Priority-Hints: An Algorithm for Priority-Based Buffer Management

by
Rajiv Jauhari
Michael J. Carey
Miron Livny

Computer Sciences Technical Report #911
February 1990

Priority-Hints: An Algorithm for Priority-Based Buffer Management

Rajiv Jauhari
Michael J. Carey
Miron Livny

Computer Sciences Department
University of Wisconsin
Madison, WI 53706

This research was supported in part by the National Science Foundation under grant IRI-8657323 and by the Digital Equipment Corporation through its Initiatives for Excellence program.

Priority-Hints: An Algorithm for Priority-Based Buffer Management

Rajiv Jauhari
Michael J. Carey
Miron Livny

Computer Sciences Department
University of Wisconsin
Madison, WI 53706

ABSTRACT

In this paper, we address the problem of buffer management in a DBMS when the workload consists of transactions of different priority levels. We present Priority-Hints, a new buffer management algorithm that uses hints provided by the DBMS access methods. The performance of Priority-Hints is compared to that of priority buffer management schemes introduced earlier for a variety of workloads. Our simulation results indicate that Priority-Hints performs consistently better than simple LRU-based algorithms. Furthermore, our algorithm approaches (and in some cases surpasses) the performance of highly sophisticated algorithms that require much more information to be provided to the buffer manager.

1. INTRODUCTION

1.1. Motivation

Priority scheduling has recently become an area of increased interest to the database community [SIGM88, Abbo88, Abbo89, Care89, Hari90]. Applications that require different levels of system response for different transactions (for example, a system that is designed to provide faster service to interactive jobs than to batch jobs) can benefit from priority scheduling at the DBMS resources, as shown in [Care89]. Several data-intensive applications such as computer-aided manufacturing, stock trading, and command and control systems may require real-time response, which can also be supported with the help of priority scheduling at the resources of the DBMS [SIGM88, Abbo89].

The use of priority in DBMS resource scheduling may lead to an increase in the extent to which buffer management impacts system performance compared to its impact in conventional database systems. Unpredictable bursty arrivals of high-priority transactions may force a priority-oriented DBMS to operate in regions where the total load on the buffer pool (i.e., the sum of the buffering requirements of transactions of all priority levels) exceeds the buffer pool capacity. In these operating regions, priority-based load control and buffer allocation policies will be required, as the use of conventional load control and allocation techniques may lead to situations of "priority-inversion," where high-priority transactions are forced to wait while low-priority transactions are allowed to make progress. Furthermore, the set of concurrently active transactions in these operating regions may include transactions of different priority levels. In this scenario, then, priority-based buffer replacement policies may also be required in order to provide preferential service to high-priority transactions. We

This research was supported in part by the National Science Foundation under grant IRI-8657323 and by the Digital Equipment Corporation through its Initiatives for Excellence program.

anticipate that all aspects of buffer management (load control, allocation, and replacement) will become both more complex and more significant when priority is used in scheduling DBMS resources.[1]

Several interesting new issues arise when buffer management decisions have to include priority considerations. One such issue is the tradeoff between the overheads introduced as a consequence of the use of priority and the advantages provided to high-priority transactions. If a buffer containing data accessed by a transaction is replaced as a consequence of priority, its data may have to be re-read from disk once the transaction resumes execution. The total load on the system may therefore increase purely as a consequence of the use of priority in buffer management, and alternative priority-based buffer replacement and allocation strategies may result in different relative increases in system load.

A second issue of interest is the extent to which information about the workload can be used by the buffer manager to improve system performance in the presence of priority. Existing buffer management schemes assume different levels of information about transactions' data access patterns [Effe84, Teng84, Chou85, Sacc86]. In this paper, we introduce a new buffer management algorithm that makes use of hints provided by the database access methods (as in the Starburst buffer manager [Haas90] and DB2 [Teng84]). This new algorithm, called "Priority-Hints," uses these hints to make priority-based buffer management decisions while trying to minimize the priority-induced overhead on the system.

A third issue in priority-based buffer management is inter-transaction buffering interference across priority levels. For example, update-intensive transactions may quickly make large numbers of buffers "dirty," making them unavailable for replacement until they are written out to disk. The performance of high-priority transactions can thus be affected adversely by low-priority updates. Another example of inter-transaction effects in the presence of priority may occur when high-priority sequential scans quickly replace a large number of buffers with pages that are accessed just once, unnecessarily depriving lower priority non-sequential transactions of buffers that need to be accessed repeatedly. Priority-based buffer management policies should be designed to minimize these effects.

Finally, the importance of using priority-based buffer management in a DBMS that already uses priority at the CPUs and the disks may itself be open to question. In this paper, we investigate all of these issues.

1.2. Related Work

A number of buffer management strategies for conventional database systems have been proposed in the literature [Effe84, Chou85, Sacc86]. Simple techniques such as Global-LRU assume no knowledge of the data access patterns, while algorithms such as Hot Set [Sacc86] and DBMIN [Chou85] attempt to take advantage of the limited number of ways in which queries access data in relational database systems. In [Chou85], it was shown that DBMIN, a relatively sophisticated buffer management strategy in which data access pattern information is supplied to the buffer manager on a per-file basis by the DBMS optimizer, performs better than most other existing buffer management strategies.

[1]This is in contrast to the trends in conventional database systems, where it may be argued that by increasing the buffer pool size with respect to the database size, and by keeping the multiprogramming level under a certain threshold, buffer management policies can be made essentially irrelevant.

In [Teng84], the design of IBM's DB2 buffer manager is described, and several techniques used to maximize buffer pool performance are discussed. For example, DB2's buffer manager distinguishes between sequential accesses and random accesses, and buffered pages that are part of sequential accesses are chosen as replacement victims in preference to randomly accessed pages. The LRU replacement policy is used within each group of buffers (sequential or random). Other performance-enhancing techniques used in DB2 include the use of prefetch for sequential scans and the use of deferred-write mechanisms for writing updated pages to disk. In the Starburst system, as described briefly in [Haas90], access methods can provide hints to the buffer manager about their expected future re-use of each buffered page in order to guide replacement decisions.

An initial investigation of the problem of priority scheduling at the physical resources of a DBMS was described in [Care89]. Algorithms for priority-based disk scheduling and CPU scheduling were presented, and priority-oriented modifications of two existing buffer management algorithms (Global-LRU and DBMIN) were described. It was shown that a single priority scheduler does not suffice to meet the goals of priority scheduling in a DBMS due to the heterogeneity and the multiplicity of its resources. Priority scheduling at the admission control component of the DBMS (the part of the system that determines whether an arriving transaction is to be allowed to enter the system, or is to be blocked outside the system until sufficient buffer resources become available) was shown to be particularly important in order to keep the system stable from the point of view of high-priority transactions as system load is increased. A homogeneous workload model was used to compare the performance of the priority-scheduling algorithms, and Priority-DBMIN (the priority version of DBMIN) was shown to perform as well as or better than Priority-LRU (the priority version of Global LRU).

In [Abbo89], priority scheduling at the CPU and the disk, as well as for concurrency control, was shown to be effective in reducing the number of transactions that miss their deadlines in a real-time DBMS. Transactions were modeled as random sequences of page accesses, and the buffer pool was modeled simply by computing the probability of finding a page in main memory assuming a uniform probability of access to the entire database. Finally, it was shown in [Hari90] that in a real-time environment, priority scheduling at the CPU and the disks may help optimistic concurrency control algorithms outperform locking schemes. Issues of buffer replacement strategies and workloads with non-random data access patterns were not included in either [Abbo89] or [Hari90].

1.3. Our Work

The design of the Priority-Hints algorithm was motivated by the intuition that the use of simple page-level information by the buffer manager (as in [Teng84, Haas90]) may improve system performance over that provided by simple LRU-based approaches. Such a performance improvement acquires great importance in priority-based systems, where significant I/O overheads may result as a consequence of using priority in buffer management decisions. Improvements in performance (relative to simple LRU-based approaches) may also be obtained by using sophisticated DBMIN-like algorithms, as shown in [Care89], but only at the cost of added system complexity; our goal is to examine whether such complexity is really required.

In order to take a detailed look at the issues involved in priority-based buffer management in the presence of mixed workloads, we have implemented a simulation model of a DBMS that uses priority scheduling. In this paper, we use this simulation model to compare the performance of our new algorithm with Priority-LRU and Priority-DBMIN. Our primary objective in this analysis is to explore the extent to which our algorithm can surpass the performance of Priority-LRU, and how close we can get to the performance of Priority-DBMIN. We also conduct experiments that shed light on the other priority-related buffer management issues raised earlier.

The remainder of this paper is organized as follows: In Section 2, we describe Priority-Hints, our new buffer management algorithm, and review the Priority-LRU and Priority-DBMIN policies introduced in [Care89]. Section 3 describes our simulation model of a priority-oriented DBMS. Section 4 presents a series of performance experiments that help us to understand the tradeoffs involved in using priority in buffer management. Finally, Section 5 summarizes our contributions and describes our plans for future work.

2. PRIORITY-BASED BUFFER MANAGEMENT ALGORITHMS

In this section we present the Priority-Hints algorithm. Our assumptions about buffer management are outlined first. We then describe Priority-Hints and briefly review Priority-LRU and Priority-DBMIN, the two algorithms presented in [Care89]. Our scheme for handling dirty data, which is common to the three algorithms, is described next; and we conclude the section with a summary of the key differences between the three algorithms.

2.1. Buffer Management Assumptions

A page is assumed to be fixed (or pinned) in the buffer pool during the interval when a transaction is processing the data on the page. As soon as the transaction has finished processing the page, it unfixes it. Fixed pages cannot be chosen as buffer replacement victims. The *owner* of a resident page is the transaction with the highest priority among the executing transactions that have accessed the page since it was brought into memory. The buffer manager associates a *timestamp* with each resident page in order to keep track of the recency of usage of pages. Each time the data in a buffer is accessed or updated, a global counter is incremented and its new value is inserted as the timestamp of the page. Thus, the larger the value of the timestamp of a page, the more recently the page was accessed. Pages in the free list (and the dirty list, described at the end of this section) are kept in LRU order using their timestamps.

Based on the number of buffers available, a transaction may be admitted to the system right away, or it may be blocked initially. Transactions blocked outside the system are queued in order of priority. Once a transaction is allowed to begin execution, it continues until it commits, is aborted as a result of concurrency control, or is *suspended*.[2] A transaction is said to be *suspended* by the buffer manager if it is temporarily prohibited from making further buffer requests; the buffers

[2]A running transaction may also be blocked temporarily if there are no free buffers available, and all of the in-use buffers are either pinned or dirty. The transaction's buffer request is then enqueued in a queue called the *Buffer Waiting Queue*. Queued buffer requests are served in priority order.

owned by the transaction are freed. The buffer manager considers *reactivating* suspended transactions at the same decision point that it considers admitting blocked transactions, which is whenever a running transaction completes or aborts. A reactivated transaction resumes its execution at the point where it was suspended.

A transaction checks whether it has been chosen as a suspension victim at instants when it has no pages fixed. We call such instants "suspension-safe" points. At suspension-safe points, the transaction "volunteers" to let all of its buffers be stolen by transactions of higher priority. Such instants occur normally during the course of executing a transaction. In the case of a sequential scan, for example, they occur at the point when the transaction unfixes one page and is about to request that another be fixed. In addition, a priority DBMS should be designed so that transactions periodically "come up for air" and check if they need to give up their buffers.

2.2. Priority-Hints: A New Buffer Management Policy

As its name suggests, Priority-Hints makes use of hints (provided by the DBMS access methods) that indicate whether a particular data page should be retained in memory in preference to other data pages. The basic ideas underlying Priority-Hints are the following:

(1) As discussed in [Teng84, Haas90], it is possible to classify the pages referenced by a transaction into two groups: pages that are likely to be re-referenced by the same transaction (such as the pages of the inner file in a nested-loops join), and pages that are likely to be referenced just once (such as the pages of a file being scanned sequentially). The pages that are likely to be re-referenced are called *favored* pages, and the others are called *normal* pages. We assume that whenever a request for a page is made to the buffer manager, the buffer manager is informed whether the requested page is favored or normal.

(2) The favored pages of a transaction should be kept in the buffer pool as long as the transaction needs to reaccess them; each normal page should be made available for replacement as soon as the transaction unfixes it. When searching for replacement victims, normal pages should therefore be considered before favored pages.

(3) If it becomes necessary to choose a favored page as a replacement victim, the most-recently-used (MRU) policy should be used to choose the victim. As discussed in [Chou85], MRU is a better approach than LRU when choosing replacement victims from a set of pages that are being repeatedly looped over, and favored pages are likely to fall into this category.

The Priority-Hints algorithm combines these ideas with the notion of priority as follows.

Buffer Pool Organization

Buffers are organized into "transaction sets," where a transaction set consists of all of the buffers owned by a single transaction.[3] Transaction sets are arranged in priority order, with recency of arrival of the owner transaction being used to break ties if there are multiple transactions of the same priority. In the buffer pool configuration shown in Figure 2.1, there are four transactions, three priority levels, and no free buffers. Priority decreases from the top to the bottom of the figure. Transaction T4 is at priority 3, T1 and T3 are at priority 2, and T2 is at priority 1. T1 arrived before T3, so the sequence in which the transaction sets are arranged is {T4,T1,T3,T2}.

A transaction set consists of two kinds of buffers: the buffers currently fixed by the owner (marked by the letter "F" in Figure 2.1), and buffers containing unfixed favored pages of the owner (marked by the letter "U" in Figure 2.1). The unfixed favored pages are maintained in MRU order with the help of buffer timestamps. Note that a transaction set contains no unfixed normal pages; whenever a normal page is unfixed, it is freed.

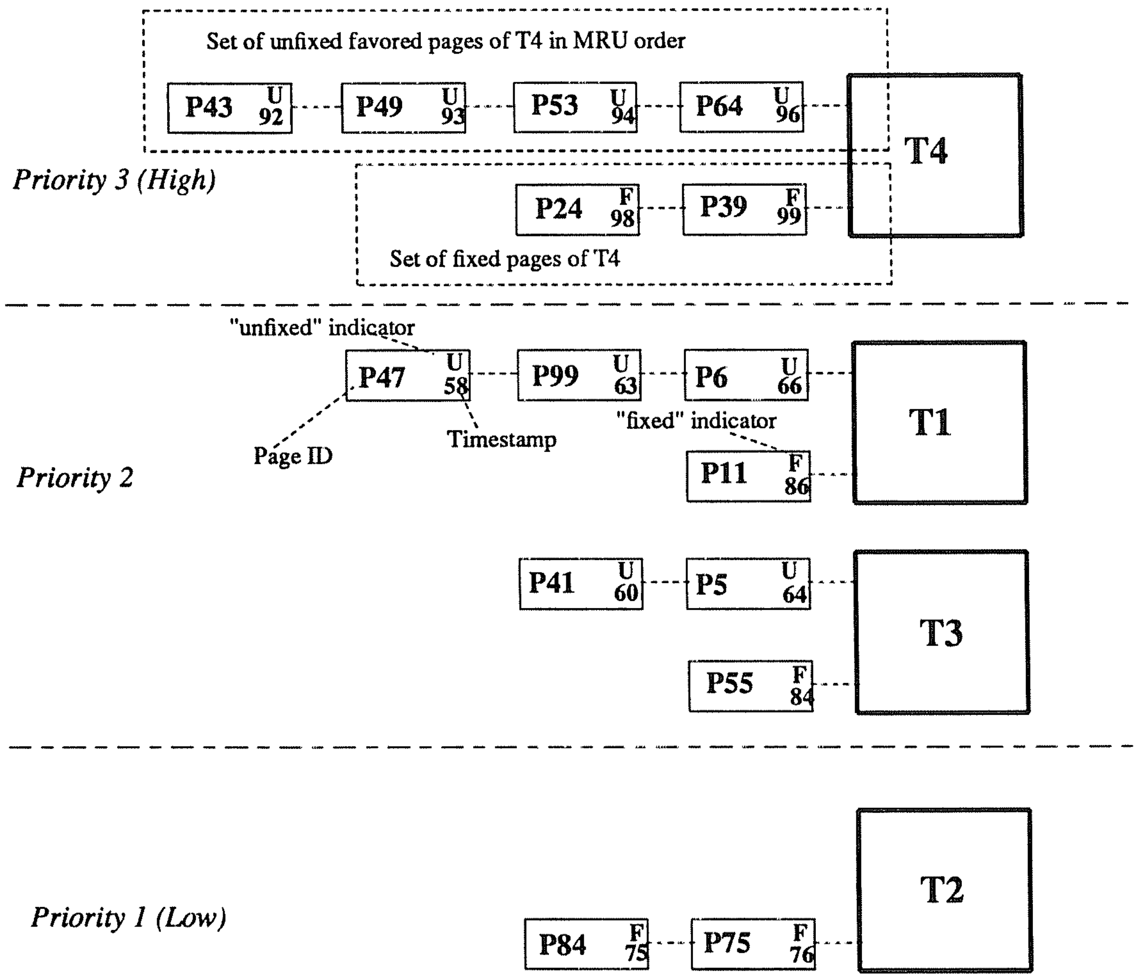

Figure 2.1: Example of Priority-Hints Buffer Pool Organization.

[3]Buffers containing pages shared by more than one transaction are owned by the transaction with the highest priority among the sharers.

Transaction Admission

Transactions are required to estimate the maximum number of pages that they will need to fix concurrently, and the buffer manager keeps track of the sum of these "fixing requirements" for all active transactions. If admitting a newly arrived transaction does not cause this sum to exceed the size of the buffer pool, the transaction is admitted. Otherwise, if there are running transactions of lower priority than that of the new arrival, the one(s) with the lowest priority among them are suspended until there are enough buffers for the new arrival, or until no lower priority transactions remain.[4] If no remaining transactions are of a priority less than the new arrival, then the new arrival is forced to wait outside the DBMS.

Buffer Replacement and Allocation

When a buffer miss occurs and there is no free page available, the buffer manager first attempts to get a replacement victim from among the unfixed favored pages of transactions of lower priority than the requesting transaction. The buffer pool searches its transaction sets in inverse priority order, starting from the lowest priority transaction, looking for unfixed favored pages. It stops searching on either of the following conditions:

(1) It finds a transaction of lower priority than the requesting transaction with an unfixed favored page; or

(2) it has reached a transaction of a priority equal to or greater than that of the requesting transaction.

In case (1), it chooses the most recently unfixed favored page of the lower-priority transaction as the replacement victim. In case (2), it chooses the most recently unfixed favored page (if any) of the *requesting transaction* itself. Note that this means that transactions cannot steal buffers from other transactions of the same priority; thus the replacement policy for favored pages is *local* rather than *global*. If no replacement victim is available, then the outstanding request is queued in the Buffer Waiting Queue. Furthermore, if there are running transactions of lower priority, the transaction with the lowest priority among them is suspended. Continuing the example of Figure 2.1, if T4 makes a buffer request for page P3, which is not in the buffer pool, the buffer manager will start its search for replacement at T2. Finding no unfixed buffer in T2's transaction set, it will look at T3's transaction set and find P5 as the replacement victim. Had there been no unfixed pages of priority 1 or 2, then P64 would have been chosen as the replacement victim.

To summarize, Priority-Hints has a *local MRU* replacement policy for favored pages, and a *global LRU* replacement policy for unfavored pages. (Recall that unfavored pages are placed on the free list at unfix time, and that the free list is maintained in LRU order.)

[4]In choosing suspension victims from among transactions of the same priority, later arrivals are chosen for suspension in preference to earlier arrivals. Also, earliest arrival time is the criterion for choosing the transaction (from among a group of waiting transactions of the same priority) that should first attempt to enter the system.

2.3. Priority-LRU: A Global LRU-Based Buffer Management Policy

In Priority-LRU, the prioritized version of Global LRU, the buffer pool is organized dynamically into priority levels as described in [Care89]. Each priority level consists of pages whose owners have the same priority, and the pages within a level are arranged in LRU order. The transaction admission policy for Priority-LRU is the same as that for Priority-Hints. The key idea of the Priority-LRU replacement policy is that the least recently unfixed page of the lowest priority should be chosen as the victim. If there are no free buffers, the search for a replacement victim starts at the lowest priority queue, where we check whether unfixed candidates are available. If such candidates are found, the least-recently-used candidate is chosen as the victim. If no candidate is found at this priority level, we move up one level, and we repeat the process until we have either found a victim, reached a priority level that exceeds that of the requesting transaction, or exhausted the search. If no victim is found, and there are transactions of lower priority running, the lowest-priority transaction is suspended as in Priority-Hints.

2.4. Priority-DBMIN: A DBMIN-Based Buffer Management Policy

As discussed in [Chou85], the primitive operations (e.g., selections, joins) of transactions in a relational DBMS can be described as a composition of a set of regular reference patterns such as sequential scans and hierarchical index lookups. These patterns are known to the query optimizer. The DBMIN buffer management policy makes use of this information in the following way: A set of buffers (called a "locality set") is allocated to each transaction for each file accessed by it. The optimum size of each locality set and the optimum replacement policy to be used within a locality set are supplied to the buffer manager by the optimizer. DBMIN guarantees that each transaction that is allowed to enter the system has the optimum number of buffers available to it, and the optimum replacement policy is used within each locality set.

Priority-DBMIN, the prioritized version of DBMIN, also allocates buffers to transactions in locality sets. A transaction is allowed to enter the system only if its optimal-sized locality sets can be accommodated in the buffer pool. Otherwise, if there are transactions of lower priority than that of the arriving transaction in the system, they are suspended in reverse priority order until sufficient buffers become available for the new arrival. As in the original DBMIN algorithm, Priority-DBMIN uses the optimizer-supplied optimum replacement policy within each locality set.

2.5. Dirty Data

Our scheme for handling dirty data is common to all three algorithms. When a transaction frees a buffer, the buffer is inserted into the free list if it is clean (i.e., if it has no update that has not been written to disk). If the data in the buffer has been updated, the buffer is placed in a queue called the dirty list. A process called the *asynchronous write engine* [Teng84] is responsible for flushing dirty buffers to disk. This write engine can be awakened in one of two ways. First, the engine is activated whenever a transaction cannot find a free buffer (i.e., whenever there is a buffer pool miss and the free list is empty). The engine will then flush each dirty page in the dirty list that is sufficiently "old" in terms of its recency of use: The timestamp of each dirty page is compared to the global timestamp maintained by the buffer manager, and the page is written out to disk if the difference is greater than *FlushThreshold*.[5] The second way that the write engine gets activated is that it periodically wakes up by itself. The same recency-of-use criterion is used to determine whether any dirty pages should be flushed to disk in this case.

Requests to write dirty buffers are asynchronous; that is, the buffer manager does not wait for the I/O to be completed before continuing its activities. This may result in some buffer requests having to wait until a buffer is flushed to disk. Write requests to the disk are therefore assigned a priority equal to the highest possible transaction priority. When its I/O is completed, a dirty buffer is marked clean and placed on the free list, and if there are any buffer requests pending, the waiting request with the highest priority is serviced. When choosing replacement victims, dirty data is avoided as long as possible. That is, if a buffer that would normally be a candidate for replacement is dirty, we ignore it in our search for replacement victims unless all candidate buffers are dirty.[6]

2.6. Discussion

In summary, the key features of Priority-Hints that distinguish it from the other algorithms discussed are the following:

(1) By realizing which pages are normal, Priority-Hints is able to free more buffers earlier in the course of a transaction's execution than Priority-LRU. In this respect, Priority-Hints behaves similarly to Priority-DBMIN.

(2) When choosing replacement victims from among non-free pages, Priority-LRU chooses the least-recently-unfixed page; Priority-Hints chooses the most-recently-unfixed page. MRU is likely to be a better policy when the replacement victim is part of a set of pages that are being looped over; in cases where pages are reaccessed randomly, the performance differences between MRU and any other replacement policy are negligible [Chou85]. Note that Priority-Hints uses MRU only for favored pages, where it may be advantageous to do so; LRU is still used for normal pages, since normal pages are placed in LRU order in the free list as soon as they are unfixed.

(3) Priority-Hints' replacement policy ensures that the favored pages of a transaction can be stolen only by a transaction of higher priority: in Priority-LRU, transactions of the same priority can steal each other's pages.[7] In Priority-DBMIN, in contrast, a transaction protects its favored pages throughout its execution; if it is not possible to protect them, the transaction is suspended. Note that Priority-Hints allows a transaction to execute even when it does not have an optimum number of favored pages, while Priority-DBMIN does not.

(4) Priority-LRU does not discriminate between transactions of the same priority when choosing replacement victims. The local replacement search strategy of Priority-Hints, however, ensures that among transactions of the same priority, all but the latest arrival will execute undisturbed as long as the latest arrival has some unfixed favored

[5]The reason that the engine checks whether each page is old enough to flush, rather than flushing all dirty pages, is to prevent unnecessary writes of pages that are frequently updated.

[6]Potential deadlocks caused by the entire set of possible replacement candidates being dirty are avoided by synchronously writing dirty buffers to disk in this exceptional situation.

[7]It is advisable to allow Priority-LRU to do this, as many of the pages owned by a transaction are likely to be accessed just once.

buffers. Thus, the performance degradation caused by stealing favored buffers is limited to one transaction at a time in Priority-Hints.

(5) Priority-Hints does not require that information such as optimum locality set sizes be provided by the optimizer, as Priority-DBMIN does. It merely requires that hints be provided to distinguish between normal and favored pages; similar hints are provided in existing DBMSs such as DB2 [Teng84] and Starburst [Haas90]. Thus, Priority-Hints requires less information than Priority-DBMIN.

(6) Finally, note that while the information supplied to the buffer manager is similar in Priority-Hints, DB2, and Starburst, Priority-Hints differs from the DB2 and Starburst buffer management algorithms in two significant respects. Firstly, it groups buffers on a per-transaction basis in order to allow a local replacement search strategy. Secondly, unfixed buffers are arranged in MRU order in Priority-Hints, unlike in DB2 or Starburst. These two factors may have a significant impact on performance, as will become clear in Section 4.

3. MODELING A PRIORITY-ORIENTED DBMS

In this section, we describe our performance model of a priority-oriented DBMS. The model, which we implemented using the DeNet simulation language [Livn89], consists of five components: the database itself; a *Source*, which generates the workload of the system; a *Transaction Manager*, which models the execution behavior of transactions; a *Resource Manager*, which models the CPU, I/O, and buffer resources of the system; and a *Concurrency Control Manager*, which implements the details of a particular concurrency control algorithm. Since we will be using workloads where concurrency control is not an issue, we will not discuss the Concurrency Control Manager further. (As described in Section 4, our workloads consist either of read-only transactions with data sharing, or updates without data sharing.) In most respects, our model is similar to the model described in [Care89]. Therefore, we describe its components very briefly here; the reader is referred to [Care89] for more details.

3.1. Modeling the Database

The database is modeled as a collection of *relations*. Each relation in turn is modeled as a collection of pages. Indices (clustered or unclustered B+ Trees) on the base relations are included in the database model. The parameters for the database model are summarized in Table 3.1.

Parameter	Meaning
NumRelations	Number of relations in database
$RelSize_i$	Number of pages in relation i
$Indexed_i$	Whether relation i has an index
$IndexType_i$	Type of index (clustered/nonclustered)
$Fanout_i$	Fanout of internal nodes of index

Table 3.1: Database Model Parameters.

3.2. The Source Module

The Source module is the component responsible for modeling the workload of the DBMS. Table 3.2 summarizes the key parameters of the workload model. A transaction may belong to any one of *NumClasses* classes, and it may have any one of *NumPriorities* priority levels. The model is that of an open system, and transactions of each $<class_i, priority_j>$ combination arrive at the system in a Poisson process with a mean arrival rate of $ArrRate_{ij}$. Transactions can be single-relation selects, single-relation select-updates, or two-relation select-joins; the type of a transaction of class i and priority j is controlled by $TransType_{ij}$. Selections can be performed via sequential scans or index scans, and we model three join methods: nested-loops joins, classic hash joins, and index joins.

For each transaction type (selection, join, or update) of a particular priority level, an execution plan is provided in the form of a set of parameters. For selections, the access path and the mean selectivity are provided as parameters. The actual selectivity is varied uniformly over the range $[Selectivity_{ijk}/2,\ 3*Selectivity_{ijk}/2]$. For select-joins, the join method and the inner and outer relations are provided in addition to the selection parameters. For select-updates, the probability of updating a page is specified as the parameter $UpdateProb_{ijk}$. Finally, times spent at the CPU for processing or updating a page are uniformly distributed: the CPU time per data page of relation k varies uniformly over the range $[DataPageCPU_{ijk}/2,\ 3*DataPageCPU_{ijk}/2]$, and similar distributions are used for $IndexPageCPU_{ijk}$ and $UpdateCPU_{ijk}$. Given a plan, the Source module generates a list of page accesses that models the sequence in which pages will be accessed by the transaction.

3.3. The Transaction Manager Module

The Transaction Manager is responsible for accepting transactions from the Source and modeling their execution. For each page accessed by the transaction, the Transaction Manager sends a read (or write) request to the Resource Manager; the Resource Manager informs the Transaction Manager when the request is completed. The Resource Manager also

Parameter	Meaning
	Overall Arrival Pattern Parameters
NumClasses	Number of transaction classes
NumPriorities	Number of transaction priority levels
	Per (Class, Priority) Parameters ($1 \le i \le NumClasses$, $1 \le j \le NumPriorities$)
$ArrRate_{ij}$	Mean exponential arrival rates of queries
$TransType_{ij}$	Type of transaction, e.g., select or select-join
$JoinMethod_{ij}$	Join algorithm used
$Outer_{ij}$	Outer relation
$Inner_{ij}$	Inner relation
$AccessPath_{ijk}$	Access path used to access *k*th relation
$Selectivity_{ijk}$	Fraction of *k*th relation selected
$UpdateProb_{ijk}$	Probability of update of an accessed page of *k*th relation
$IndexPageCPU_{ijk}$	CPU time for processing a page of *k*th relation's index
$DataPageCPU_{ijk}$	CPU time for processing a page of *k*th relation
$UpdateCPU_{ijk}$	CPU time for page update of *k*th relation

Table 3.2: Workload Model Parameters.

informs the Transaction Manager when a transaction is suspended or reactivated. When the Resource Manager decides to reactivate a suspended transaction, the Transaction Manager ensures that the reactivated transaction resumes execution at the point where it was suspended.

3.4. The Resource Manager Module

The Resource Manager controls the physical resources of the DBMS, including the CPU, the disk, and the buffer pool in main memory. Three versions of the Resource Manager have been implemented, supporting the Priority-LRU, Priority-DBMIN, and Priority-Hints buffer management algorithms, respectively.[8] The parameters of the Resource Manager are summarized in Table 3.3.

CPU and Disk Models

The DBMS has *NumCPUs* CPUs and a single priority queue for outstanding CPU requests. The actual CPU where a request is processed is selected at random from among the idle CPUs, if any. The length of each CPU request from a transaction is its per-page CPU processing time; each transaction voluntarily gives up the CPU after processing or updating one page, as in the priority-based round robin CPU scheduling scheme described in [Care89]. There are *NumDisks* disks in the system, with requests at each disk being priority-scheduled according to the prioritized elevator algorithm [Care89]. We model the data as being uniformly distributed across all disks and across all tracks within a disk. The total time required to complete a disk access is computed as the sum of its seek time, rotational latency, and transfer time components. As in [Bitt88, Care89], there is a square root relationship relating seek time to seek distance, and the rotational latency and transfer time are together modeled as a single parameter called *DiskConst*.

Buffer Manager Models

The Buffer Manager component of the resource manager encapsulates the details of the buffer management scheme employed. The number of page frames in the buffer pool is specified as *NumBuffers*. The asynchronous write engine is

Parameter	Meaning
NumCPUs	Number of CPUs
NumDisks	Number of disks
NumTracks	Number of tracks per disk
DiskConst	Sum of rotational and transfer delays
SeekFactor	Factor relating seek time to seek distance
NumBuffers	Number of buffer frames in buffer pool
EngineSleepTime	Period after which write engine wakes up
FlushThreshold	Threshold used to choose pages that are "sufficiently old" to be written to disk.

Table 3.3: Parameters of the Resource Manager.

[8]In addition, Resource Managers supporting Global-LRU and Global-Hints, two non-priority buffer management algorithms, have also been implemented. The Global-Hints algorithm is introduced in Section 4.

activated automatically after *EngineSleepTime* seconds, and *FlushThreshold* is the threshold used to choose "sufficiently old" dirty pages to be flushed to disk. A separate buffer manager component has been implemented for each buffer management algorithm studied.

4. EXPERIMENTS AND RESULTS

In this section, we present performance results for the priority buffer management algorithms described earlier. In [Care89] it was shown that results for two priority levels can be generalized to multiple priority levels, so we consider just two priority levels: "low" versus "high" priority. In addition, our workload consists of three types of single-query transactions: "looping", "random reaccess (RR)", and "scanning" transactions. As their names suggest, looping transactions (such as nested-loops joins) reaccess some of their pages sequentially a number of times, RR transactions (such as hash joins) randomly reaccess some of their pages, and scanning transactions (such as clustered-index selections) touch each page just once. Looping transactions and scanning transactions represent two ends of the spectrum of buffer access characteristics typical in relational databases, so we will concentrate mainly on these two types of transactions. In order to study how our algorithm fares in the middle of the spectrum, we also present an experiment where the workload consists of RR transactions alone.

4.1. Performance Metrics

As discussed earlier, we use an open queuing system to model the DBMS. Our primary performance metric will be the average response time ratio (RTRatio) for transactions at each priority level. We define the RTRatio of a transaction as the ratio of the actual response time of the transaction to its estimated response time on an unloaded system with an infinitely large buffer pool. A transaction's response time is computed by subtracting the time at which the transaction commits from the time at which it was submitted to the DBMS. The response time of the transaction in an unloaded system is estimated by summing the CPU requirements associated with the page accesses of the transaction and by assuming one I/O per distinct page referenced by the transaction.[9] That is, only one I/O is assumed for a page, whether it is touched just once by a transaction or accessed repeatedly. The RTRatio of a transaction, then, reflects the effects of the finite size of the buffer pool and the presence of competing transactions on the response time of the transaction. As the load on the system increases, contention for buffers causes increased I/O (and an increase in the time spent waiting outside the system) while contention for disks and CPUs causes increased disk and CPU waiting times. These factors tend to increase the RTRatio of transactions. On the other hand, if there is significant data sharing, the RTRatio of a transaction would tend to be reduced because part of the transaction's read and write sets would already be in main memory. When the workload consists of a mix of transactions with different data access patterns and different sizes, the RTRatio provides a performance measure that is equally valid for all transactions, independent of the transaction mix.

[9]The cost of writing dirty data to disk is not included in this sum, since the asynchronous write engine operates independently of transactions.

In all the experiments described here, low-priority transactions are assumed to be running in the background, and we investigate the performance impact of the arrivals of foreground high-priority transactions on the system. The arrival rate of high-priority transactions is thus varied while keeping the arrival rate of low-priority transactions fixed. As shown in [Care89], a priority-oriented DBMS can remain stable for high-priority transactions long after the combined arrival rate has become high enough to make the system unstable for low-priority transactions. Consequently, we present response time results for low-priority and high-priority transactions separately for each experiment. Each simulation was run long enough so that the 95% confidence intervals of the RTRatios of the high-priority transactions were within 10% of the mean.

4.2. Base Parameter Settings

We first present the parameters that were used in our base experiment. As subsequent experiments are discussed, we describe the variations in the parameters for each experiment. The workload-independent parameters and the mix of transactions for each priority level in the base experiment are listed in Table 4.1. Details of the parameters for each type of transaction are presented in Table 4.2.

Workload-Independent Parameters

The database is modeled as a collection of 50 relations. We use five different relation sizes in our experiments — 1000 pages, 500 pages, 5 pages, 4 pages, and 3 pages — with the database containing 10 relations of each size. The 1000-page and 500-page relations each have a clustered index available, while the smaller relations are not indexed. There are four CPUs and four disks in the system. Each disk has 1000 tracks, and the sum of the rotational latency and the transfer time per disk access is 15 milliseconds. The prioritized elevator disk scheduler has two priority queues, one for each priority level. The factor relating seek distance to seek time is 0.6 milliseconds, so the expected disk access time is between 15

Parameter	**Setting**
NumRelations	50
RelSize$_i$	1000 pages, 500 pages, 5 pages, 4 pages, 3 pages (10 relations of each size)
Indexed$_i$	YES (1000-page & 500-page relations) NO (3- , 4- , and 5-page relations)
IndexType$_i$	Clustered (1000-page & 500-page relations)
Fanout$_i$	20 (1000-page & 500-page relations)
NumCPUs	4
NumDisks	4
NumTracks	1000
DiskConst	15 ms
SeekFactor	0.6 ms
NumBuffers	50
High-Priority Workload Mix	50% looping, 50% scanning
High-Priority Arrival Rate	0-15 trans/sec
Low-Priority Workload Mix	50% looping, 50% scanning
Low-Priority Arrival Rate	5 trans/sec

Table 4.1: Base Parameter Settings.

and 30 milliseconds.

As stated in Section 1, the operating region of greatest interest to us is when the combined buffer requirements of all transactions exceeds the capacity of the buffer pool. In order to simulate the behavior of the system in this region of operation without incurring excessive simulation costs, we kept the buffer pool relatively small in our experiments. Thus, there are 50 buffer frames in the buffer pool in the base experiment. One point that should be noted here is that, from a performance perspective, it is not the actual size of the buffer pool that is most significant. Instead, two ratios are more important: the ratio of the combined buffering requirements of concurrent transactions to the size of the buffer pool[10], and the ratio of the size of the buffer pool to the size of the database. For this study, we vary the first of these ratios by varying the arrival rate of high-priority transactions in all our experiments. We study the effects of varying the second ratio in Experiment 2 by changing the database size.

Workload Parameter Settings

The workload for the base experiment consists of two types of transactions. Looping transactions consist of select-joins, with the result of a selection using a clustered index on a 500-page outer relation being joined to a smaller inner relation. The selectivity of the outer relation selection varies uniformly between 0.5% and 1.5%. The inner relation is chosen uniformly from among the 30 small relations of sizes between 3 and 5 pages. *Page Accesses* is the expected number of page accesses for the transaction, with repeated references being counted as one access each time.[11] Scanning transactions

Parameter	**Looping**	**Scanning**
TransType	Select-join	Scan
JoinMethod	Nested Loops	-
Outer RelSize	500-page	1000-page
Inner RelSize	3-5 pages	-
Outer AccessPath	Clustered Index Scan	Clustered Index Scan
Inner AccessPath	Sequential Scan	-
Outer Selectivity	1%	1%
IndexPageCPU	4 ms	4 ms
DataPageCPU	4 ms	4 ms
Page Accesses	43	13
Locality Set Sizes (index, outer, inner)	1, 1, 3-5	1, 1
Replacement Policies	MRU, MRU, MRU	MRU, MRU, -
Fixing Requirements	3	2

Table 4.2: Workload Parameter Settings.

[10]Here the term "buffering requirements" of a transaction refers to the optimum number of buffers needed by the transaction.

[11]For example, for a 1% select of a 500-page outer relation (using an index with three levels) followed by a nested-loops join with a 4-page inner relation, 43 pages are accessed: 3 to traverse the index, and then 20 pairs of (outer, inner) pages. In our model of a nested-loops join, we unfix the outer page when we unfix each inner page in order to have frequent suspension-safe points (see Section 2.1). This is why there are 20 (outer,inner) page pairs, with each of five outer pages looping over the set of four inner pages.

are clustered-index scans on a 1000-page relation, with the selectivity varying uniformly between 0.5% and 1.5%.[12] For each relation accessed by a transaction, the actual relation accessed is chosen uniformly from among the 10 relations of that size. The locality set sizes and replacement policies for Priority-DBMIN are also listed in Table 4.2, as are the fixing requirements for the two types of transactions. Note that the pages of the inner relations in both LS and IR transactions will be "favored" in the Priority-Hints model. The parameters for the base experiment were chosen to provide a moderate background load. (In the absence of any high-priority transactions, and with a low-priority arrival rate of 5 transactions/second, the disk utilization was in the range of 40%-50% for the different buffer management algorithms and the CPU utilization was below 20%.)

4.3. Experiment 0: Buffer Management Without Priority

We precede the description of the base experiment by the analysis of an experiment in which priority does not affect buffer management decisions. This will help us to separate the effects of buffer management algorithms *per se* on system performance from the impact of using priority in subsequent experiments. The workload in this experiment is the same as that described in Tables 4.1 and 4.2 (except that foreground and background transactions have the same priority). In Figure 4.1, we show the RTRatios of the foreground transactions when Priority-LRU, Priority-DBMIN, and Priority-Hints are used.[13] In order to understand the behavior of Priority-Hints relative to Priority-LRU, we also present the results for an algorithm we call "Global-Hints." Global-Hints differs from Priority-Hints in that when a buffer miss occurs and there are no free pages, the buffer manager searches the buffer pool for the globally most-recently-unfixed page and chooses it as the replacement victim. In contrast, under the same conditions in Priority-Hints, the buffer manager attempts to find unfixed pages of lower priority to choose as replacement victims, as explained in Section 2. Finding no pages of lower priority (since all transactions have the same priority in this experiment), Priority-Hints chooses the most-recently-unfixed page *of the requesting transaction* as the replacement victim. Thus, Priority-Hints' replacement policy is local, while Global-Hints' replacement policy is global.

In Figure 4.1, we see that Priority-DBMIN provides the best performance. The performance of Priority-Hints is close to that of Priority-DBMIN over a wide range of arrival rates, although with Priority-Hints, the system saturates at a lower arrival rate than with Priority-DBMIN. Finally, Global-Hints provides better performance than Priority-LRU, but is significantly worse than Priority-Hints.

Priority-DBMIN provides better performance than Priority-Hints because the admission control policy of Priority-DBMIN uses its knowledge of the optimum number of buffers required for each transaction, while Priority-Hints' admission control policy cannot distinguish between the buffer requirements of looping transactions and those of scanning tran-

[12]In Experiment 4, where we study the impact of updates on priority buffer management, non-key attributes of the tuples selected by the clustered index scans are updated. In all other experiments, the workload consists of read-only transactions, since we are interested here primarily in buffering issues rather than concurrency control.

[13]The RTRatios of the background transactions are (as one would expect) almost exactly the same as the RTRatios of the foreground transactions, and are thus not presented here.

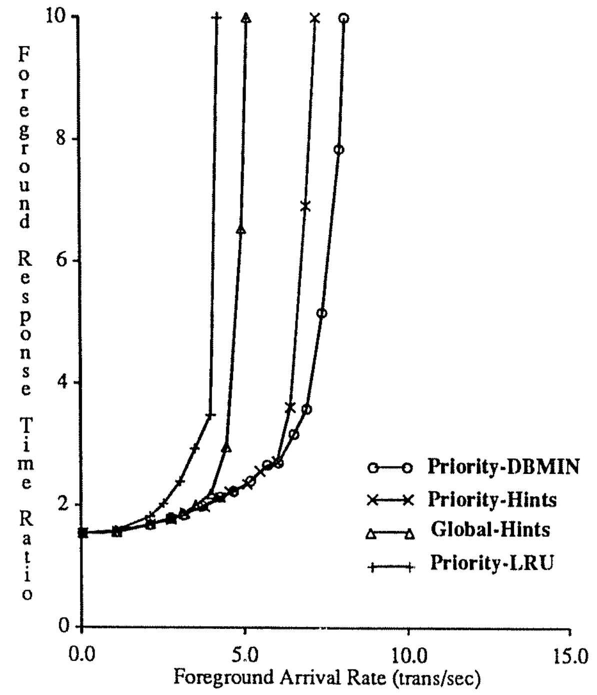

Figure 4.1: No Priority.

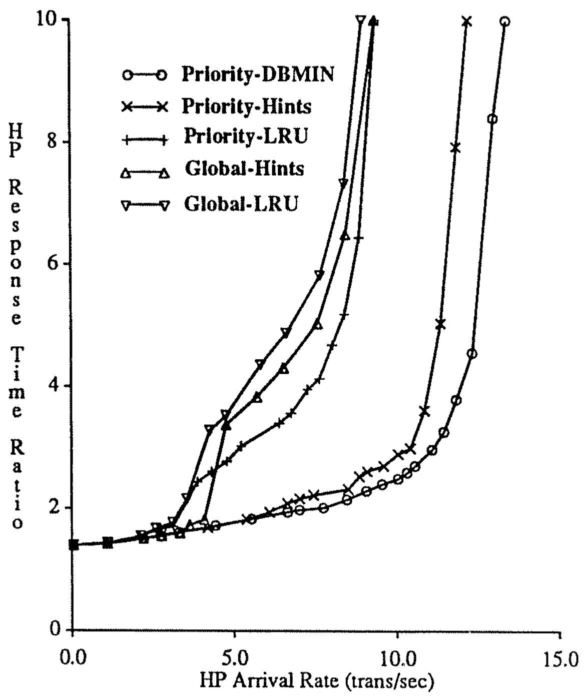

Figure 4.2: High Priority.
(Base Experiment)

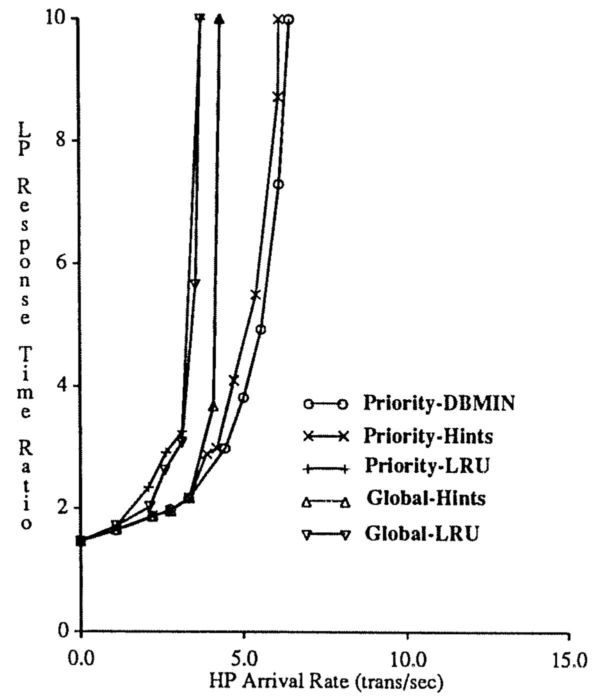

Figure 4.3: Low Priority.
(Base Experiment)

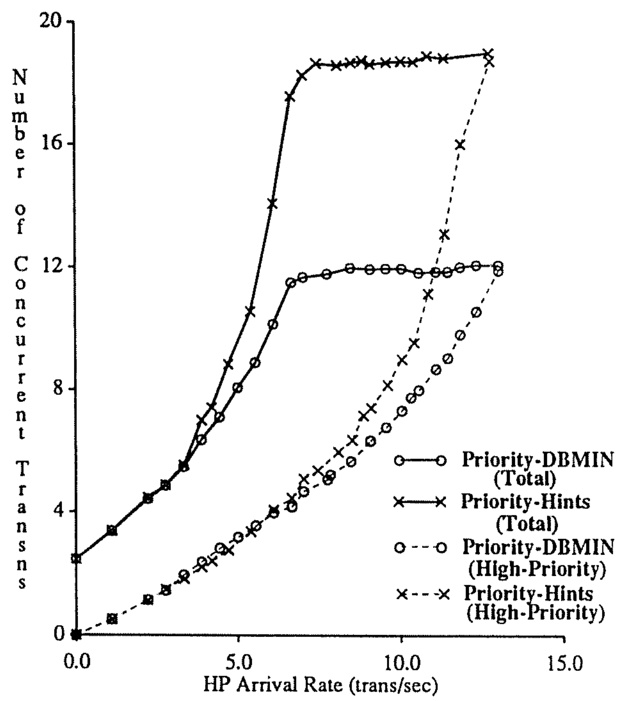

Figure 4.4: Number of Concurrent Transactions.
(Base Experiment)

sactions. Consequently, Priority-Hints allows looping transactions to enter the system even when their loops cannot be guaranteed to fit in the buffer pool. This results in a higher buffer miss ratio for Priority-Hints than for Priority-DBMIN, and causes the system to become unstable at a lower arrival rate.

When we move to the Global-Hints algorithm from Priority-Hints, the local search for MRU replacement victims is replaced by a global search. This change results in a significant performance degradation for the following reason: In Priority-Hints, for looping transactions, once a transaction is able to obtain enough buffers to keep its loop (the inner relation in the nested-loops join) in memory, it proceeds quickly since it never has to give up any buffers. (Recall that no transaction can steal buffers from any other transaction in Priority-Hints for this workload.) In contrast, Global-Hints steals buffers indiscriminately from all transactions, often depriving looping transactions that have their entire working set in memory of some of their favored buffers. This causes a significant increase in disk activity for Global-Hints as compared to Priority-Hints. As a result, the system becomes unstable for Global-Hints at a foreground arrival rate of approximately 5 transactions/second, while Priority-Hints keeps the system stable until an arrival rate of about 7.5 transactions/second.

Finally, the difference between the curves for Global-Hints and Priority-LRU is caused by two features of Global-Hints. Firstly, MRU is a better search strategy for buffer replacement than LRU when the workload contains looping transactions. Secondly, Global-Hints frees unfavored pages as soon as it unfixes them; Priority-LRU does not. This results in favored pages being chosen as replacement victims more frequently in Priority-LRU than in Global-Hints.

This experiment shows that even in the absence of priority, Priority-Hints' local MRU replacement strategy for favored pages provides significantly better performance than Priority-LRU for our base workload, and matches the performance of Priority-DBMIN over a wide range of arrival rates. It also isolates the relative impact of the following buffer management features: admission control, which causes the difference between Priority-DBMIN and Priority-Hints; local vs. global search for replacement victims, which causes the difference between Priority-Hints and Global-Hints; and the use of MRU vs. LRU search strategies in a looping workload, which is the major factor causing the gap between the curves for Global-MRU and Priority-LRU. We can now begin our investigate the performance of the system when the workload consists of transactions of different priority levels.

4.4. Experiment 1: The Base Experiment

In this experiment, we study the impact of using priority in buffer management for our base workload. Figure 4.2 shows the RTRatios for high-priority transactions for five buffer management algorithms: Priority-DBMIN, Priority-Hints, Priority-LRU, Global-Hints, and Global-LRU. RTRatios for low-priority transactions are shown in Figure 4.3. We explain the results of the base experiment in detail in order to provide insights into the important issues involved; this will allow us to present the results of subsequent experiments more briefly.

Comparing the curves for each algorithm in Figure 4.1 with the corresponding curves in Figure 4.2, we see that we have achieved the primary goal of priority scheduling, which is to provide a higher level of performance for high-priority transactions. For example, the system remains stable for a foreground arrival rate of up to 12 transactions/second in Figure 4.2 for Priority-Hints, while the system saturates at a foreground arrival rate of about 6 transactions/second in Figure 4.1 for the same algorithm. Of course, there is a corresponding price which is paid by low-priority transactions, as is made clear by comparing Figures 4.1 and 4.3. A secondary goal of priority scheduling is to minimize the penalty imposed on low-priority transactions; distinctions between the different algorithms in this respect will become clear as we describe subsequent experiments.

From Figure 4.2, we see that the behavior of Priority-Hints for high-priority transactions is very close to that of Priority-DBMIN, and both are superior to the other three algorithms. An interesting feature of the behavior of these two algorithms is the tradeoff between the time spent by transactions waiting outside the system in Priority-DBMIN and the time spent inside the system competing for resources in Priority-Hints. Priority-DBMIN's conservative admission policy causes the transactions' mean time spent waiting outside the system to increase more as the load on the system is increased than does Priority-Hints' liberal admission policy. However, since there are more transactions within the system in Priority-Hints than in Priority-DBMIN, the buffer miss ratios and the mean waiting times at the disks are higher for Priority-Hints than for Priority-DBMIN.[14] This tradeoff will be referred to again in the following sections, where we will refer to it as the "conservative-liberal (C-L)" tradeoff. In Figure 4.4, we present the mean number of transactions (both total and high-priority) that are allowed to run concurrently by the two algorithms. Figure 4.5 shows the mean normalized DiskTime and the mean normalized OutWaitTime for high-priority transactions for the two algorithms. DiskTime is the time spent by a transaction at the disks, including the actual I/O service time and time spent waiting for disk service; OutWaitTime is the time spent by a transaction waiting outside the system. DiskTime and OutWaitTime for a transaction are normalized by dividing each of them by the expected response time of the transaction in an unloaded system.

Figure 4.4 indicates that Priority-Hints allows up to 18 high-priority transactions into the system, while Priority-DBMIN limits the number of concurrent high-priority transactions to 12. Figure 4.5 shows the consequences of this. The DiskTime curves reflect the relative disk contention in the two algorithms, and Priority-DBMIN is the clear winner in limiting disk contention; both the average buffer miss ratio and the average disk waiting times per transaction are higher in Priority-Hints than in Priority-DBMIN. In contrast, Priority-Hints is the winner in limiting OutWaitTime up to an arrival rate of up to 11 transactions/second. As the load is increased beyond this, however, the disk utilization nears 100% for Priority-Hints, causing the system to saturate and Priority-Hints' OutWaitTime to exceed that of Priority-DBMIN. Priority-DBMIN's conservative admission control policy enables it to keep the system stable for arrival rates of up to 13 transactions/second.

Figure 4.4 also reveals an important point about the range of operation of greatest interest to us. There is a gap between the total number of concurrent transactions and the number of high-priority transactions for most arrival rates

[14]Contention for the CPU is not a significant factor in this experiment.

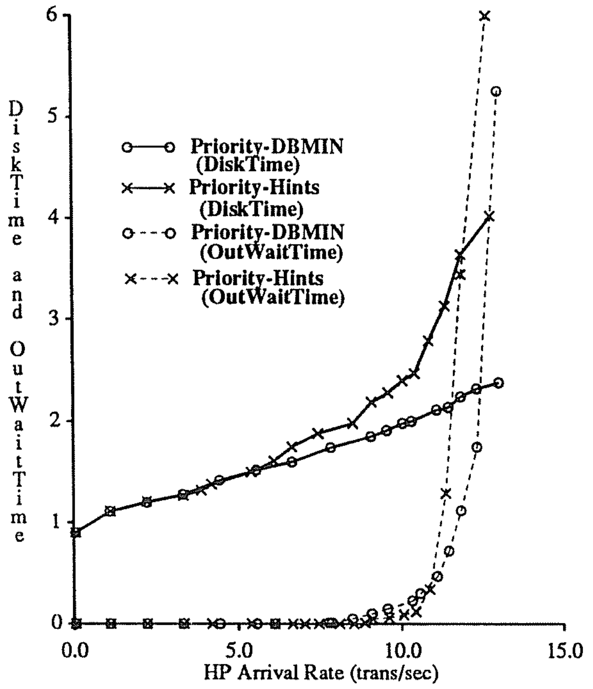

Figure 4.5: DiskTime and OutWaitTime. (Base Experiment)

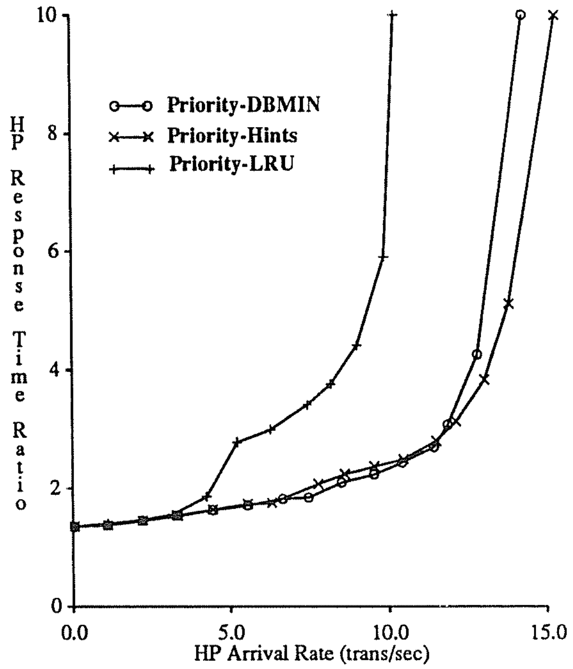

Figure 4.6: High Priority. (Medium Data Sharing)

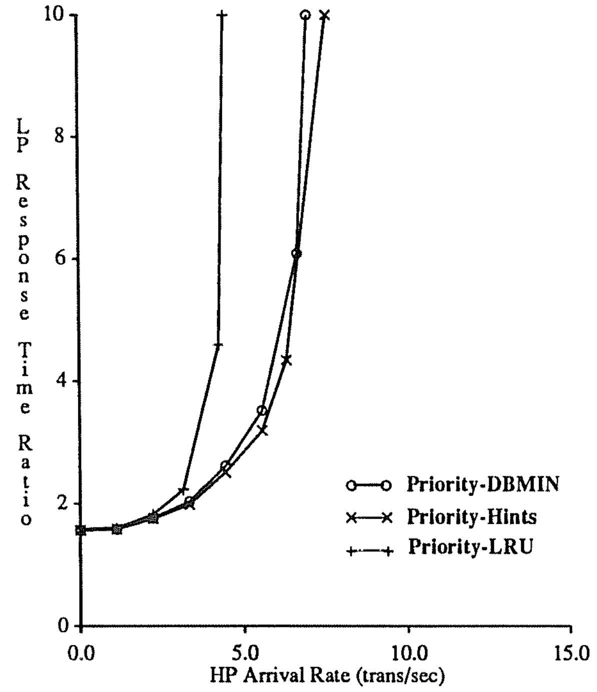

Figure 4.7: Low Priority. (Medium Data Sharing)

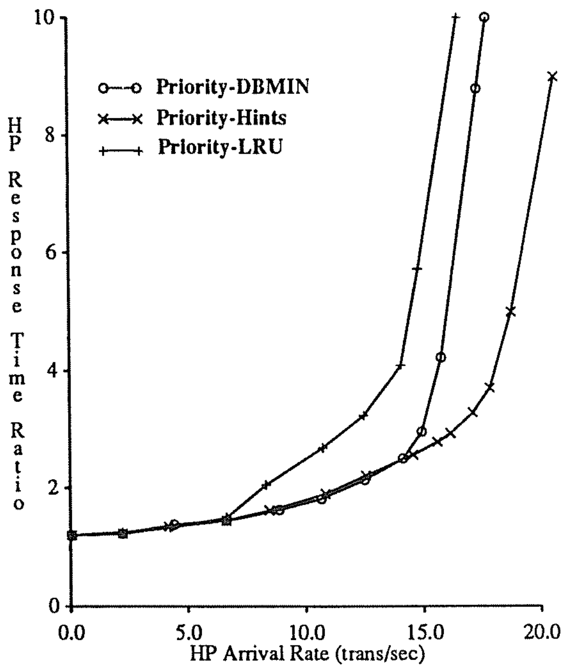

Figure 4.8: High Priority. (High Data Sharing)

shown; this gap corresponds to the number of low-priority transactions running in the system. When the curves for the total number of transactions in the system flatten out, the buffer pool has become fully utilized, but Figure 4.4 shows that there are still significant numbers of low-priority transactions running in the system. It should be clear that priority-based buffer replacement policies will be most useful in this range of operation, since buffers owned by low-priority transactions can be stolen by high-priority transactions. In Figure 4.4 we also see that as the arrival rate of high-priority transactions increases, low-priority transactions are gradually displaced by high-priority transactions (due to the use of priority-based admission control policies) until finally only high-priority transactions remain active. Thus, both priority-based admission control and priority-based buffer replacement have an important role in determining performance over a fairly wide range of arrival rates.

Figure 4.2 shows us that at low loads, all the algorithms provide similar levels of performance to high-priority transactions. As the high-priority load increases, the curves for Global-LRU, Global-Hints, and Priority-LRU soon branch away from the curves for Priority-Hints and Priority-DBMIN. Global-LRU performs worst of all: it does not distinguish between transactions of high and low priorities, and it uses the LRU criterion for replacement. As for Global-Hints, it actually performs better than Priority-LRU for a small range of arrival rates (from about 3 transactions/second to 4 transactions/second). In this range, Global-Hints' use of MRU (and the fact that it frees unfavored pages quickly) result in lower response times for low priority transactions than Priority-LRU provides. As the arrival rate of high-priority tasks is still fairly low, there are enough free buffers available that high-priority transactions' buffers are not stolen by low-priority transactions even though Global-Hints' ignores priority in its replacement policy. Priority-LRU is already unstable for low-priority transactions, so it sometimes steals buffers from high-priority transactions since few low-priority transactions have unfixed buffers left at this load. As the load is increased, however, Priority-LRU's protection of high-priority buffers begins to dominate the effects of Global-Hints' use of MRU, since more of the buffers are now owned by high-priority transactions. The RTRatio of high-priority transactions remains lower for Priority-LRU than for Global-Hints until the system becomes unstable for both algorithms at a high-priority arrival rate of approximately 8 transactions/second.

In Figure 4.3, the curves for Priority-Hints and Priority-DBMIN are fairly close. This is because the criteria used for suspending low-priority transactions differ in these two algorithms, making the C-L tradeoff more even for low-priority transactions than it is for high-priority transactions. (High-priority transactions cannot be suspended in either algorithm.) Priority-DBMIN suspends a low-priority transaction immediately when one of its unfixed buffers is required by a high-priority transaction. In contrast, Priority-Hints does not suspend low-priority transactions as frequently as Priority-DBMIN does. It allows them to continue execution as long as their "fixing requirements" can be satisfied; of course, this increases the low-priority load on the disks. As long as there is sufficient disk capacity in the system to handle this increased low-priority load in Priority-Hints, the two algorithms provide similar performance for low-priority transactions.

Figure 4.3 also shows that there is very little difference in the performance provided by the LRU algorithms (Global-LRU and Priority-LRU) for low-priority transactions. Priority-LRU steals buffers from low-priority transactions in preference to depriving high-priority transactions of their buffers, so one might expect Priority-LRU to provide worse performance for low-priority transactions. Recall, however, that the CPUs and the disks use priority scheduling in these experiments; also, in the range of arrival rates for which the system is stable for low-priority transactions, there are relatively few high-priority transactions in the system. The globally least-recently-used buffer is therefore quite likely to belong to a low-priority transaction rather than to a high-priority transaction. This is why the curves for Priority-LRU and Global-LRU are so close to each other in Figure 4.3.

Low-priority transactions perform better under the Global-Hints algorithm than under the two LRU algorithms in Figure 4.3 because their buffer miss ratios are lower as a consequence of the use of MRU. The reason that Global-Hints performs worse than Priority-Hints there, even for low-priority transactions, is again related to the use of priority at the CPUs and at the disks. As the arrival rate of high-priority transactions increases, Global-Hints hurts high-priority transactions more than Priority-Hints does (since Global-Hints ignores priority in choosing replacement victims). Consequently, more and more of the system's disk capacity is used to satisfy high-priority transactions in Global-Hints. This makes the disk waiting times of low-priority transactions higher, causing the system to become unstable for low-priority transactions at a lower load in Global-Hints than in Priority-Hints.

The base experiment confirms the result of [Care89] for our mixed workload: the use of priority in buffer management is a clear win (independent of the algorithm) if the response time of high-priority transactions is the main criterion of system performance. However, the conclusions are more mixed for low-priority transactions: their performance may be worse for some priority-based buffer management algorithms (e.g., Priority-LRU) than for algorithms that do not consider priority in replacement decisions (Global-Hints). The base experiment also shows that for both low-priority and high-priority transactions, the performance provided by Priority-Hints is significantly better than the performance provided by Priority-LRU, and that Priority-Hints performs almost as well as Priority-DBMIN for both priority levels. In subsequent experiments, we limit ourselves to showing the relative behavior of the three priority-based algorithms, and do not include further results for Global-LRU or Global-Hints.

4.5. Experiment 2: Increasing the Relative Buffer Pool Size

In this experiment, we reduce the size of the database while keeping all other parameters the same as in the base experiment. Thus, the ratio of the size of the buffer pool to the size of the database is higher in this experiment than in the base experiment. Increasing the relative size of the buffer pool in this manner results in increased data sharing: i.e., data brought into the buffer pool at the request of one transaction is more likely to be found in memory when it is accessed by other transactions. When data accesses are distributed uniformly over the database, the extent of data sharing is inversely proportional to the database size if all other parameters are kept fixed. In the base experiment, the database consisted of 50 relations and a total of 15,120 data pages, while the buffer pool had 50 buffers (see Table 4.1). This represents a fairly low level of data sharing (and we did not even take index pages into account when computing the sum of 15,120 pages!). We consider two levels of data-sharing in this experiment: one where there are 25 relations in the database (five relations each of 1000 pages, 500 pages, 5 pages, 4 pages, and 3 pages), and one where there are just 5 relations (one of each size). When there are 25 relations in the database, the level of data sharing is double the level of data sharing in the base experiment; when there are just 5 relations, the level of data sharing is ten times that in the base experiment.

In Figure 4.6, we present the RTRatios of high-priority transactions for the three algorithms with 25 relations in the database. Figure 4.7 shows the RTRatios of low-priority transactions for the same database. As one would expect, the performance of all three algorithms improves over their performance in the base experiment. The key difference between the trends shown in Figures 4.4 and 4.2 is the fact that Priority-Hints now provides better performance than Priority-DBMIN at high loads. As the level of data sharing increases, Priority-DBMIN's conservative admission control policy proves to be more and more harmful: it simply underutilizes resources by failing to consider the possibility of data-sharing. The same conclusion holds true for low-priority transactions, where Priority-Hints consistently performs as well as or better than Priority-DBMIN.

Figures 4.8 presents the RTRatios for high-priority transactions in the case where there are just 5 relations in the database. Priority-Hints and Priority-DBMIN provide the same level of performance for high-priority transactions until Priority-DBMIN's admission control policy causes it to block transactions unnecessarily outside the system; beyond this point, Priority-Hints is better. As for Priority-LRU, one might initially expect that all three inner relations of the nested-loops joins (a total of 12 pages) would always remain in memory, and that it should therefore perform as well as Priority-Hints. However, recall that there are now 1500 data pages (plus index pages) that are not part of inner relations. A large fraction of the page requests made to the buffer manager are for one of these 1500 pages. In Priority-Hints and Priority-DBMIN, the looping pages will remain in memory as long as there are some high-priority transactions that need them. Also, these two algorithms free unfavored pages as soon as they are unfixed, so it is quite likely that a free page will be available even when the load is high. In Priority-LRU, where transactions can steal buffers from other transactions of the same priority, and where looping pages are treated just like other pages, looping pages are frequently chosen as replacement victims. This is why the curve for Priority-LRU diverges from the other two in Figure 4.8.

This experiment shows that as the relative size of the buffer pool increases, the performance of Priority-Hints improves to a greater extent than the performance of Priority-DBMIN. This is a consequence of the latter's conservative admission control policy. Also, both Priority-Hints and Priority-DBMIN provide better performance than Priority-LRU, even under very high data sharing.

4.6. Experiment 3: Changing the Transaction Mix

In the base experiment, the high-priority workload consisted of a mix of an equal number of looping and scanning transactions. In this experiment, we vary the proportion of looping transactions in the high-priority workload while keeping all other parameters as in the base experiment. (The low-priority workload still consists of a 50% looping, 50% scanning mix.)

Figure 4.9 shows the RTRatios for high-priority transactions when the high-priority workload consists entirely of scanning transactions. RTRatios for the low-priority transactions are shown in Figure 4.10. In Figure 4.9, the curves for the three algorithms coincide almost exactly. This is not unexpected, since the high-priority workload is insensitive to buffer replacement and the admission control criteria for the three algorithms coincide for scanning transactions.[15] The interesting feature of this experiment is the relative performance of low-priority transactions, as shown in Figure 4.10: here, Priority-Hints performs far better than Priority-LRU. The reason for this is that, unlike Priority-Hints, Priority-LRU does not free scanning pages as soon as they are unfixed. High-priority transactions thus keep stealing looping pages from low-priority transactions in Priority-LRU, while the more appropriate action is to choose high-priority scanning pages as replacement victims.

Figures 4.11 and 4.12 show the RTRatios for high-priority transactions and low-priority transactions, respectively, when the high-priority workload consists entirely of looping transactions. There is now a relatively larger difference between the performance of high-priority transactions for Priority-Hints and Priority-DBMIN than in the base experiment. The C-L tradeoff for high-priority transactions favors Priority-DBMIN in this experiment, since a larger proportion of the high-priority workload now benefits from conservative admission control. However, note that the tradeoff is still quite even for transactions of low priority. This is because Priority-DBMIN now prevents more scanning low-priority transactions from entering the system than Priority-Hints does, and it also suspends them more often.

From Experiment 3, we learn that Priority-Hints tends to perform as well as or better than Priority-LRU, independent of the proportion of looping and scanning transactions. In particular, when the high-priority load is insensitive to buffer management, Priority-Hints provides much better performance than Priority-LRU for low-priority transactions. As the proportion of looping transactions in the high-priority workload increases, Priority-DBMIN's admission control policy makes it perform better than Priority-Hints for high-priority transactions, but the two algorithms provide very similar support for low-priority transactions.

4.7. Experiment 4: Low-Priority Updates

In all of the experiments described thus far, the workload consisted entirely of read-only transactions. Clearly, introducing updates into the workload results in an increased load on the disks, as dirty pages have to be written back to disk before they can be replaced in the buffer pool. As discussed in Section 1, the key issue of interest from a priority standpoint is the effect (if any) of updates of low-priority transactions on the performance of high-priority transactions. In this experiment, we investigate this issue.

We introduce the following changes to the base workload parameters for this experiment. The high-priority workload consists of 100% looping, read-only transactions. The low-priority workload consists of 100% scanning update transac-

[15]The number of buffers required to meet the fixing requirements of a scanning transaction is equal to the sum of the sizes of its locality sets; see Table 4.2.

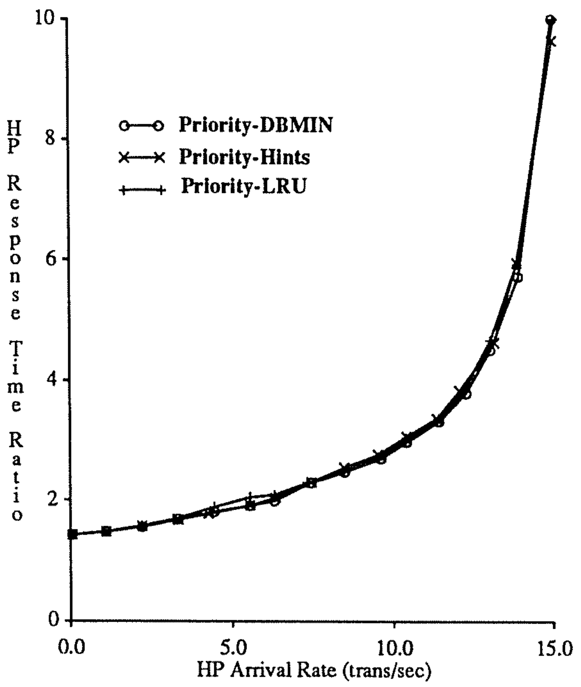

Figure 4.9: High Priority.
(HP Load = 100% scanning)

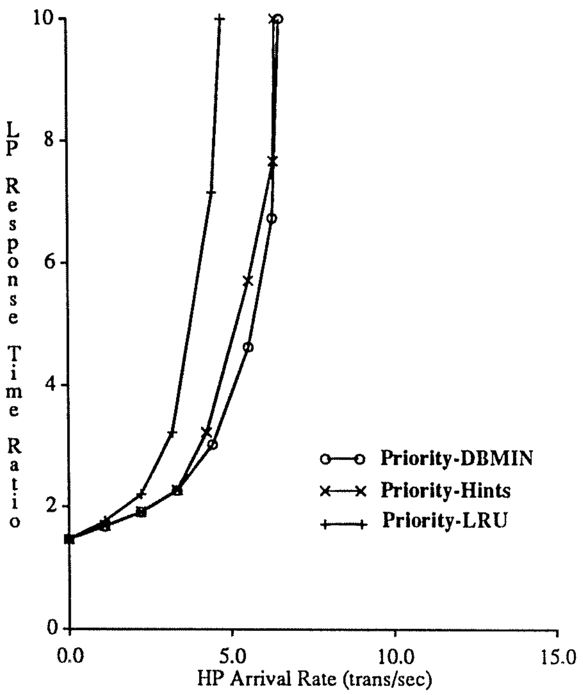

Figure 4.10: Low Priority.
(HP Load = 100% scanning)

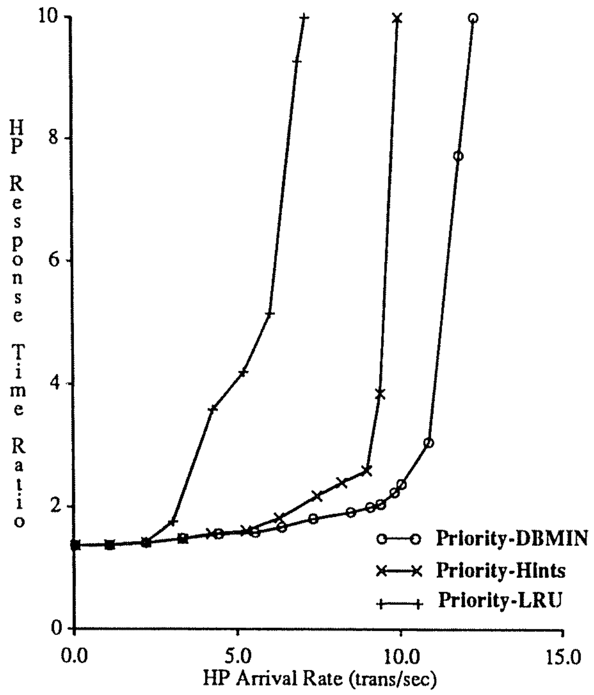

Figure 4.11: High Priority.
(HP Load = 100% looping)

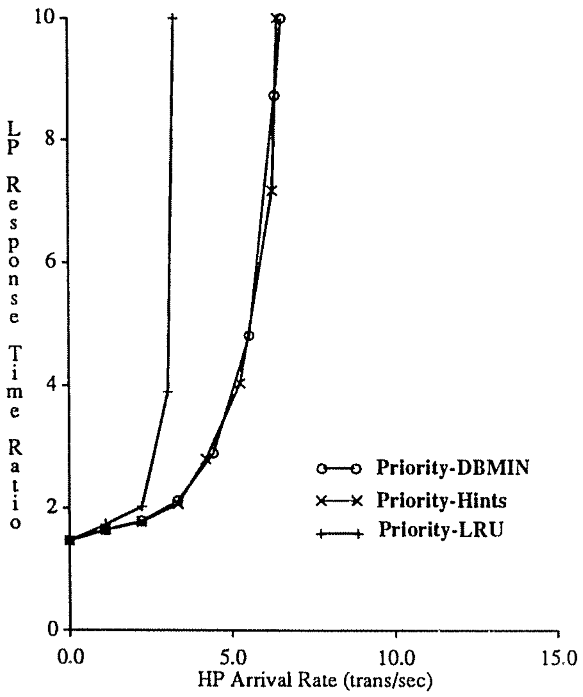

Figure 4.12: Low Priority.
(HP Load = 100% looping)

tions; a clustered-index scan is used to select tuples, and non-indexed attributes of the selected tuples are updated. Low-priority transactions arrive at a mean arrival rate of 5 transactions/second, as before. The CPU cost of updating a page is set at 1 millisecond, and the probability that a page is updated is varied from 0% to 100% in steps of 20%. *EngineSleepTime* is set at 0.5 seconds. (The results presented here are actually not affected by the *EngineSleepTime* parameter, as the asynchronous write engine is activated at every buffer miss.) *FlushThreshold* is set to zero in order to study the worst-case negative impact of low-priority updates on high-priority performance. We ensured that the write sets of concurrent transactions do not intersect in this experiment, so concurrency control is not an issue. Other parameters are the same as in the base experiment.

Figure 4.13 shows the high-priority RTRatios for the three buffer management algorithms, with the arrival rate of high-priority transactions set at 2 transactions/second, as the low-priority update probability is increased from 0% to 100%. In order to isolate the impact of the use of priority, we also present the corresponding curves (i.e., the RTRatios for the looping transactions) for the case when all transactions have the same priority. From the curves labeled "1 Priority" in Figure 4.13, we see that when all transactions have the same priority, looping transactions suffer severely when the update probability is increased in the case of Priority-LRU. Note that in this experiment, the pages that are updated are unfavored. Recall also that all three algorithms ignore dirty pages in their search for replacement victims. Priority-Hints and Priority-DBMIN place dirty unfavored pages in the dirty list as soon as they are unfixed, just as they place unfavored clean pages in the free list at unfix time. In contrast, Priority-LRU does not free any pages until transaction commit time unless all possible replacement victims are dirty, in which case a dirty page is synchronously written out to disk to prevent deadlock as explained in Section 2.5. This causes dirty unfavored pages to accumulate in the buffer pool in Priority-LRU. This is why Priority-LRU performs much worse than Priority-Hints and Priority-DBMIN in the absence of priority, while Priority-Hints performs as well as Priority-DBMIN. When priority is introduced, all three algorithms provide improved performance for the high-priority transactions. However, Priority-LRU's tendency to delay the flushing of its low-priority updates to disk causes the RTRatios of the high-priority transactions to increase significantly as the update probability increases, while Priority-Hints and Priority-DBMIN manage to keep their high-priority transactions almost immune to the presence of low-priority updates at this load.

In Figure 4.13, the high-priority arrival rate is kept fixed at 2 transactions/second. In Figure 4.14, we vary the high-priority arrival rate from 2 to 7 transactions/second, while keeping the low-priority arrival rate fixed at 5 transactions per second. The results shown are for Priority-Hints; similar trends were observed for Priority-DBMIN and for Priority-LRU. Figure 4.14 shows that as the low-priority update probability is increased, the performance of high-priority transactions is affected adversely in Priority-Hints. Note also that when the high-priority arrival rate is increased, the curves tend to flatten out as the update probability is increased. The increase in disk activity caused by the more frequent arrivals of high-priority transactions, coupled with the high-priority asynchronous writes caused by low-priority updates, causes the disks to saturate at update probabilities of about 50%. As the update probability is increased beyond this, the system gradually becomes unstable for low-priority transactions. Consequently, there are fewer low-priority transactions inside the

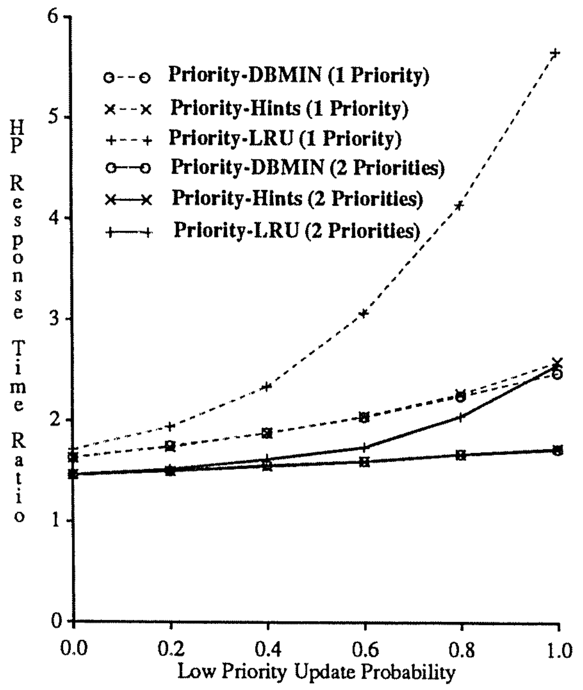

Figure 4.13: Fixed High Priority Load.
(LP Load = Updates)

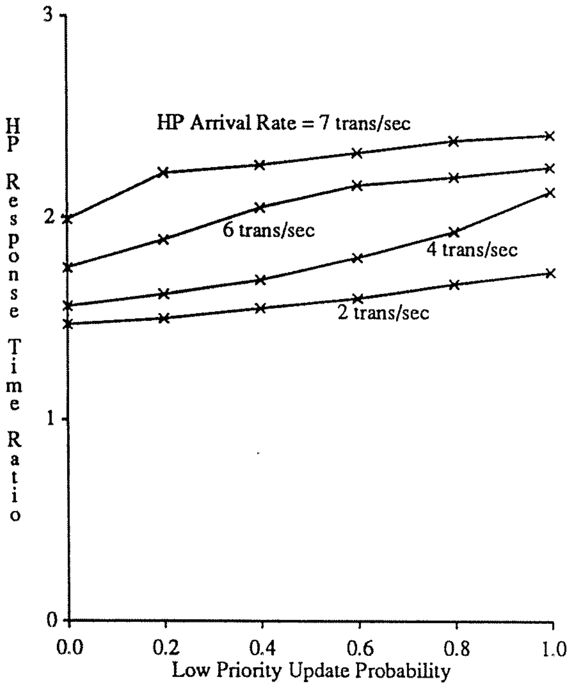

Figure 4.14: Varying High Priority Load
(Priority-Hints; LP Load = Updates)

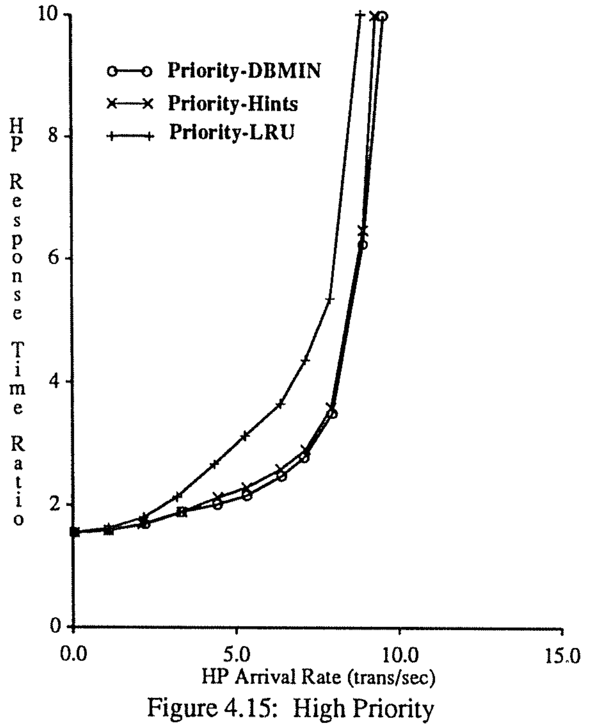

Figure 4.15: High Priority
(1 CPU, 4 disks)

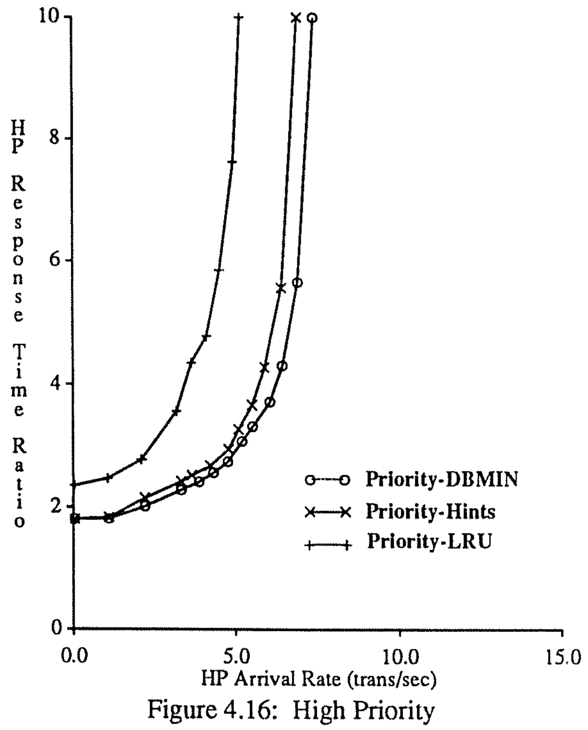

Figure 4.16: High Priority
(1 CPU, 2 disks)

system, and the effects of their updates on high-priority transactions become less significant.

In this experiment, we have shown that although low-priority updates can affect high-priority performance, their impact can be reduced significantly if Priority-Hints or Priority-DBMIN, rather than Priority-LRU, is employed.

4.8. Experiment 5: Changing System CPU and I/O Capacity

In the experiments described thus far, there have been 4 CPUs and 4 disks in the system. In this experiment, we vary the number of CPUs and the number of disks in the system while keeping the buffer pool size fixed in order to understand the impact of different resource capacities on the relative performance provided by the three buffer management algorithms. In Figure 4.15, we present the RTRatios for high-priority transactions when there is a single CPU in the system; all other parameters are set as in the base experiment. (The number of disks is still 4.) Figure 4.16 shows the high-priority transactions' RTRatios for a system with 1 CPU and 2 disks, again with all other parameters set as in the base experiment.

In Figures 4.15 and 4.16, we see that Priority-Hints and Priority-DBMIN again provide very similar levels of performance, while both these algorithms perform better than Priority-LRU. Even in Figure 4.15, where the CPU becomes the bottleneck resource, Priority-LRU performs significantly worse than the other two algorithms. In Figure 4.16, there is a wider gap between the curves for Priority-Hints and Priority-DBMIN, since the workload becomes disk-bound again. Figures 4.15 and 4.16 demonstrate that the performance differences between Priority-DBMIN and Priority-Hints occur as a consequence of higher disk activity when the latter policy is used; if the workload is CPU-bound, these two algorithms will provide similar performance. We also conducted other experiments where we varied the buffer pool size. The results (not shown here due to space considerations) confirm that Priority-Hints' performance is close to that of Priority-DBMIN, and superior to Priority-LRU, independent of the CPU, I/O and main memory capacities of the system.

4.9. Experiment 6: Random Reaccesses

In the experiments described thus far, the workloads consisted entirely of looping and scanning transactions. In this experiment, we examine the performance of Priority-Hints under a workload consisting entirely of classic hash joins [Shap86]. This workload, where a transaction reaccesses a number of its pages randomly, was chosen because its pattern of data accesses is similar to that of other types of hash-based joins and of non-clustered index selections. As stated in Section 4, we believe that such a workload represents the "middle ground" of buffer access characteristics between the extremes represented by looping and scanning transactions.

The classic hash join algorithm is one of the simplest hash-based join strategies, and consequently it is one of the simplest to model. In our model of this strategy, the inner relation is read into the buffer pool, and a hash-table on its tuple identifiers is built in the private workspace of the transaction. Then, for each selected tuple of the outer relation, this hash table is probed for matching tuples in the inner relation. Note that this variant of the classic hash join allows pages of the inner relation to be replaced in the buffer pool, though such replacements should clearly be avoided as much as possible for performance reasons. Unlike most hash-based join methods, this algorithm does not require a large block of memory to be

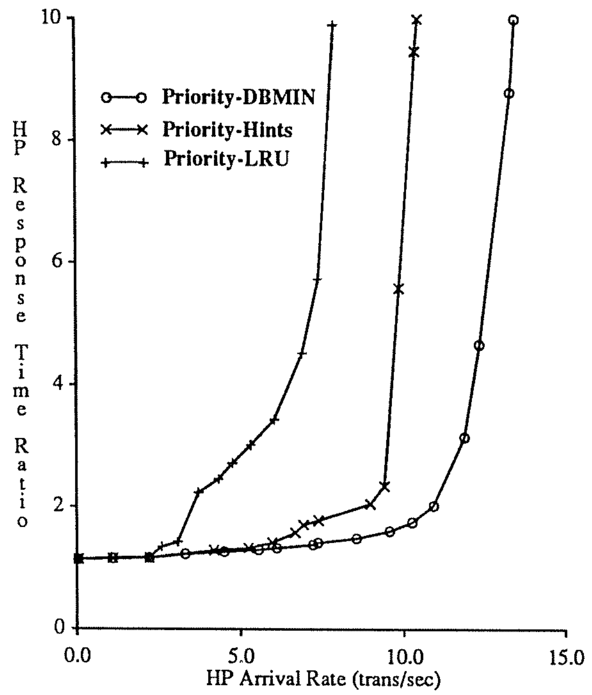

Figure 4.17: High Priority
(Random Reaccess)

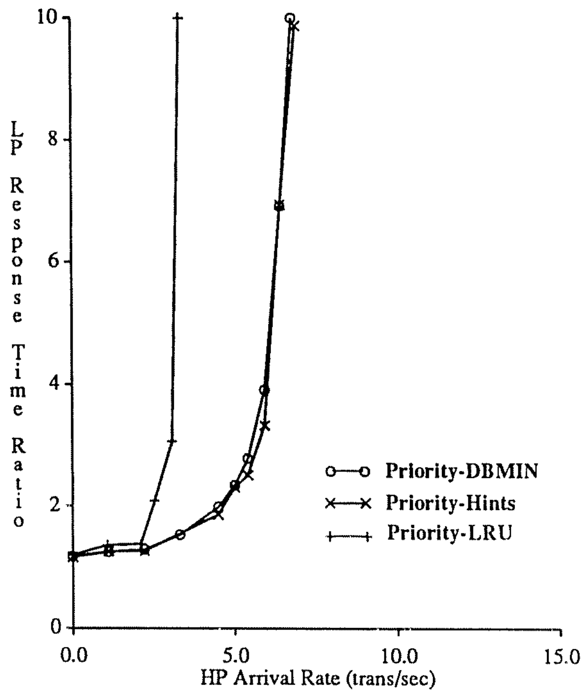

Figure 4.18: Low Priority
(Random Reaccess)

fixed in the buffer pool for the duration of the join in order to compute the join correctly. We anticipate that in priority-based database systems, where low-priority transactions should allow some of their pages to be replaced by high-priority transactions, other hash-join methods would also have to be modified to allow such replacements.

The database consists of ten 1000-page relations, each with a clustered index, and ten smaller relations, each consisting of 5 pages. The other workload-independent parameters are the same as in the base experiment. Each transaction consists of a select-join, with the result of a 0.2% clustered index selection on a 1000-page relation being joined with a 5-page inner relation using the variant of classic hashing described above. Each page of the outer relation has 10 tuples, so 20 probes of the inner relation are required for the two pages of data selected from the outer relation (we assume that each tuple of the outer relation joins with exactly one tuple of the inner relation). As before, for each relation accessed by a transaction, the actual relation is chosen uniformly from among the 10 relations of that size. *IndexPageCPU* and *DataPageCPU* are both set to 4 milliseconds, while *PageAccesses* is 48 (5 pages of the inner relation read in while building the hash table, 3 pages of outer relation index traversal, and finally 20 pairs of (outer,inner) page accesses, with each of the two selected outer pages appearing in 10 pairs). The locality set sizes for Priority-DBMIN are 1, 1, and 5 (index, outer, inner), and the replacement policies are MRU for each locality set. Clearly, the pages of the inner relation will be "favored" in the Priority-Hints algorithm, just as they were in the case of looping transactions.

The RTRatios of the high-priority transactions and the low-priority transactions are shown in Figure 4.17 and Figure 4.18, respectively. In spite of the fact that the workload in this experiment consists of transactions that randomly reaccess their pages, so the use of MRU provides no advantage, Priority-Hints provides significantly better performance than Priority-LRU. This indicates that even when the use of the MRU policy is irrelevant, the classification of pages into "favored" and "unfavored" sets by Priority-Hints enables it to provide better performance than Priority-LRU. Finally, since Priority-DBMIN keeps the entire inner relation fixed in memory for the duration of the join, it provides better performance than Priority-Hints for high-priority transactions under heavy high-priority loads. For low-priority transactions, however, the performance of Priority-Hints and Priority-DBMIN is very similar for reasons closely related to those discussed in explaining Figure 4.12 of Experiment 3.

5. CONCLUSIONS AND FUTURE WORK

In this paper, we have shown that buffer management can have a very significant impact on the performance of a priority-based database system, especially when the unpredictability of the workload forces the system to operate in regions where the total buffer requirements of the concurrent transactions exceed the system's buffer capacity. We have introduced a new buffer management algorithm, called Priority-Hints, that uses page-level information provided by the database access methods to make priority-based buffer management decisions. The performance of our new algorithm has been compared to the performance of Priority-LRU and Priority-DBMIN, two algorithms proposed earlier for priority-based buffer management. A number of performance insights have been obtained as a result of simulation experiments. Priority-Hints was shown to perform better than Priority-LRU for all of the workloads considered here. For most workloads, Priority-Hints performed almost as well as Priority-DBMIN; for some workloads with data sharing, Priority-Hints actually provided better performance than Priority-DBMIN. Even when the workload consisted of transactions of equal priority, Priority-Hints performed significantly better than Priority-LRU and almost as well as Priority-DBMIN.

These results are significant for several reasons: First, in previous studies [Chou85, Care89] it has been shown that DBMIN-like approaches to buffer management provide better performance than approaches based on simple strategies such as LRU. Still, most existing database systems continue to use LRU-based approaches because they do not require as much information as DBMIN does. Second, the type of information used by the buffer manager in Priority-Hints is already being provided to buffer managers in existing database systems [Teng84, Haas90]. Our algorithm has the advantages of both DBMIN-based approaches and LRU-based approaches; it provides good performance while requiring little information. Finally, we have also shown that Priority-Hints adapts itself dynamically as data sharing increases, while Priority-DBMIN's more static approach to buffer allocation can cause its performance to suffer in the presence of data sharing.

We plan to continue our work in the area of priority-based DBMS scheduling. First, we intend to look at some of the buffer-related issues that were not examined in this paper. For example, we plan to investigate the impact of using asynchronous read-ahead for sequential scans, and we will extend our algorithms to workloads consisting of multi-query transactions. We also intend to study the problem of concurrency control conflicts in a priority-oriented DBMS. One issue of interest here is control algorithm design; for example, one possibility is to use priority-based wound-wait across priority levels, combined with 2PL with deadlock detection within a priority level. Finally, we plan to extend our performance study to the real-time context, where the workloads contain transactions with deadlines and importance levels.

REFERENCES

[Abbo88] Abbott, R., and Garcia-Molina, H., "Scheduling Real-Time Transactions: A Performance Evaluation," *Proc. 14th VLDB Conf.*, Los Angeles, CA, Aug. 1988.

[Abbo89] Abbott, R., and Garcia-Molina, H., "Scheduling Real-Time Transactions with Disk Resident Data," *Proc. 15th VLDB Conf.*, Amsterdam, Aug. 1989.

[Bitt88] Bitton, D., and Gray, J., "Disk Shadowing," *Proc. 14th VLDB Conf.*, Los Angeles, CA, Aug. 1988.

[Care89] Carey, M. J., Jauhari, R., and Livny, M., "Priority in DBMS Resource Scheduling," *Proc. 15th VLDB Conf.*, Amsterdam, Aug. 1989.

[Chou85] Chou, H-T., and DeWitt, D., "An Evaluation of Buffer Management Strategies for Relational Database Systems," *Proc. 11th VLDB Conf.*, Stockholm, Sweden, Aug. 1985.

[Effe84] Effelsberg, W., and Haerder, T., "Principles of Database Buffer Management," *ACM Trans. on Database Sys.* 9(4), Dec. 1984.

[Haas90] Haas, L., et al, "Starburst Mid-Flight: As the Dust Clears," *IEEE Transactions on Knowledge and Data Engineering*, March 1990, to appear.

[Hari90] Haritsa, J., Carey, M. J., and Livny, M., "On Being Optimistic About Real-Time Constraints," *Proc. ACM Symp. on Principles of Database Systems (PODS)*, Nashville, TN, April 1990.

[Livn89] Livny, M., *DeNet User's Guide*, Version 1.5, Computer Sciences Dept., Univ. of Wisconsin, Madison, 1989.

[SIGM88] *SIGMOD Record* 17(1), Special Issue on Real-Time Data Base Systems, S. Son, ed., March 1988.

[Sacc86] Sacco, G.M., and Schkolnick, M., "Buffer Management in Relational Database Systems," *ACM Trans. on Database Sys.*, 11(4), Dec. 1986.

[Shap86] Shapiro, L.D., "Join Processing in Database Systems With Large Main Memories," *ACM Trans. on Database Sys.*, 11(3), Sept. 1986.

[Teng84] Teng, J., and Gumaer, R. A., "Managing IBM Database 2 buffers to maximize performance," *IBM Sys. J.* 23(2), 1984.