall transaction deadline guarantee ratio can be improved compared to a buffer allocation scheme which ignores the timing information. In particular, we consider the following four policies for buffer allocation.

- **Alloc0**: *Allocating the global buffer to all the concurrent transactions.* Under this scheme, all the transactions are treated equally in using the global buffer, regardless of their timing constraints. This scheme represents a general method used for the buffer management in non real-time database systems. In this work, we use this scheme as a baseline for performance comparisons.

- **Alloc1**: *Allocating the global buffer to the transaction which has the earliest deadline among the concurrent transactions.* This scheme attempts to speed up the execution of the most time-critical transaction.

  Note that **Alloc1** may lead to low buffer utilization, since it allocates the entire global buffer space to only one transaction at a time. If the buffer space is relatively large, we may allocate the buffer to a group of transactions so that the buffer can capture

more inter-transaction locality and restart-transaction locality. The problem of low buffer utilization is overcome by the scheme described in the following.

- **Alloc2**: Let $T_i (i = 1, 2, ...n)$ be the total of n concurrent transactions in the system. The allocation scheme is described by the following algorithm.

  1. sort T_i by $T_i.dl$ in *ascending* order, for $i = 1, 2, ...n$;
  2. allocate the global buffer to the first m T_j's such that the following condition holds.

$$\sum_{j=1}^{m} T_j.ws \leq G\_buffer\_size < \sum_{j=1}^{m+1} T_j.ws \quad (1)$$

  This approach is an extension of **Alloc1**. Here the global buffer may be allocated to more than one transactions depending on the total buffer size and the size of transaction working sets. However, the basic idea is still the same, i.e. allocating the buffer space to the transactions with shorter deadlines.

- **Alloc3**: *Allocating the global buffer to any m concurrent transactions such that Eq.(1) holds.* This policy does not address transaction's timing constraints. This random allocation scheme is used for performance comparison with the allocation policy, Alloc2.

To describe how these 4 policies are actually utilized we use the concept of *buffer ownership*. A transaction is called a *buffer owner* if it has been allocated the global buffer space by any buffer allocation policy used; otherwise, it is called a *buffer user*. For any of its page references not in its private buffer, a transaction, no matter whether it is a buffer owner or a buffer user, will search the global buffer first. If there is a buffer hit, then the transaction will read the referenced page from the global buffer to its private working space ($P\_buffer_{T_i}$). The buffer owner and buffer user will behave differently in the situation where a buffer miss occurs: The buffer owner(s) can fetch the missing page from the disk to the global buffer, whereas a buffer user can only read the missing page from the disk to its private buffer.

With regard to transaction reference locality, this buffer allocation scheme captures (a part of) the inter-transaction locality by letting all the concurrent transactions access the global buffer. On the other hand, it captures the restart-transaction locality only for the buffer owners. Under this scheme, it is likely for the transactions with the buffer ownership to make better use of the global buffer, thus reducing their execution time.

4.2 Buffer Replacement

The replacement policy comes into play when there are no free buffer frames for newly fetched pages. In a real-time database environment, the replacement scheme should aim not only at minimizing the buffer fault rate, but also at maximizing the number of transactions in meeting their timing constraints. In this study, we examine two replacement policies:

- **LRU** - *replacing the least recently used page.* LRU is the one most commonly implemented in commercial database systems [8, 14]. In this study, it is used as a base algorithm for the purpose of performance comparisons.
- **LRU_dl** - *LRU with transaction deadline constraints.* This policy is a modification of LRU, in which we use transaction deadline information such that the pages in the buffer accessed by more time-critical transactions will not be as "easily" replaced compared to what would be done under the original LRU policy.

Now let us look at the LRU_dl replacement policy in detail. In LRU_dl, there is an LRU stack - a *logical* presentation of the G_buffer. Each entry of the stack has three fields:

- $LRU\_stack.pdl$ - the page deadline, which is the largest deadline value of the transactions that are accessing the page;
- $LRU\_stack.usr$ - the number of transactions that are accessing the page;
- $LRU\_stack.ptr$ - the pointer to the *physical* page residing in G_buffer.

We also define a parameter LRU_s_wnd, called the *search window* of the LRU stack, which is the distance from the bottom entry to the entry counted by LRU_s_wnd. With this data structure and the defined parameter, LRU_dl is described by the following algorithm.

Search the LRU stack backwards from the bottom up to LRU_s_wnd to replace one page,
  if a page satisfies ($LRU\_stack.pdl < t$) **or** ($LRU\_stack.usr = 0$), **then**
    replace the page
  if no such page is found within the search window **then**
    replace the page at the bottom of LRU stack;
  put the new page on top of the LRU stack.

There are three major concerns in LRU_dl. First, the basic LRU replacement discipline is used to capture the inter-transaction locality **and** the restart-transaction locality. Second, the replacement condition ($LRU\_stack.pdl < t$) takes transaction deadline information into account. Here, within the search window, a page held by a transaction already missing its deadline will be replaced. This policy tries to prevent the bottom page from being replaced if it is still being used by some active transaction still trying to meet its deadline. Third, the condition ($LRU\_stack.usr = 0$) enhances the algorithm performance for the situation where a page within the search window is not being used by any concurrent transactions while its LRU_stack.dl has not yet expired.

Note that the deadline search window is a critical parameter to LRU_dl. If the window is too small, LRU_dl may perform the same as the original LRU. On the other hand, if the window is large, without hardware assistance, LRU_dl may incur a large amount of overhead due to search operations. Even worse is that a large search window may destroy the property of capturing inter-transaction locality - one of our goals in the design of the algorithm. The sensitivity of search window setting is studied through experiments.

In summary, we plan to investigate the global buffer management in two aspects, i.e., buffer allocation and buffer replacement. The allocation and replacement algorithms presented here are considered to be orthogonal to each other. That is, they can be combined with each other in various ways to construct the buffer management component.

5 The Implementation

As discussed in the previous section, the global buffer management consists of two processing components - buffer allocation and buffer replacement. In the following, we show how these two components are integrated and implemented on RT-CARAT.

RT-CARAT is implemented as a set of cooperating server processes which communicate via efficient message passing mechanisms. In the system, there is one process, called *transaction manager* (TM), which is responsible for centralized control functions, such as CPU scheduling, message passing and commit operations. In addition, there is a pool of *data managers* (DM's) which carry out transaction operations such as data access and concurrency control. Based on the system organization, we embed the two components of the global buffer management in TM and DM processes, respectively. We let TM carry out the buffer allocation operation, and DM the replacement policy and buffer write-through operation. The following pseudo code gives the functional description of our implemented buffer management system.

- Buffer allocation (TM):

  Upon arrival of a new transaction,
  decide G_buffer ownership for each T_i based on **Alloc***i*, (i = 0, 1, 2, 3)

- Buffer replacement (DM):

  For each page reference of T_i not in P_buffer, search G_buffer.
  If G_buffer hit, **then**
  copy the page: $G\_buffer \Rightarrow P\_buffer_{T_i}$
  else
  read the page: $disk \Rightarrow P\_buffer_{T_i}$;
  If T_i has G_buffer ownership, **then**
  If there is no free buffer frame available for a new page, **then**
  do page replacement in G_buffer by **LRU** or **LRU_dl**;
  copy the page: $P\_buffer_{T_i} \Rightarrow G\_buffer$

The above is a high-level description of the implementation. The details differ from one algorithm to another. For instance, the implementation for LRU_dl involves the operations on data structures such as $LRU\_stack.pdl$, $LRU\_stack.ws$, and $LRU\_stack.usr$.

Note that a transaction's buffer ownership is dynamic. It is possible that a particular T_i may be a G_buffer owner upon arrival and later be preempted (with respect to the ownership) because other transactions with earlier deadlines arrive after it.

6 Experimental Results

In this section, we discuss some performance studies conducted on RT-CARAT. Our goal is to analyze the effectiveness of the proposed buffer management schemes in the presence of various overheads and resource (data, CPU and I/O) contentions. In addition, we want to know how important the real-time oriented buffer management is in real-time databases, as compared to other processing components such as CPU scheduling, conflict resolution and I/O scheduling.

6.1 The Test Environment

Currently, the RT-CARAT testbed is a centralized, secondary storage real-time database system. The testbed is complete. It contains all the major functional components of a transaction processing system, such as transaction management, data management, log management, and communication management. The concurrency control approach used in the experiments is two-phase locking. The recovery scheme, as mentioned in the previous sections, is based on after-image journaling mechanism.

Besides the global buffer management, there are two other major processing components, *CPU scheduling* and *conflict resolution*, considered in the experiments. From our previous studies [10], it has been shown that CPU scheduling and conflict resolution play an important role in real-time database systems. In a real-time database using a global buffer, we want to know how important the buffer management is compared with them. Here we consider the following algorithms for CPU scheduling and conflict resolution, respectively.

- CPU scheduling policies:

  SCH0 - a multi-level feedback queue;
  SCH1 - earliest deadline first.

- Conflict resolution policies:

  CRP0 - the lock-holding transaction will never be aborted;
  CRP1 - Among the conflicting transactions, the one with the earliest deadline becomes the lock holder. Others with compatible locks can also hold the lock.

Note that SCH1 and CRP1 are the real-time oriented policies, while SCH0 and CRP0 are the policies commonly found in traditional database systems. In the experiments, SCH0 and CRP0 are used as default algorithms for CPU scheduling and conflict resolution, unless otherwise specified.

RT-CARAT is a closed queueing network. Users submit transaction requests one after another. A transaction performs a certain number of predefined operations, called *steps*, and each operation may access a certain number of records and do a certain amount of computation. A transaction terminates upon completion or a termination abort. The transaction deadline is randomly generated from a uniform distribution within a deadline

window, $[d\_base, \alpha \times d\_base]$, where *d_base* is the window baseline and α is a variable determining the upper bound of the deadline window. For each workload in the experiments, *d_base* is specified first by the formula:

$$d\_base = avg\_rsp - stnd\_dvi$$

where *avg_rsp* is the average response time of the same real-time transactions when executed in a non real-time database environment, and *stnd_dvi* is the standard deviation of the response time.

Table 1: Experimental Settings

Parameter	Setting
Disks	disk1: database; disk2: log.
Database size	1000 blocks (6000 records)
Multiprogramming level	8
x (steps per transaction)	8 steps
y (records accessed per step)	3 records
z (computation time per step)	0 units
GB_size	0 - 500 blocks
Access distribution	uniform, skewed
P_w (prob. of *write* transactions)	0.05 - 0.30
α (deadline window factor)	2.0 - 6.0
IO (I/O operations per I/O request)	1 - 10
LRU_s_wnd (search window)	0 - 50

Table 1 summarizes the parameter settings in our experiments. The table is divided into two parts. The first part presents the parameters that are kept constant across all workloads, and the second part are those that change according to the different performance measurements. In our experiments, two separate disks are used, one for the database and the other for the log. The database consists of 1000 physical blocks (512 bytes each) with each block containing 6 records for a total of 6,000 records. In all the experiments, the multi-programming level in the system is 8. Transaction length is described by $T(x, y, z)$, where x is the number of steps per transaction, y is the number of records accessed in each step, and z is the computation time per step. For these experiments, this set of parameters are fixed at $T(8, 3, 0)$.

A critical factor in buffer management is the global buffer size - *GB_size*. A small buffer size may cause buffer contention among concurrent transactions. In our experiments, we study various effects on buffer performance by varying the buffer size. Access distribution is another important factor in studying buffer management. In order to exercise the proposed buffer management protocols with different data access patterns, we consider both uniform and skewed access in our experiments[2]. In addition, we choose P_w, the probability of write

[2]Note that the reference string from each transaction is random. The looping behavior is not considered, since intra-transaction locality is captured by the private buffer in our buffer model which is not the focus of this study.

transactions among the concurrent ones, as a variable, since it affects transaction conflict rate and hence restart-transaction locality. Also, deadline window factor, α, is a timing-related parameter which specifies the deadline distribution of real-time transactions. The smaller the α value, the tighter the transaction deadlines and vice versa. In our performance studies, in order to create an I/O-bound system, we have to physically increase the number of disk accesses for each I/O request. The number of such I/O operations is specified by the parameter IO. For instance, if IO is set to 1, each I/O request will lead to one disk access operation; if IO is set to 4, then each I/O request will perform four disk access operations (three of which are redundant). Finally, the parameter LRU_s_wnd is a variable used in the performance studies on buffer replacement policies.

Note that the notion of *working set* is used in our proposed buffer management schemes. But for the tested workloads, we have no exact knowledge about the working set of each transaction. Here we extend the transaction working set to a larger data group, called *working page set*, which is equal to $x \times y$, assuming that every record reference ends up at a different page. In practice, any transaction's working set is always less than or equal to its working page set.

In the following experiments we use the following metrics for performance evaluation.

- Deadline guarantee ratio - the percentage of submitted transactions that complete by their deadlines.
- Total hit ratio - the percentage of buffer references that result in a buffer hit.
- Weighted hit ratio (WHR) -

$$WHR = \frac{1}{buffer\ references} \times \sum_{i=1}^{n} hit(i) \times 2^{(n-i+1)}$$

  where n is the maximum number of concurrent transactions; $hit(i)$ is the number of buffer hits resulting from the transaction that has the ith smallest deadline value among the n concurrent transactions. In other words, every time a buffer hit occurs, we compare the deadline of this referencing transaction with those of other concurrent transactions. The counter $hit(i)$ is incremented by one if the referencing transaction's deadline is the ith smallest among the n concurrent transactions. We weight $hit(i)$ by an exponential factor $2^{(n-i+1)}$. Obviously, the smaller the i, the larger the $hit(i) \times 2^{(n-i+1)}$. For a given number of buffer references, the more the buffer hits from referencing transactions with short deadline values, the larger the WHR will be. Thus, from the measurement of WHR we can tell how the buffer allocation algorithms perform with respect to transaction timing constraints. With the same buffer hit ratio, the larger the WHR, the better the buffer allocation algorithm.

- Hit-deadline ratio - the percentage of buffer references that result in a buffer hit from transactions which have not missed their deadlines yet.

  This metric is used for the evaluation of buffer replacement algorithms. For a given number of buffer references, the greater the number of the buffer hits resulting from

referencing transactions whose deadlines have not expired yet, the larger the hit-deadline ratio, and the better the buffer replacement algorithm.

We also collect statistics on CPU utilization, I/O utilization, response time, and transaction restart ratio.

The data collection in the experiments is based in the method of replication. The statistical data has 95% confidence intervals with no greater than $\pm$ 2% of the point estimate for deadline guarantee ratio. In the following graphs, we only plot the mean values of the performance measures.

6.2 Experiments

The experiments consist of three parts:

1. **System calibration**: This experiment attempts to characterize the testbed system along the dimensions of CPU overhead, I/O overhead, buffer size, and data access distribution. This measurement is essential to our performance studies, since we need to set up system parameters and to identify proper workloads for the following experiments.
2. **Buffer management with buffer allocation**: The purpose of this experiment is to study the proposed real-time buffer allocation schemes.
3. **Buffer management with buffer replacement**: In this experiment we investigate the proposed real-time buffer replacement scheme.

6.2.1 System calibration

The previous performance studies conducted on RT-CARAT show that RT-CARAT is a CPU-bound system [10]. In such a system, a software-oriented buffer management component might not be able to improve system performance due to CPU contention. To study the performance of the proposed buffer management schemes, we need to work with a workload where the buffer management system can improve performance. To identify such a workload, we increase I/O operations in each transaction step, making the system vary gradually from CPU-bound to I/O-bound. We calibrate the system by measuring its CPU utilization, buffer hit ratio, and transaction response time as functions of the I/O operations.

In this experiment, since we focus on system calibration, we consider a non real-time database system where transactions have no timing constraints. The buffer management component is Alloc0 and LRU for buffer allocation and buffer replacement, respectively. Transactions are equal in length with $T(x, y, z) = T(8, 3, 0)$ and $P_w = 0.2$.

Figures 2 and 3 show the system performance in terms of CPU utilization, with uniform access and skewed access, respectively. The skewed access follows the 80-20 rule, i.e., 80% of the invoked transactions access 20% of the database. As one would expect, the CPU utilization decreases as I/O operations increase. Here an important observation is that the

CPU utilization with $GB\_size > 0$ is much higher than that with $GB\_size = 0$. In general, CPU utilization will increase as buffer size increases, since the larger the buffer, the more references will be in the buffer, and the more I/O operations will be eliminated. But the results in Figures 2 and 3 show that the buffer management leads to an increase of CPU utilization, even when the buffer size is very small ($GB\_size = 25$). This means that the buffer management in RT-CARAT does have significant overhead.

The measures of buffer hit ratio are illustrated in Figures 4 and 5. The results restate the knowledge that buffer hit ratio is independent of I/O operations, and is increased by skewed access as well as by increase of buffer size. Here we just want to show the relative performance of the global buffer with respect to the parameters of buffer size and access distribution.

Figures 6 and 7 plot the transaction response time as a function of I/O operations. It is interesting to note that when I/O operations are less (i.e., when the system is CPU-bound), the transaction response time under buffer management is even longer than without data buffering (i.e. $GB\_size = 0$). This is because, as we discussed above, the implemented buffer management component has relatively large overhead. The buffer management will not improve system performance unless the CPU utilization is relatively low and I/O overhead is relatively high. For the workload tested in this experiment, the buffer management is seen to be useful only when the CPU utilization is lower than 80% or $IO > 4$. Another critical factor to the system performance is buffer size. If the size is too small to capture the transaction reference locality, the buffer management will do nothing but incur overhead to the system. This can be clearly seen in the two figures for $GB\_size = 25$.

The results from this experiment have identified a workload setting, i.e. $IO = 4$, beyond which the system will benefit from global buffer management. Thus, we should start our performance studies on the global buffer management with $IO > 4$.

6.2.2 Buffer management with buffer allocation

In this set of experiments, to focus on the performance study of real-time oriented buffer allocation schemes, we choose LRU as buffer replacement policy. In other words, the global buffer management considered here is a combination of LRU replacement policy with various proposed buffer allocation schemes. Buffer size is a critical parameter to system performance. Since we study the buffer allocation policies, we need to consider the situation where buffer contention exists. Hence the buffer size is varied within a range from the value around a single *working page set* of a transaction (24 blocks) to the value around the total *working page sets* of concurrent transactions (192 blocks). To create a high inter-transaction locality and restart-transaction locality, we exercise the workloads with skewed access using 80-20 rule.

The effectiveness of buffer allocation schemes

We first determine the effectiveness of the proposed real-time oriented buffer allocation policy (Alloc2) through experiments.

Figures 8-11 show the performance of buffer allocation schemes with respect to CPU utilization, deadline guarantee ratio, buffer hit ratio, and weighted hit ratio, with $IO = 6$, $P_w = 0.2$ and $\alpha = 4.0$. The CPU utilization shown in Figure 8 indicates again that buffer management does have large overheads. With $GB\_size = 30$, CPU utilization increases at least 30% once the buffer management is employed. Among the four allocation schemes, Alloc0 incurs the largest overhead, since the scheme allocates the global buffer space to all the concurrent transactions. In that case, every buffer miss will result in page fetching, and even replacement operations, for the global buffer. In contrast with Alloc0, the overhead of Alloc1 appears the lowest. This is because under Alloc1, there is only one buffer owner at any time. Remember that it is only the buffer owner that can do page fetching and replacement for the global buffer.

Figure 9 illustrates the performance of real-time transactions with respect to deadline guarantee ratio. As one would expect, the deadline guarantee ratio increases as the buffer size increases. Considering the buffer allocation schemes, however, there is no significant performance difference. Figure 10 plots the buffer hit ratio as a function of GB_size. Again, there is no significant performance difference among the proposed allocation schemes. Note that the random allocation scheme, Alloc3, performs the same as Alloc2 which is expected to provide best performance among the four protocols.

We also exercised the workloads with the deadline setting $\alpha = 2$. The performance results are basically the same as what we presented above.

At this point the reader may wonder if the allocation schemes function as intended. The answer is yes. We measured the effectiveness of the allocation schemes by WHR, the weighted hit ratio. From Figure 11 we can see that Alloc2 outperforms all other schemes with respect to the weighted value, i.e., Alloc2 does allocate the available buffer space to more time-critical transactions as compared to other schemes. We also observed that the performance difference between Alloc2 and others (except Alloc1) diminishes as the buffer size becomes large. This is because when the buffer size is large enough, it will capture the inter-transaction locality and restart-transaction locality for all the concurrent transactions, regardless of their timing constraints. Alloc1 becomes the worst as the buffer size increases, since this single ownership allocation scheme captures the least transaction reference locality. Comparing the two non real-time allocation schemes, Alloc0 and Alloc3, Alloc3 has larger WHR value than Alloc0. This is because unlike Alloc0 which lets all the concurrent transactions share the global buffer space, Alloc3 allocates the buffer only to the limited number of concurrent transactions, depending on the available buffer size. Due to its non real-time nature, however, Alloc3 does not performs as well as Alloc2 does.

Our experiments have shown that the proposed buffer allocation scheme Alloc2 does work in favor of more time-critical transactions. But with respect to the metric - deadline guarantee ratio, the use of Alloc2 does not improve performance.

In a locking-based database system with limited CPU and I/O resources, transaction deadline guarantee ratio under (real-time oriented) buffer management may be constrained by one or more of the following factors: It may be that the overhead from protocol implementation overshadows the benefit from buffer management. Even if the overhead from buffer management is negligible, CPU contention can still block transaction processing. Similarly,

I/O may be a bottleneck when the system becomes I/O bound. Finally, data contention may result in transaction blocking due to the use of locking scheme for concurrency control.

We have shown in the above discussion that the overhead incurred from buffer management is large and not negligible. In order to get performance gains from data buffering, we increased physical I/O operations. As a result, the implementation overhead is no longer a significant factor. To identify the main reason why the proposed buffer allocation scheme does not improve the performance of real-time transactions, we need to analyze the situations where resource contention (data, CPU, and I/O) exist.

In the following experiments, we only compare Alloc2 with two baselines NBuf (no global buffer) and Alloc0. We still consider the situation where buffer contention exists. From previous experiments (see Figure 11), we know that in terms of weighted hit ratio, there is a large performance difference between Alloc2 and Alloc0 when *GB_size* is less than 150 blocks. In the following set of experiments, we set *GB_size* at 90 blocks, which is large enough to hold about half of the total *working page sets* of concurrent transactions and yet "small enough" to show the performance gain of Alloc2 over Alloc0 with respect to weighted hit ratio. The deadline setting is $\alpha = 4$.

Buffer allocation vs. conflict resolution

In this experiment, we consider the use of CRP1 so as to determine if real-time oriented conflict resolution is more important than the buffer management in the presence of data contention. To create data contention, we gradually increase P_w from 0.05 to 0.30.

Figure 12 compares the weighted hit ratio for the buffer management with and without applying CRP1. First of all, the four curves show increasing hits as P_w increases. This is because an increase of P_w results in an increase in the data conflict rate, and in turn, the increase of restart-transaction locality, and the increase of the weighted hit ratio. Second, applying CRP1 increases the weighted hit ratio for the allocation scheme Alloc0 but not for Alloc2. This can be explained as follows. Under CRP0, the transaction aborts result from deadlock only, whereas under CRP1 most transaction aborts result from conflict resolution. Overall, the transaction abort rate under CRP1 is higher than that under CRP0 for any given P_w. Note that it is the transactions with longer deadlines that are aborted under CRP1. For Alloc2, since it allocates the buffer space only to some of the concurrent transactions with shorter deadlines, the increased restart-transaction locality (the increased abort rate) due to the use of CRP1 will not affect the weighted hit ratio. For Alloc0, because of its random allocation policy, the increased restart-transaction locality will lead to the increase in overall buffer hit ratio, and, in turn, the weighted hit ratio.

The transaction deadline guarantee ratio is plotted in Figure 13. The reader can clearly see that under CRP1 transactions perform much better than they do without the use of CRP1. This holds even for the system where no global buffer is employed. Comparing Figures 12 and 13, we know that the performance gain does not come from the buffer management, otherwise NBUF_CRP1 would not outperform NBUF and LRU_Alloc2_CRP1 would not outperform LRU_Alloc2 either. It is CRP1 - the real-time oriented conflict resolution that significantly improves the performance.

Buffer allocation vs. CPU scheduling

From the above experiment we have seen that it is the real-time oriented conflict resolution (CRP1), not the real-time oriented buffer allocation scheme, that improves transaction performance in the presence of data contention. Now we examine what kind of role CPU scheduling plays in connection with the buffer management.

Figures 14, 15 and 16 depict the performance of real-time transactions versus IO, with respect to CPU and I/O utilization, weighted hit ratio and deadline guarantee ratio. Here the highest CPU utilization, as shown in Figure 14, is less than 80%, which means that there is no severe CPU contention in the system. When SCH1 is applied, we can still see a slight performance improvement in terms of deadline guarantee ratio in Figure 16. Comparing Figures 15 and 16, especially the curves of LRU_Alloc2 (Alloc2, but no SCH1), LRU_Alloc0_SCH1 (SCH1, but no Alloc2), and LRU_Alloc2_SCH1 (both Alloc2 and SCH1), we notice that the performance gain does not come from real-time oriented buffer management, otherwise LRU_Alloc2 would outperform LRU_Alloc0_SCH1 in terms of deadline guarantee ratio, since its weighted hit ratio is much higher than that of LRU_Alloc0_SCH1.

Discussions

As we mentioned above, the real-time oriented buffer management may be constrained by one or more factors. Through carefully designed experiments we have shown that one such factor is data contention. Under high data contention, the performance improvement can be achieved through real-time oriented conflict resolution, but not real-time oriented buffer allocation.

The conducted experiments also indicate that CPU scheduling is more important than buffer allocation. Note that to avoid the negative effect from the implementation overhead of the buffer management, we created a system (by increasing physical I/O operations per I/O request) which is not CPU bound. In other words, for the tested workloads (IO=4,5, ...,8), there was no strong CPU contention. For a CPU bound system, we can forsee, from our experimental results obtained here, that CPU scheduling is the key factor to system performance and real-time oriented buffer allocation is not important even if the overhead of the buffer management is negligible.

Another factor which may block transaction processing is disk I/O. Under I/O contention, the real-time oriented buffer allocation scheme is expected to play an important role in improving the performance of real-time transactions, since the essential goal of data buffering is to reduce disk I/O. But on the other hand, the allocation scheme may not be helpful if data contention exists or reference locality is not high. In our experiments, the I/O utilization is over 80% (see Figure 14) and the average I/O queue length is over 2.3 (not plotted), for LRU_Alloc0 and LRU_Alloc2, when $IO \geq 7$. In such an environment, we do not see any significant performance improvement due to the real-time oriented buffer allocation scheme. Here the main reason, we believe, is the data contention - the issue we just discussed above.

Another issue we haven't addressed is I/O scheduling. We would like to know in an I/O bound environment how the I/O scheduling affects the system performance as compared

to the buffer allocation scheme. Unfortunately, we can not explore this interesting issue due to limitations on the physical testbed. A recent study based on simulation [2] has demonstrated that I/O scheduling is a key processing component in real-time database systems. It is likely that I/O scheduling is more important than the buffer allocation.

The last factor we want to point out is the buffer contention. Our discussion on buffer allocation has focused on the situation where buffer contention exists. We notice from the performance results that the allocation schemes Alloc0 and Alloc2 will produce the same performance when the buffer size is large enough to hold the total working page sets of the concurrent transactions (see Figure 11). Obviously, we do not need to worry about buffer allocation if buffer contention is no longer a problem.

6.2.3 Buffer management with buffer replacement

In this experiment, we investigate the proposed buffer replacement scheme. To focus on the performance of buffer replacement, we choose Alloc0 - the simplest scheme for buffer allocation. Also, to make the replacement operation meaningful, the buffer size should be chosen at least as large as the total *page working sets* of concurrent transactions. In the experiments, the buffer size is first set at 250 blocks which is larger than the total page working sets of 192 blocks for $x = 8$ and $y = 3$. Transactions access the database uniformly with $P_w = 1.0, \alpha = 6$ and $IO = 6$. Under such a workload, the measured CPU utilization is about 40% and the I/O utilization is above 90%. The measured transaction restart ratio is around 15% under CRP0 and 55% under CRP1, respectively.

We first examine the effectiveness of the proposed buffer replacement scheme, LRU_dl. Figure 17 shows the total hit ratio and the hit-deadline ratio versus the LRU search window (LRU_s_wnd). As compared with the total hit ratio under LRU policy which is irrelevant to the search window, the total hit ratio under LRU_dl decreases as LRU_s_wnd increases. This is due to the fact that LRU_dl breaks the LRU discipline, i.e., it may replace a page which is not the least recently used. On the other hand, LRU_dl reduces the buffer hits mostly for those transactions that have missed their deadlines. This is illustrated by the curve of hit-deadline ratio, where the ratio is a constant (25%) when LRU_s_wnd is less than 20. Remember that the global buffer is designed to capture the inter-transaction locality as well as restart-transaction locality. Increasing LRU_s_wnd will reduce buffer hits resulting from inter-transaction locality. This is why the hit-deadline ratio decreases too when LRU_s_wnd is greater than 20. Note that the hit-deadline ratio becomes a constant again when LRU_s_wnd is larger than 40. This is because for the exercised workload, the conditions for page replacement (see LRU_dl algorithm) can often be satisfied within last 40 pages of the LRU stack. Overall, the results presented here indicate that the proposed buffer replacement algorithm LRU_dl is effective in terms of preventing the not-yet-missing-deadline pages from being replaced from the global buffer. However, it is not good enough to raise the hit-deadline ratio.

To see how LRU_dl affects the overall transaction performance and to see how other processing components, like *conflict resolution policy*, performs as compared to buffer management, we examine the transaction deadline guarantee ratio under LRU, LRU_dl, LRU-CRP1 and LRU_dl-CRP1. The results in Figure 18 show that LRU_dl performs basically

the same as LRU no matter whether CRP1 is applied or not. This is understandable since LRU_dl does not change the hit-deadline ratio. On the other hand, incorporating CRP1 into the system increases the transaction deadline guarantee ratio from 79% to 90%, achieving about 14% performance improvement.

We further exercised workloads with different buffer sizes, deadline distribution, and access distribution (skewed access). The results are similar to what we have shown above. For the sake of brevity, we do not illustrate those results here.

Discussions

Our experiments have shown that buffer replacement in real-time database systems is not a simple issue. The proposed replacement algorithm, LRU_dl, attempts to balance the trade-off between capturing transaction locality and meeting transaction timing constraints. In practice, the performance of the algorithm is complicated by system parameters and different workloads, such as buffer size, data access distribution, transaction deadline distribution, data contention, resource (CPU and I/O) contention as well as the buffer search window.

However, the changes of various parameters and workloads in the experiments do not change the performance of the algorithm, i.e., it does not improve the buffer performance with respect to hit-deadline ratio and deadline guarantee ratio. This may result from the following reasons. First, the simple LRU is superior to the sophisticated LRU_dl. The mechanism used in LRU supports the basic idea of "keeping the pages in the buffer for not-yet-missing-deadline transactions". Second, in an integrated system, like RT-CARAT, the performance of buffer replacement schemes may be limited by other processing components where system bottlenecks exist. Our experimental results have shown that under data contention, the use of real-time oriented conflict resolution largely improves the performance of real-time transactions. This implies that rather than developing sophisticated replacement policies, it is more important to resolve data conflicts. Similarly, as we discussed for buffer allocation, I/O scheduling can be another more important processing component than buffer replacement.

7 Concluding Remarks

Buffer management plays an important role in traditional database systems. In this work, we explore this processing component for supporting real-time transactions. We have developed several buffer allocation and buffer replacement policies which take transaction timing constraints into account. The proposed allocation and replacement schemes are integrated to serve as a buffer management component and are implemented on our real-time database testbed.

The experimental results obtained from the testbed indicate that under two-phase locking, the real-time oriented buffer management schemes do not significantly improve system performance. With regard to buffer allocation, we have shown that data contention is a constraint on the performance improvement of buffer management. Under data contention,

conflict resolution becomes a key factor in real-time transaction processing. In addition, CPU scheduling is more important than buffer allocation, even if the system is not CPU bound. Concerning buffer replacement, we have seen that the complicated real-time oriented replacement algorithm performs no better than a simple LRU policy. Again, under data contention, it is the conflict resolution that significantly improves transaction performance. This study suggests that rather than developing sophisticated buffer management schemes, it is more important to improve the performance of other processing components, such as conflict resolution, CPU scheduling and I/O scheduling.

It is interesting to note that our result may appear to contradict to that in Carey et. al.'s work [1] where it is claimed that regardless of whether the system bottleneck is the CPU or the disk, priority scheduling of the critical resource must be complemented by a priority-based buffer management policy. These diverse conclusions result for the following reasons. First, admission control (i.e. the controlling of multi-programming level according to the buffer usage) is an important processing component of the buffer management in [1]. As we mentioned in Section 2, in [1], it is the admission control, not the allocation or replacement scheme, that leads to the performance gain. In this work we focus on the effect of buffer management. Second, only read transactions are exercised in [1], and thus the issue of data contention is not taken into account. On the other hand, our system includes both read and write transactions. As a result, data contention becomes a main influential factor for buffer performance. Third, as a part of a complete transaction processing system, recovery is taken into account in our buffer model. Since the recovery scheme requires memory space (a private buffer) for each individual transaction, we leave the intra-transaction locality to the private buffer management. Our work focuses on the global buffer management which captures restart-transaction locality as well as inter-transaction locality. In [1], on the other hand, there is no concern for recovery. They only consider a global buffer that mainly captures intra-transaction locality and inter-transaction locality.

Another important remark is that the results presented in this work are obtained from the system that employs two-phase locking mechanism for concurrency control. The conclusion that conflict resolution is a dominant factor is relevant to the locking approach used for concurrency control. The performance of real-time oriented buffer management may differ from one concurrency control mechanism to another. For example, under real-time optimistic concurrency control scheme [11], lock contention is no longer a problem. Rather, high transaction restart rate is a main concern. Under such a system, real-time oriented buffer management may play an important role in improving the performance of real-time transactions. As ongoing research, buffer management with optimistic concurrency control is under investigation.

Acknowledgements

We would like to thank Professor Krithi Ramamritham and Professor Don Towsley for their comments and suggestions on this work.

References

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

[2] Chen, S., J. Stankovic, J. Kurose, and D. Towsley, "Performance Evaluation of Two New Disk Scheduling Algorithms for Real-Time Systems," *Submitted for publication*, August, 1990.

[3] Dan, A., D.M. Dias and P.S. Yu, "Database Buffer Model for the Data Sharing Environment", *Proceedings of the 6th Data Engineering Conference*, LA, Feb. 1990.

[4] Dan, A., D.M. Dias and P.S. Yu, "An Approximate Analysis of the LRU and FIFO Buffer Replacement Schemes", *ACM SIGMETRICS*, May 1990.

[5] Dan, A., D.M. Dias and P.S. Yu, "Buffer Modelling for a Data Sharing Environment with Skewed Data Access", submitted to *IEEE Transaction on Computer Systems*, 1990.

[6] Dias, D.M., B.R. Iyer, J.T. Robinson and P.S. Yu, "Design and Analysis of Integrated Concurrency-Coherency Controls," *Proceedings of the 13th VLDB Conference*, Brighton, 1987.

[7] Dias, D.M., B.R. Iyer, J.T. Robinson and P.S. Yu, "Integrated Concurrency-Coherency Controls for Multisystem Data Sharing," *IEEE Transaction on Software Engineering*, Vol.15, No.4, April, 1989.

[8] Effelsberg, W. and Theo Haerder, "Principles of Database Buffer Management," *ACM Transactions on Database Systems*, Vol.9, No.4, Dec. 1984.

[9] Elhardt K. and R. Bayer, "A Database Cache for High Performance and Fast Restart in Database Systems," *ACM Transactions on Database Systems*, Vol.9, No.4, Dec. 1984.

[10] Huang, J., J.A. Stankovic, D. Towsley and K. Ramamritham, "Experimental Evaluation of Real-Time Transaction Processing," *Proceedings of the 10th Real-Time Systems Symposium*, Santa Monica, CA, Dec. 1989.

[11] Huang, J. and J.A. Stankovic, "Concurrency Control in Real-Time Database Systems: Optimistic Scheme vs. Two-Phase Locking Approach," *A Technical Report, COINS 90-66*, University of Massachusetts, July 1990.

[12] Kearns, J.P., and S. DeFazio, "Diversity in Database Reference Behavior," *Performance Evaluation Review*, Vol.17, No.1, May 1989.

[13] Kirk, D., "SMART (Strategic Memory Allocation for Real-Time) Cache Design," *Proceedings of the 10th Real-Time Systems Symposium*, Santa Monica, CA, Dec. 1989.

[14] Sacco, G.M. and M. Schkolnick, "Buffer Management in Relational Database Systems," *ACM Transaction on Database Systems*, Vol.11, No.4, Dec., 1986.

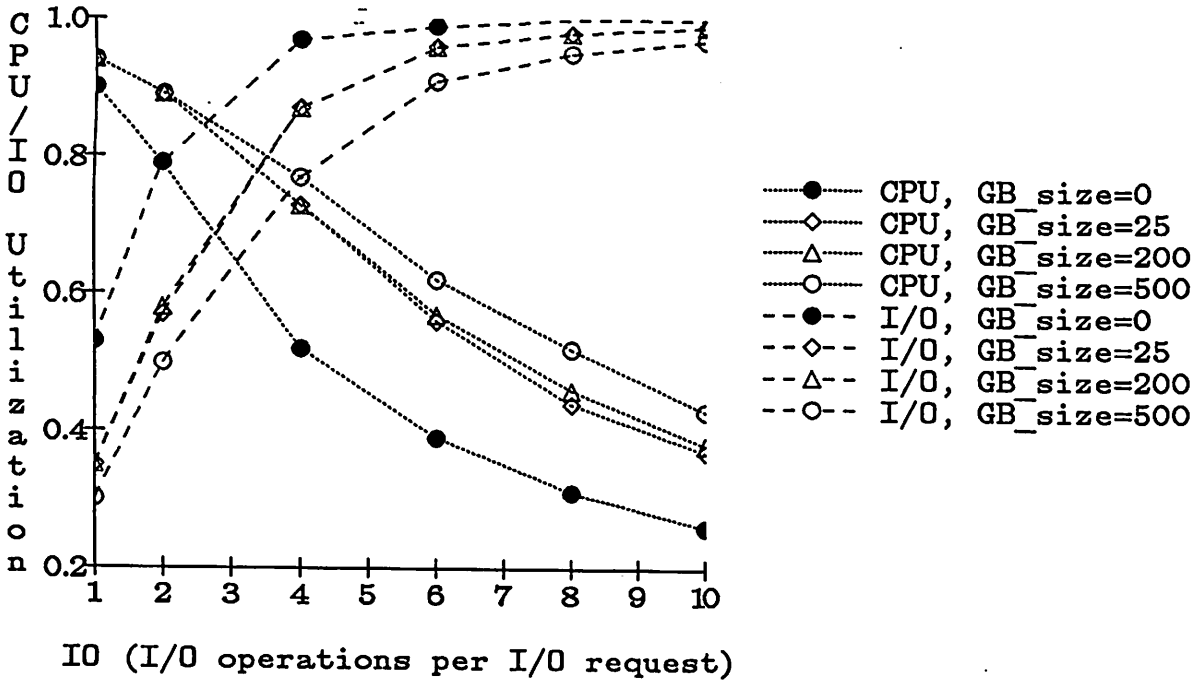

Fig. 2: System calibration, uniform access

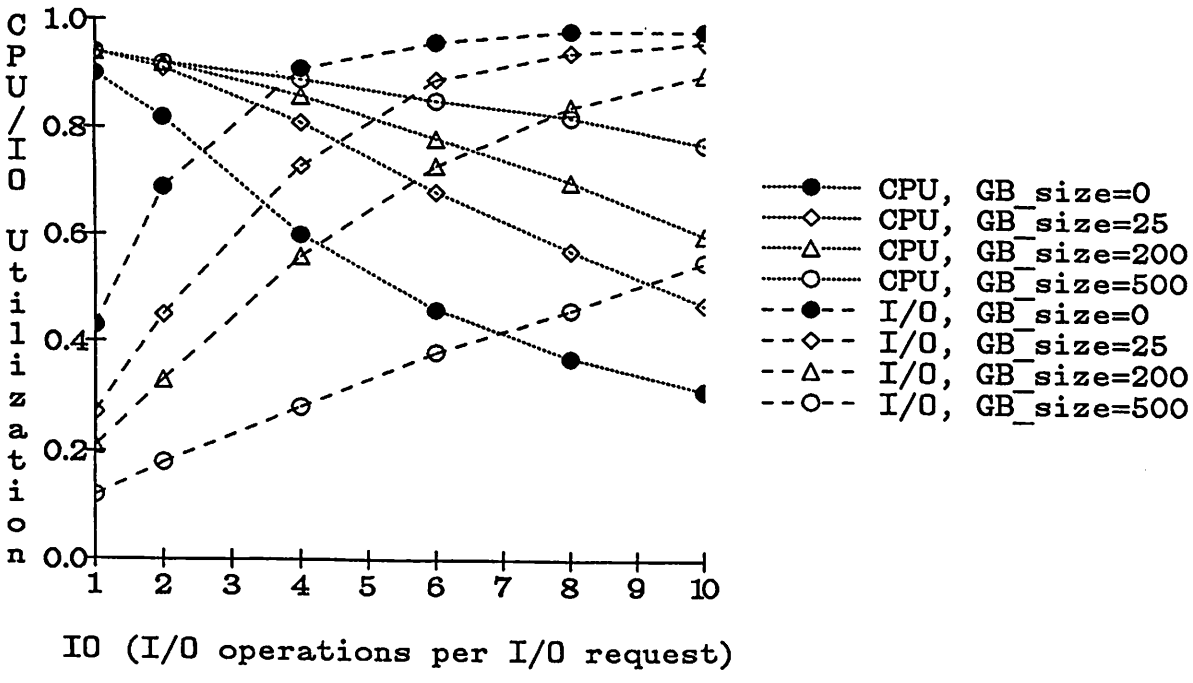

Fig. 3: System calibration, skewed access

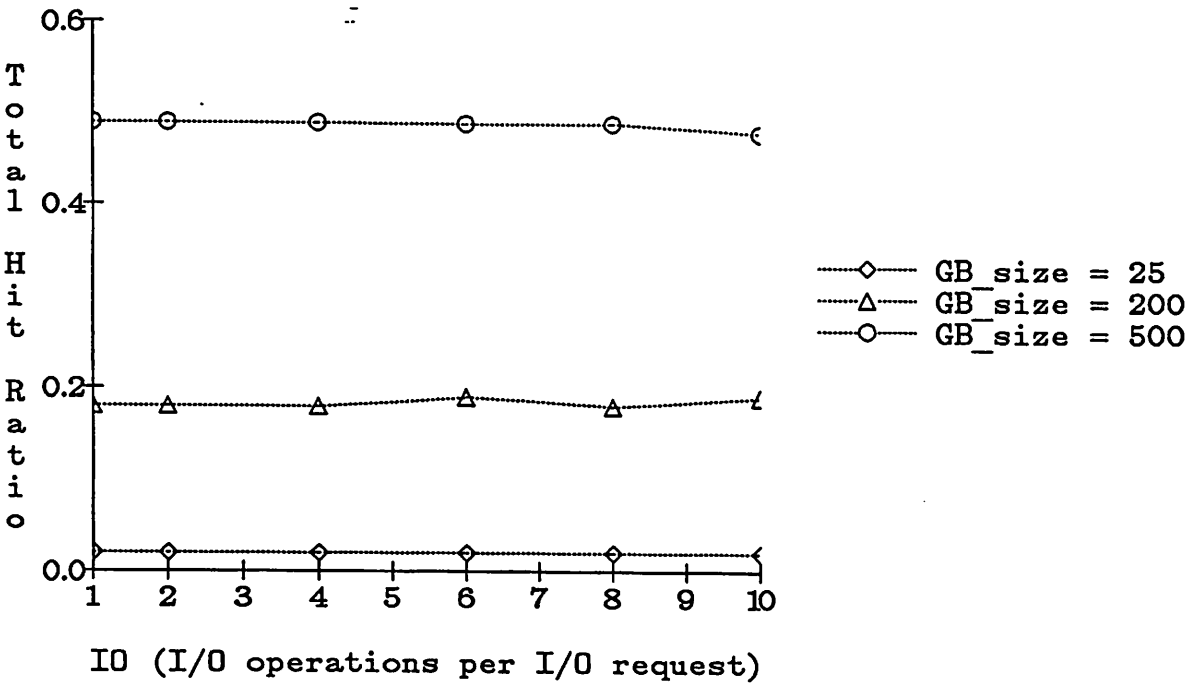

Fig. 4: System calibration, uniform access

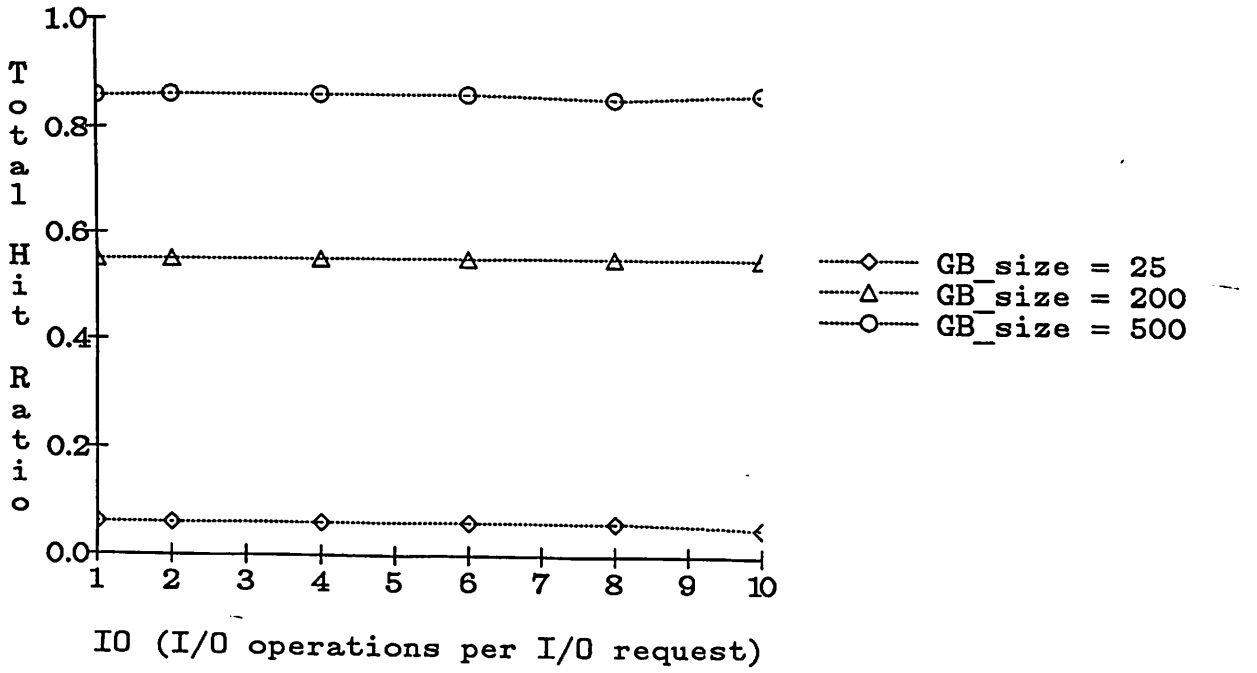

Fig. 5: System calibration, skewed access

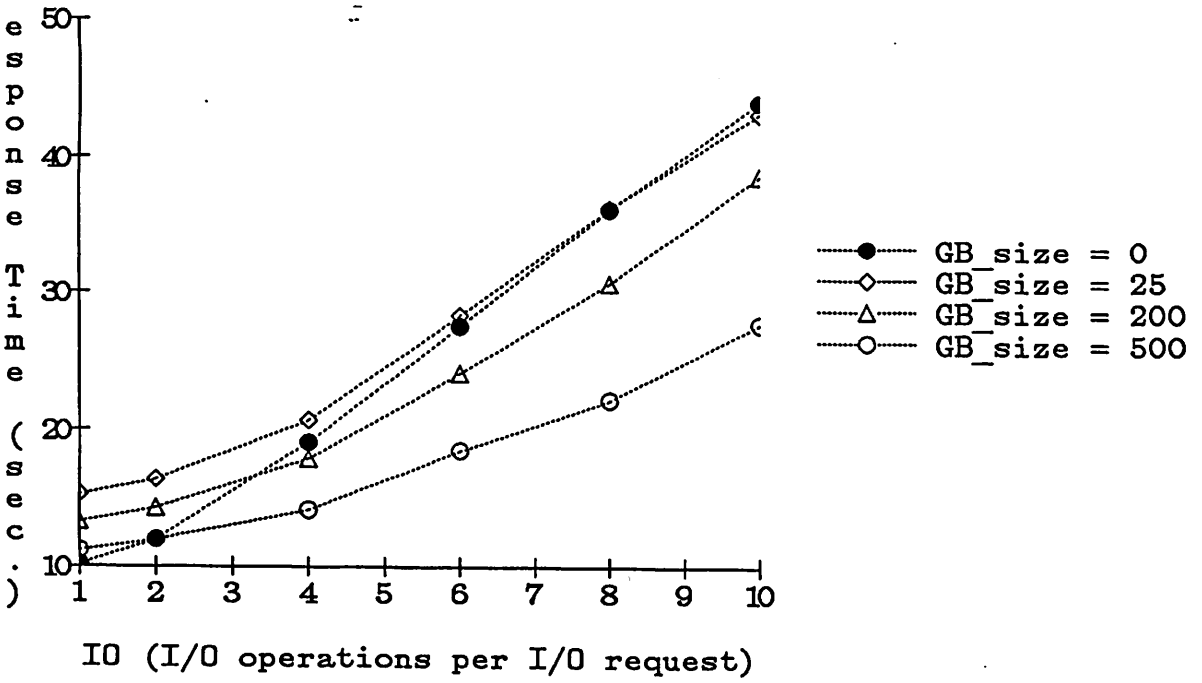

Fig. 6: System calibration, uniform access

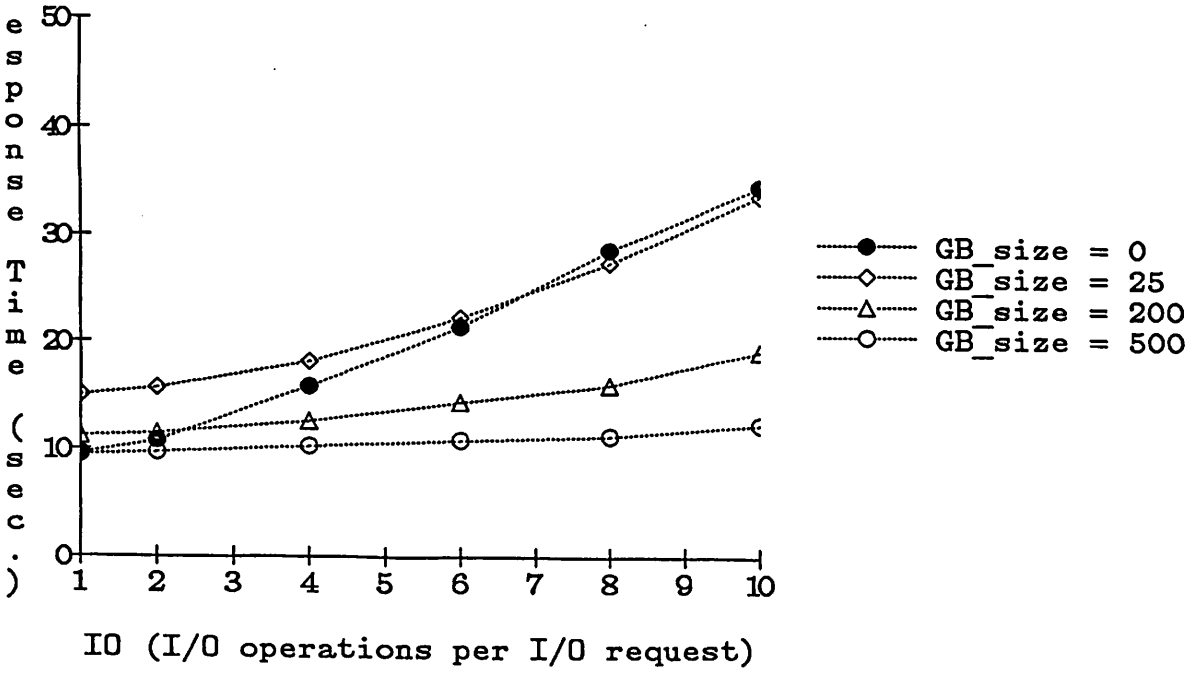

Fig. 7: System calibration, skewed access

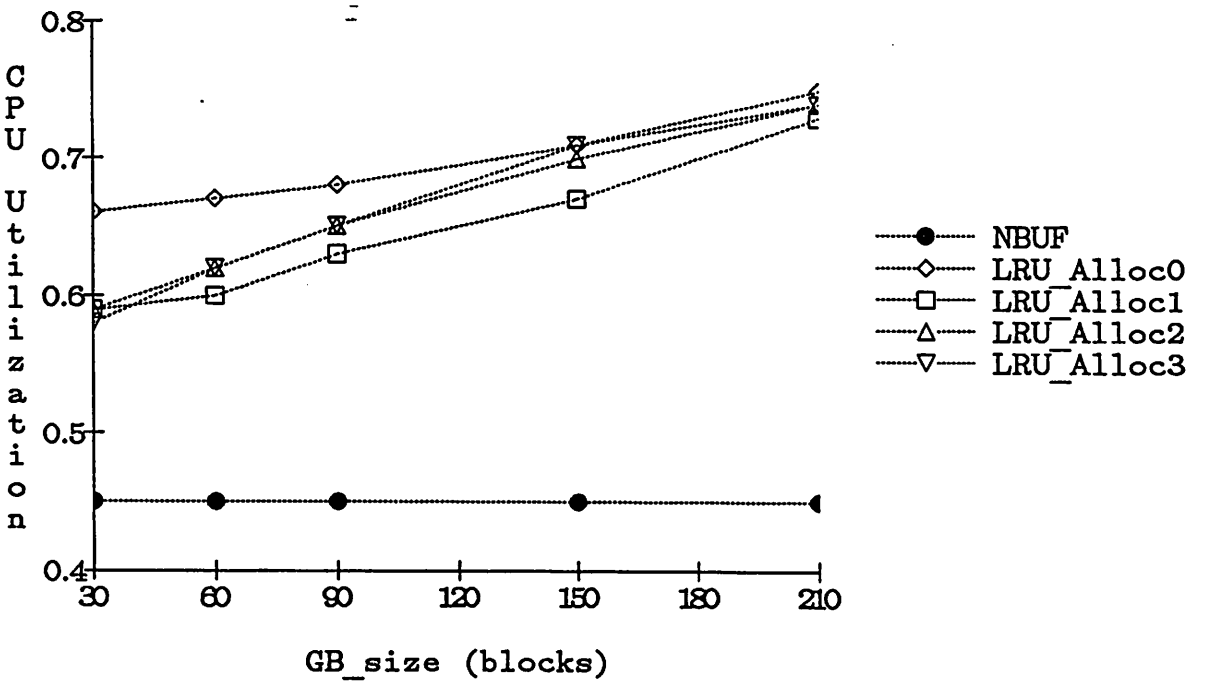

Fig. 8: Comparisons of allocation schemes

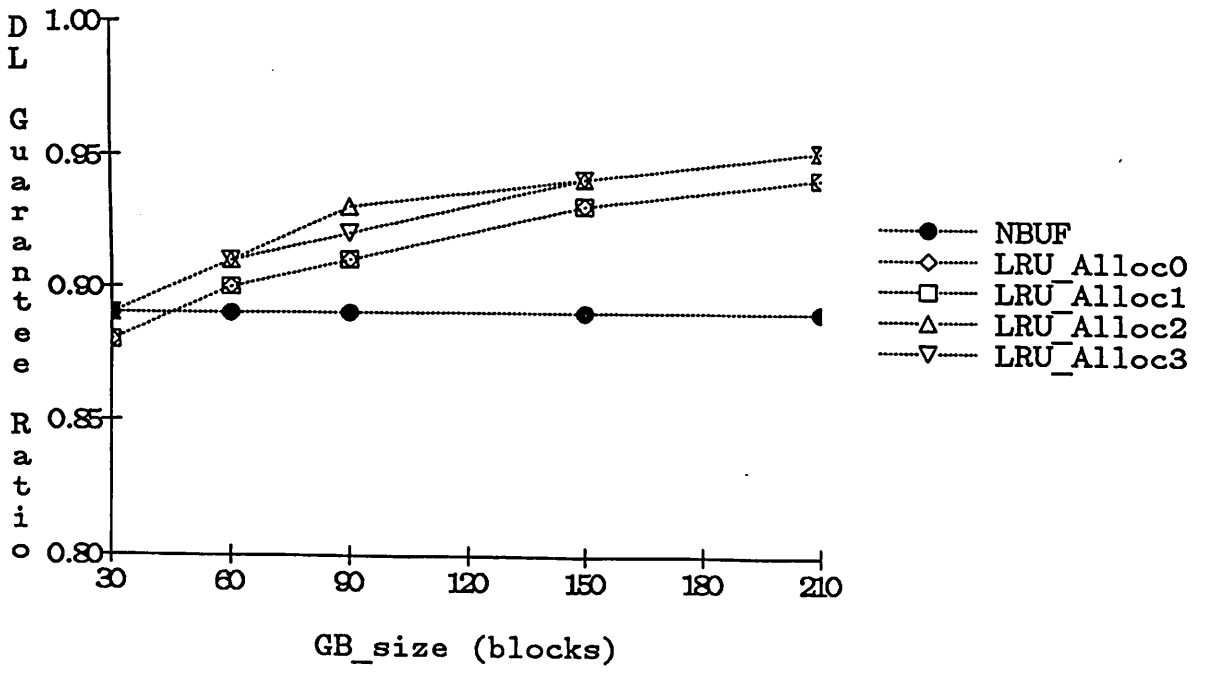

Fig. 9: Comparisons of allocation schemes

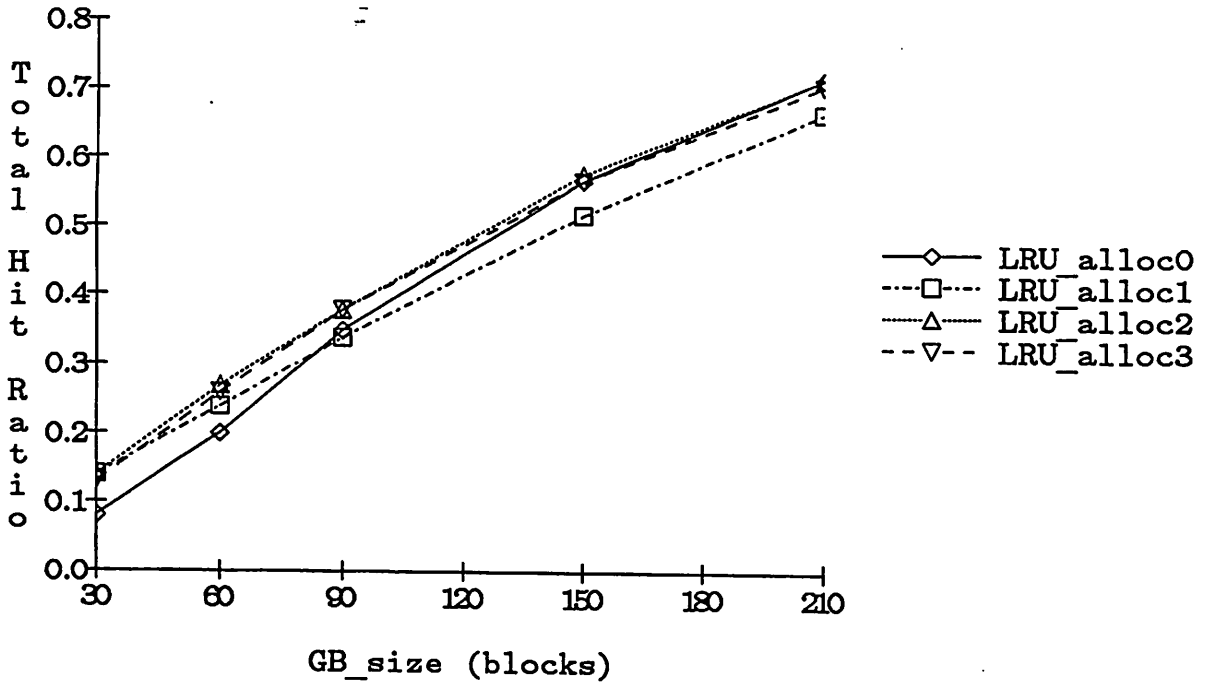

Fig. 10: Comparisons of allocation schemes

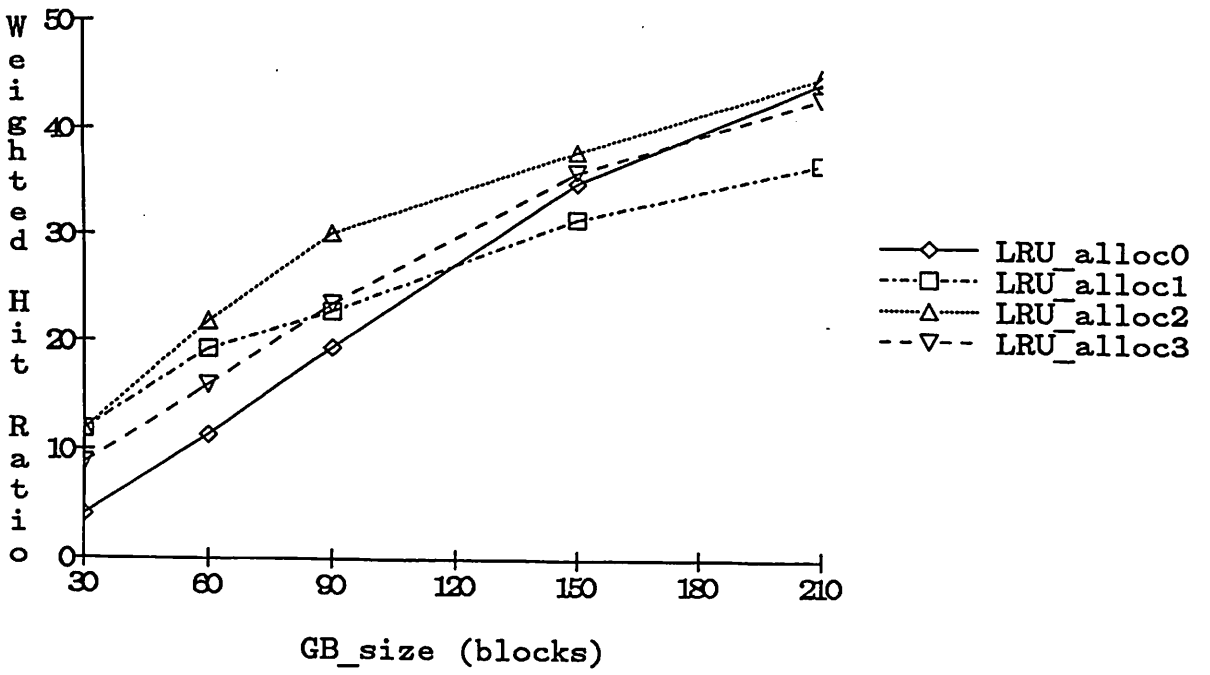

Fig. 11: Comparisons of allocation schemes

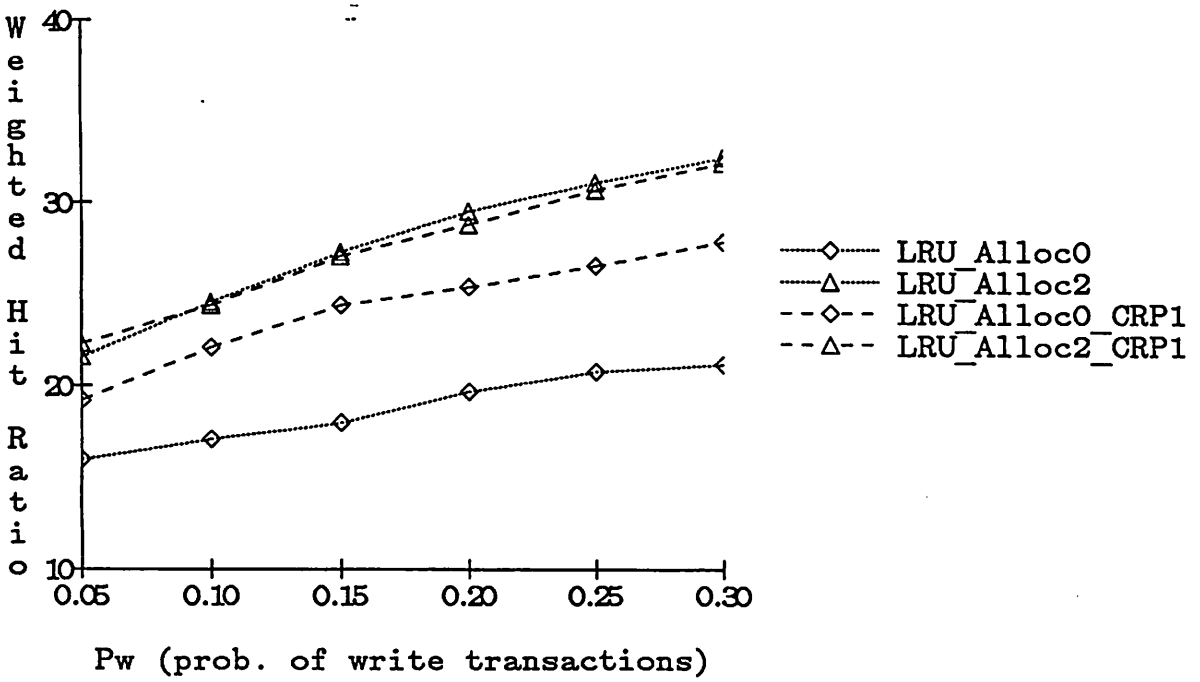

Fig. 12: Allocation vs. conflict resolution

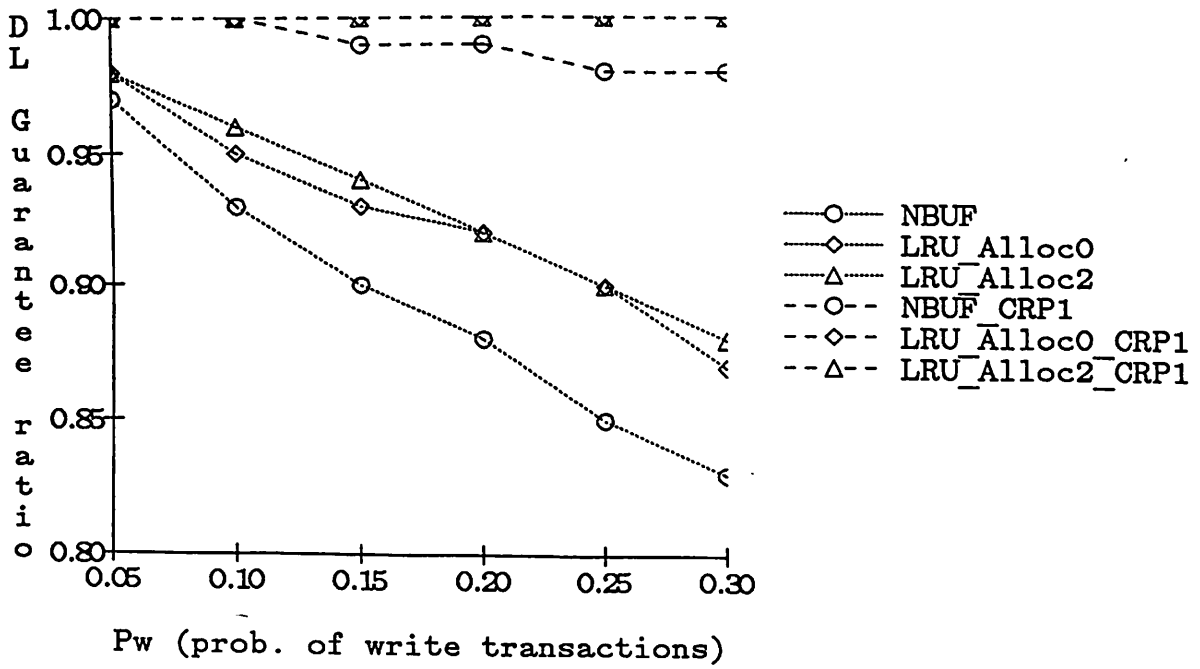

Fig. 13: Allocation vs. conflict resolution

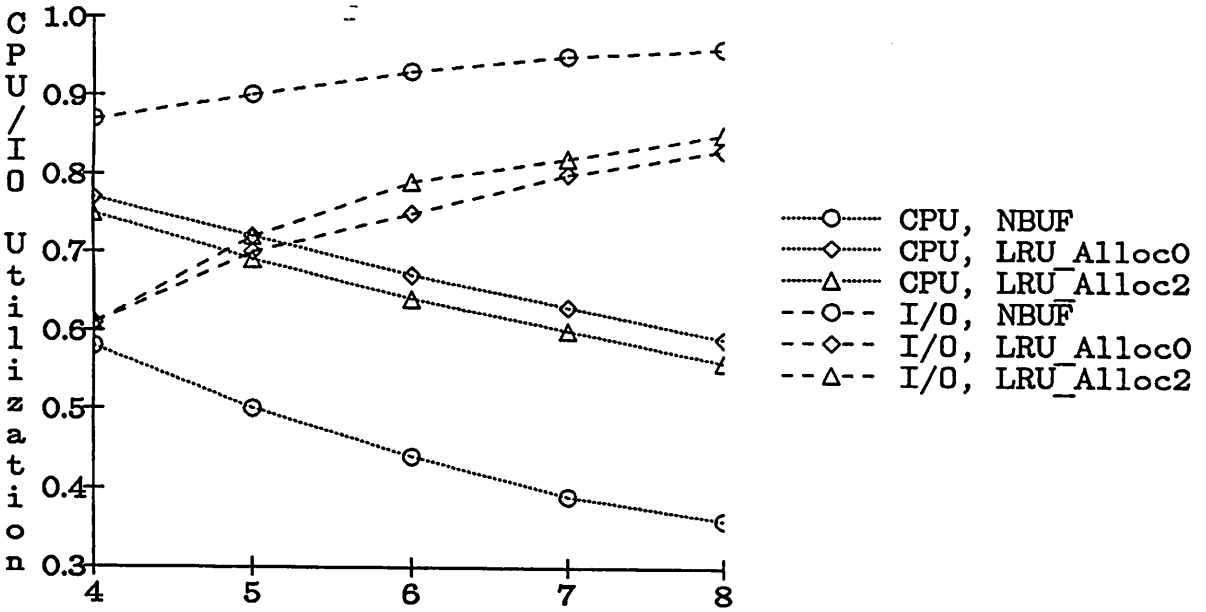

Fig. 14: Allocation vs. CPU scheduling

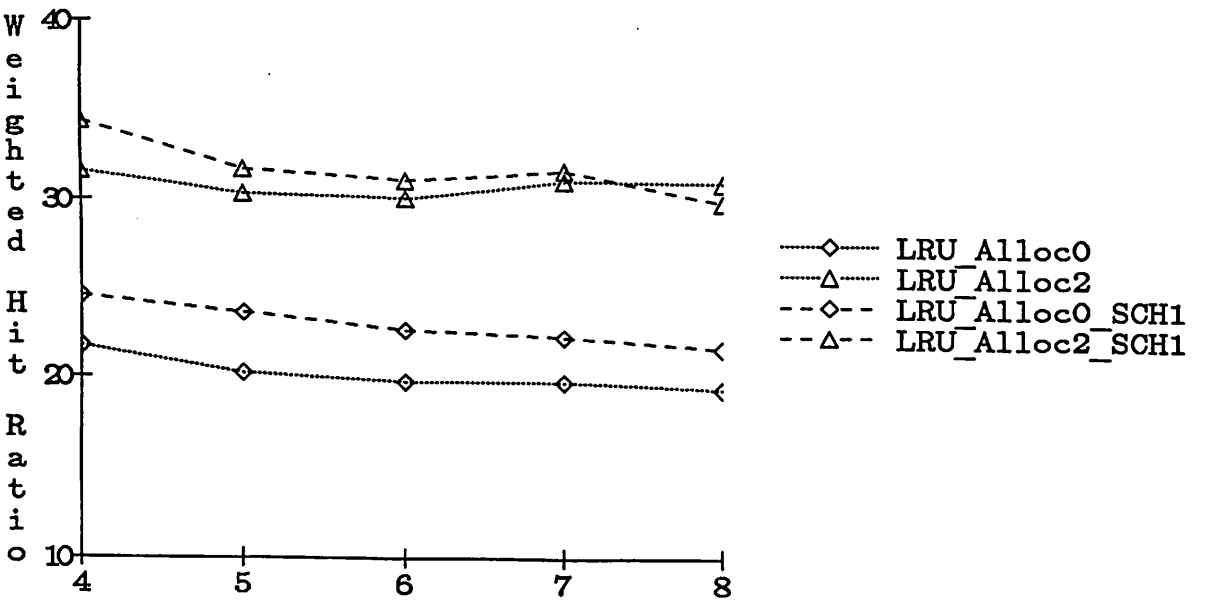

Fig. 15: Allocation vs. CPU scheduling

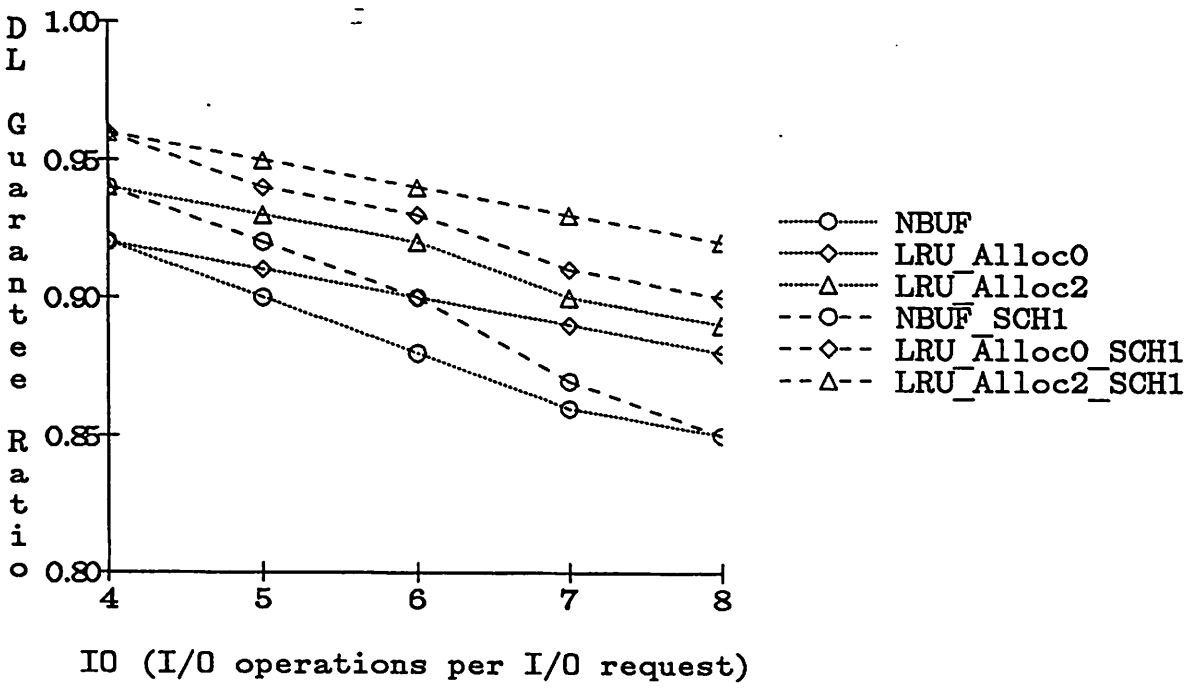

Fig. 16: Allocation vs. CPU scheduling

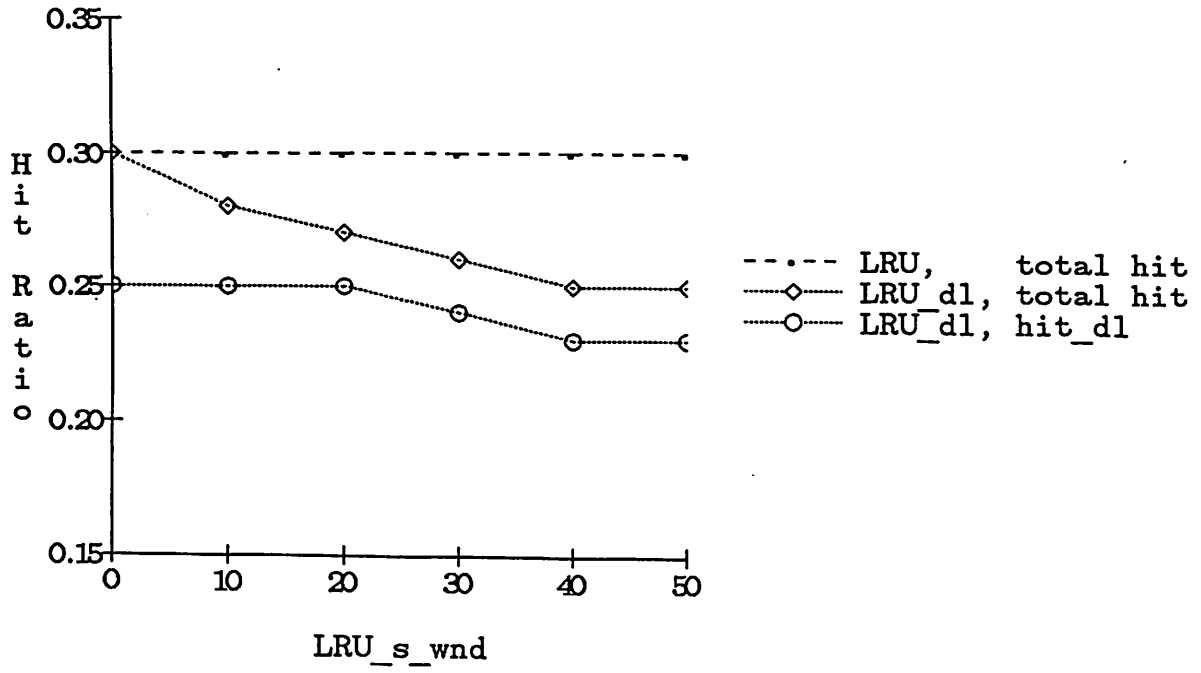

Fig. 17: Comparisons of replacement schemes

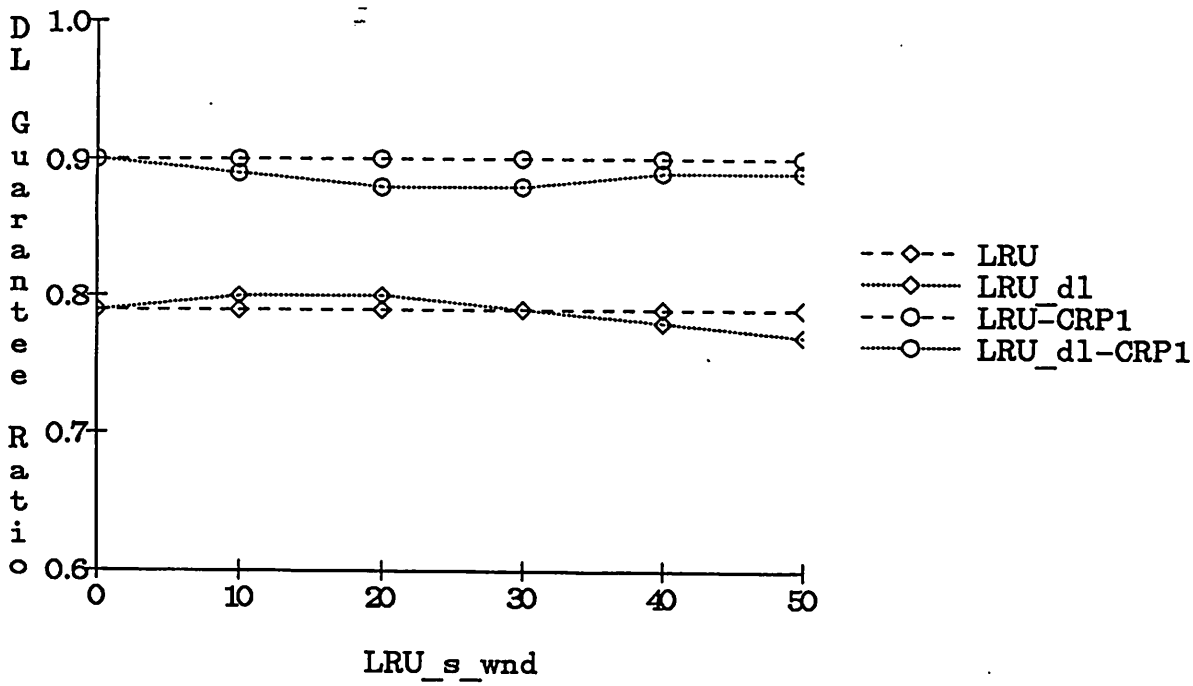

Fig. 18: Replacement vs. conflict resolution